THE UNIVERSITY OF ALBERTA

An Interactive Classifier Programming Language

by

© **Keith Fenske**

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science.

Department of Computing Science

Edmonton, Alberta

Spring 1988

ISBN 0-315-52767-6

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Keith Fenske

TITLE OF THESIS: An Interactive Classifier Programming Language

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1988

Keith Fenske

Keith Fenske
3612 – 107 Street N.W.
Edmonton, Alberta, Canada
T6J 1B1

Tuesday, 5 January 1988

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled "An Interactive Classifier Programming Language" submitted by Keith Fenske in partial fulfillment of the requirements for the degree of Master of Science.

Dr. Jonathan Schaeffer (Supervisor): J Schaeffer

Dr. Duane A. Szafron: Duane Szafron

Dr. Jia-Huai You: Jia-Huai You

Prof. Werner B. Joerg (External):

Dr. Stan Cabay (Chairman): Stan Cabay

Tuesday, 5 January 1988

Abstract

A classifier programming environment is developed in two parts: an interactive programming language and a user classifier system. The programming language supports simple data objects (numbers, strings, lists), global variables, basic control statements ("for", "if", "repeat", "while"), pre-defined functions, and user-defined functions with local variables. The user classifier system is any program that can communicate with the programming language through special functions written in an application-dependent module.

Classifier systems are artificial intelligence applications which learn by applying genetic algorithms to bit strings. A bit string represents all possible binary numbers with a given pattern: "0" for zero bits, "1" for one bits, and "#" for any bit ("don't cares"). Knowledge is encoded into bit strings via a message list and a rule list. Messages are the current state of the classifier system; rules are legal transitions to new states. Rules are created with genetic operators for bit inversion, bit replacement ("mutation"), and bit exchange ("crossover"). Rules acquire strength when they lead to successful goal states. To limit the creation of non-functional rules, stronger rules displace weaker rules ("survival of the fittest").

The following compiler implementation issues are discussed: internal data storage, lexical tokens in the input, design of a grammar syntax, semantic actions for creating a parse tree, interpretation and execution of the parse tree (with error recovery), and construction of a communication interface between the language and a classifier system. Particular attention is paid to keeping the language small while maintaining flexibility of use.

Acknowledgements

I would like to thank the people who have been dedicated to helping me finish my degree:

Jonathan Schaeffer, my supervisor

Sheila and Norbert Berkowitz, my mentors

Melvin and Beth Fenske, my parents

Lingyan Shu, my friend

Much time has been spent and many words said; now the text is in your hands.

Table of Contents

1. Introduction

Classifier systems are rule-based artificial intelligence applications which use bit strings to represent knowledge in the form of messages and rules. Learning takes place when new rules are created from existing rules by applying genetic operators. There is no attempt to understand the meaning of the messages or rules: any rule which leads to a successful goal state is considered desirable and is given a high strength. (Rules which lead to a failure state are eliminated.) The user's control over a classifier system is restricted by the constantly changing nature of the rule and message lists. In many ways, trying to work with a classifier is like trying to program a computer in machine code when the instruction set keeps changing.

Classifier systems are simple machines which respond to simple commands: add a message; add a rule; show me all rules; create a new rule by genetic crossover; generate a new message list; etc. Users want more complex operations: execute the following sequence of commands until a certain condition is true; save all information in a file; repeatedly change a previously saved state in different ways and observe the results; etc. Giving the user a direct connection to the classifier would be tedious, because the user would have to type all of the commands himself (in bit string form) and manually inspect all output to look for conditions of interest.

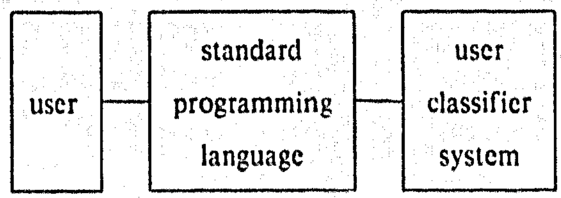
Our research group into classifier systems (Robert Andrew Chai, Keith Fenske, Jonathan Schaeffer, Dale Schuurmans, and Lingyan Shu) quickly developed a need for a classifier "front end" to provide standard programming constructs such as variables, control statements, pre-defined functions, etc. It was not clear how this front end was to be implemented.

The fastest solution would have been to write a library of subroutines that a user could call from programs written in Pascal [Bo81] or "C" [Ke78]. This was rejected for two reasons. First, the user would be working in a language with far more rigid concepts of syntax and data type checking than are necessary for the relatively simple requirements of classifier systems. Second, every time the user made a change, he would have to recompile his program. A more interactive solution was necessary.

The next choice would have been an interpreted language such as LISP [Wi84]. This would have given the user the necessary amount of interaction. He would have been able to try something, check the result, and try something else — all within the same LISP session. LISP's definition of recursive lists has the appropriate amount of structure for representing data in a classifier system. The problem with LISP is that a software designer must choose one of two options: either the application is written in LISP and allows the user to invoke it with LISP expressions, or the application hides the fact that it is written in LISP and maintains complete control over its execution. The first option forces the user to learn LISP, and LISP control "statements" are not well-suited to casual programming. The second option forces the interpreted LISP program to interpret a user's program, resulting in a significant loss in speed. Rejecting LISP was not easy, because having an interpreter that (at any time) can evaluate dynamically-created expressions is worth some loss in performance.

A decision was made to design a new programming language. This language is called `face`, is interactive, can represent the data in an arbitrary classifier system, and is close enough to existing languages (Pascal and "C") that only minor training is necessary. The language has been kept simple from the user's point of view. Data follows the representations in LISP: there are numbers, strings, and lists. No data type checking is performed, as this appeared to be unnecessary. Control flow is procedural with expressions, function definitions, and control statements ("for", "if", "repeat", "while"). The resulting language looks like the pseudo-code often used to describe the execution of algorithms.

The first major design question was the relationship between the programming language and the classifier system. Should they be two parts of the same program? Or should they be two different programs? Combining them together into a single executable module gives the programming language better access to data in the classifier system, but restricts how the classifier is written and works (see Chapter 11). Separating them limits communication between the two programs, but allows the classifier to be modified or replaced without changing the programming language. The approach used here places the programming language between the user and his classifier system:



The connection between the user and the programming language is the familiar terminal with a keyboard. Of course, an operating system like UNIX® [Ke84] allows much more sophisticated connections, but the standard input is assumed to be an interactive terminal. (Throughout this document, the word "UNIX" refers exclusively to the trademarked software product of AT&T Bell Laboratories.) The connection between the programming language and the classifier system is a UNIX pipe.

The second major design question was the implementation language. "C" was chosen to make the best use of the UNIX compiler-development tools LEX [Le78] and YACC [Jo78].

During implementation, the person writing the programming language (Keith Fenske) and the person writing the classifier system (Robert Chai) worked at different speeds, depending upon their other duties. Not having an exact specification of the classifier system was a solid benefit for the programming language. Whenever there was some doubt about how a feature would be implemented (or used), a generalizing assumption was made. The result is a programming language which is fully functional *by itself* and may be used or debugged without reference to a classifier. Attached to the language are application-dependent functions which communicate with the classifier. The syntax of the functions, the types of their parameters, and the results they return are all regular objects in the programming language. Changing the classifier system involves no changes to the language; only modifications to the communication support routines are required.

An overview of classifier systems is next. Then the representation of data is discussed. Given data structures and the skeleton of a programming language, lexical and syntactical grammars are developed. Associated with the grammars are semantic rules for creating parse trees, which are executed by an interpreter (with traps for error conditions).

Pre-defined functions are chosen to provide the user with a complete programming environment. A communication interface to the classifier system is attached. Finally, comments on the success or failure of certain features are presented.

2. Classifier Systems

A classifier system is an artificial intelligence program based on bit strings and genetic operators. There have been numerous papers published in this area, and more than a few conferences held, so an introduction is best served by presenting some background material and then deferring to one of John H. Holland's papers [Ho86] which describes applications in function optimization and robotics. Also mentioned in the Holland paper are a classifier to play the card game of "poker" (S. Smith, 1980) and a classifier to detect leaks in a pipeline (D. Goldberg, 1983). See [Ho86-2] for related work and [Ri86] for an implementation.

2.1. Bit Strings

A bit string is a string consisting of the characters "0", "1", and "#" (don't care). The following are examples of legal bit strings:

"0"

"1101#"

"0#1#0"

Each bit string represents all possible binary numbers with a given pattern. The binary numbers must have as many digits as there are characters in the bit string. Where the string has a "0", the same position in the numbers must have the binary digit 0; where the string has a "1", the numbers must have the bit 1; and where the string has a "#", then any bit is acceptable. In the previous example, the first string represents only one number (zero), the second string represents two numbers, and the third string represents four numbers:

bit string	binary numbers	decimal equivalent
"0"	0	0 (zero)
"01#"	010	2
	011	3
"0#1#0"	00100	4
	00110	6
	01100	12
	01110	14

Bit strings are easy for computers to work with because their alphabet is limited to three characters, and they can be stored as logical masks and values. (In a "mask", significant digits are ones and ignored digits are zeros.) The mask for "01#" would be 110 and the value would be 010. An arbitrary binary number matches the bit string "01#" if the logical AND of the number and the mask 110 is equal to the value 010. Logical operations are the fastest instructions on a computer.

2.2. Messages and Rules

Information about a classifier system's current state is stored in messages, which are fixed-length bit strings. Generally, only the characters "0" and "1" are allowed. All messages have the same length, based on the assumption that if everyone works with objects of the same size, then programming is easier. Collectively, the messages are referred to as the "message list".

Rules act upon the message list. Each rule has a condition bit string and an action bit string, and is written as:

condition / action

If there is a message in the message list with the same pattern as the condition, then the action is invoked. For example, if the message list is:

"00111"

"01010"

then the rule:

"01010" / "11111"

matches the second message, and the action "11111" is invoked. Invoking an action posts a new message to the message list; in this case, the message "11111".

Conditions may have "don't cares" ("#") for some of their pattern bits. Actions may also have "don't cares", in which case the corresponding bits from the matching message are passed through unchanged. The previous rule could be modified to match *both* of the original messages:

"0##1#" / "1####"

This new rule changes the first bit in a message to a one, and passes on the other bits unchanged. If it is applied to the first message ("00111"), then the result is "10111"; applied to the second message ("01010"), the result is "11010".

Conditions may be negated by placing a minus sign ("-") before the condition. A negated condition is true only if there are no messages which match the given pattern:

-"000##" / "00000"

says that if there are no messages with zeros in the first three bits, then create a new message that is all zero.

More than one condition may be specified in the same rule. All conditions must be satisfied before the rule's action is invoked:

"0####" , -"####1" / "0###1"

looks for a message that has a zero in the first bit, checks that no message has a one in the last bit, and then creates a new message beginning with a zero and ending with a one.

Holland is not very clear about the exact meaning of multiple conditions. Let c_1, c_2, up to c_r be the conditions. Let a be the action. Then Holland says on page 603:

> The condition part of the classifier C is satisfied if each condition c_i is satisfied by some message M_j on the current message list. When the classifier is satisfied, an outgoing message M^* is generated as before using the message M_j satisfying condition c_1 and the action part a.

Must the *same* message M_j satisfy all conditions? Or can there be r possibly different messages satisfying the separate conditions? If the same message must satisfy all conditions, then it is unnecessary to identify M_j as the message satisfying condition c_1. If different messages can be used to satisfy the individual conditions, then Holland's example on page 605 makes more sense. Suppose that a message M_5 is to be generated when the following "Boolean" condition is true:

```
(C1 and C2) or [C3 and (not C4)]
```

That is, when conditions c_1 and c_2 are jointly satisfied, or when condition c_3 is satisfied and c_4 can not be satisfied. This may be rewritten as two classifier rules:

```
C1 , C2 / M5
C3 , -C4 / M5
```

If c_1 through c_4 were single-valued Boolean variables (true or false), then this wouldn't work — because classifiers have no concept of variables, Boolean or otherwise. Fortunately, conditions are bit string patterns representing sets of messages, and are "satisfied" when the sets are not empty.

Thus, the answer to the earlier question is, yes, the messages may be different. A multiple-condition rule is satisfied if there are messages which independently satisfy each of the conditions. (Multiple conditions could be replaced by single conditions and flag bits in the messages, if it weren't for negative conditions.)

2.3. Genetic Operators

A classifier system is given an initial set of messages and rules by the "environment" (that is, the user). One execution cycle consists of the following steps:

— receive messages from the environment (if any),
— find all rules whose conditions are satisfied,
— generate a new set of message strings,
— send messages to the environment (if any).

While indefinite cycles are possible with this scheme, it fails to do any useful work except in the extreme case where the rules are already perfectly developed. For the classifier to learn, it must adapt its existing rules to new situations. The adaptation scheme has two parts: assigning a "strength" to each rule, and specifying a method of creating and testing new rules.

The strength of a rule is a measure of how often the rule has lead to a successful goal state. Goal states are characterized by a large number called "pay-off". (Failure states are usually characterized by a large negative number.) The more frequent the success, the higher the strength. Strong rules are given preference over weaker rules in the hope that they will again be successful. Strength is a dynamic quantity which is constantly updated according to the current situation. Please refer to Holland's paper for a description of the "bucket-brigade" algorithm.

New rules are created by "genetic" operators. The "crossover" operator takes two strong rules, swaps some of the bits at random, and creates two new rules. The "invert" operator takes one rule, inverts some of the bits, and creates a new rule. The "mutate" operator takes one rule and replaces some of the bits. If these new rules are functional (lead toward a goal state), then they acquire strength and may displace their parent rules.

If they are non-functional, or are special cases of rules that already perform well, then they may be eliminated by other new rules.

Genetic operators imitate the apparent changes to DNA molecules that occur during sexual reproduction. Consider the following two rules:

"000" / "##0"

"001" / "##0"

These rules have the same action and their conditions differ in only one position. If the condition in either rule is mutated to have a don't care in the third position, then a more general rule is created:

"00#" / "##0"

This new rule is preferred over the first two rules because it can be applied in both situations where the first two rules apply. As the new rule gets used, it acquires strength. The first two rules are not used, and their strength is reduced. Eventually, they become sufficiently weak that they are eliminated. (This assumes, of course, that these rules are functional!)

The genetic operators depend upon dynamic strengths to judge the fitness of each rule.

2.4. User Environment

The classifier system functions as a machine. A user feeds it some initial messages and rules. Commands are given to apply genetic operators or to generate the next list of messages. These messages are inspected. If a goal state is reached, then the classifier system is rewarded with a pay-off. If a failure state is reached (as happens when the classifier is playing a game and makes an illegal move), then a negative pay-off occurs. Otherwise, the process is repeated.

No matter whether the classifier system "wins" or "loses", the rules and adjusted strengths are an improved description of the application problem. This description may not agree with a human description, but it still makes the classifier better prepared to solve the same problem again. The new information should be used in further trials, or saved and reloaded at a later time.

2.5. Classifier Example

To demonstrate how a classifier system can be used, here is a simple example which teaches the classifier to turn on a light if two switches are in same position, and to turn off the light otherwise. This example is easy because only one message is exchanged between the programming language and the classifier system at each step.

Messages need three bits: two for the switches and one for the light. Let "0" mean that a switch (or the light) is off and let "1" mean on. Then the message string:

```
"010"
```

says that the first switch is off (left bit), the second switch is on (middle bit), and the light is off (right bit).

The classifier is expected to find a set of legal rules for turning the light on and off. At least one initial rule must be given to the classifier so that it can create other more meaningful rules. Using the syntax of the pre-defined functions described in a later chapter, we can supply a dummy rule:

```
rule({"00#","##0"});
```

telling the classifier to turn the light off if both switches are off. This is *not* one of the legal rules that we want the classifier to find!

The classifier is trained by picking random test cases until it makes at least 99 (for example) correct answers between failures:

```
correct := 0;
while (correct < 99)
do
    # generate and test
end;
```

The first part of the "generate and test" procedure is to apply genetic operators to the classifier rule list:

```
crossover();
invert();
mutate();
```

These operators create new rules which may or may not be legal. The rules are tested by choosing switch and light settings at random and sending these settings to the classifier as a message. Let *first* be the first switch, *second* be the second switch, and *light* be the light:

```
first := random(2);
second := random(2);
light := random(2);
```

assigns all three variables to be non-negative random integers less than 2 (that is, 0 or 1). A message can be created by converting the numbers (above) into the characters "0" or "1", and then concatenating the characters together into a string:

```
new := "01"[first+1]
    + "01"[second+1]
    + "01"[light+1];
```

This string *new* is sent to the classifier with:

```
message(new);
```

Suppose that the switches are on and the light is off. Then the message:

```
message("110");
```

will be sent to the classifier. The classifier is told to apply its rules to the current message list (consisting of one message in this example):

```
generate();
```

generating one new message as the first element in the global variable *messlist*. If the resulting message has the correct setting for the light (and does not change the switches):

```
if (messlist[1][3] = "01"[(first=second)+1])
   and (messlist[1][1:2] = new[1:2])
```

then the classifier is rewarded and the number of correct answers is incremented:

```
then
    payoff(999);
    correct := correct + 1;
```

where "999" is just a large number with no special meaning. If the light has the wrong setting, then the classifier is punished:

```
else
     payoff(-999);
     correct := 0;
end;
```

This random testing is repeated until the classifier gives consistently good results, at which point it should have exactly four strong rules in the global variable *rulelist*:

```
"00#" / "##1"
"01#" / "##0"
"10#" / "##0"
"11#" / "##1"
```

(returning to Holland's syntax for specifying rules). Depending upon how the classifier is implemented, it may prefer to replace the don't cares ("#") in the actions with the literal bits from the conditions.

2.6. Observations

Much of the description above assumes restrictions which are only necessary for keeping the classifier system simple:

(1) Replacing fixed-length strings with variable-length strings would allow the classifier to create longer messages and rules to encode additional information observed about the problem domain. The cost would be in sign-extending older messages or rules.

(2) Instead of having one or more conditions per rule, we could allow *zero* or more conditions. A rule with zero conditions would be equivalent to a message, since the null condition can be defined as true. A general implementation could then remove the distinction between messages and rules!

(3) Without strong initial rules, the classifier system randomly creates and eliminates a large number of rules before it reaches a state with some pay-off (however small). Then it randomly searches near the first pay-off. If the state space is considered to be a flat surface, then the system acts like a drunk sailor wandering around a lamp post. (This is known as a "random walk" in probability theory. See [Fe68] for a description, and for the more general case of Markov processes.) Only when the classifier develops rules leading to a goal state does this behavior moderate. Hence, one suggestion for improving the performance of a classifier system is to initially train it in situations very close to a goal state.

(4) Bit strings are deceptive. Because they have only two values (zero or one), they make the problem of coding an application look simple. This simplicity is not real, and can result in different performance depending upon the coding scheme. For example, if a field in a classifier message has three possible values (say: yes, no,

and maybe), then two bits must be allocated. One scheme is:

00 = yes

01 = no

10 = maybe

If this field is randomly changed ("mutated"), then it should have an equal chance of going from one value to another. The classifier system does not respect this desire. By mutating a bit at random, it can turn "yes" into "no" or "maybe", "no" into "yes", and "maybe" into "yes" — but it can never turn "no" into "maybe" or vice versa. (Ignoring, of course, the possibility of the illegal combination 11.) Thus, the coding of message fields affects the performance of the classifier system.

3. Representation of Data

The representation of data is a major decision in any programming language. The amount of work required to implement basic operators such as addition ("+") is proportional the number of different data types which may appear as operands.

3.1. Numbers

Numbers are necessary for representing the "strength" of rules in the classifier system. Either real numbers or integers are acceptable. (Even if the classifier wants real numbers, integers can be scaled by the communication routines.) Numbers are also necessary for general programming in the classifier language. It would be hard to loop through a set of statements, or to count objects, without numbers.

Stealing a trick from APL [Gi76], the user has no control over the internal representation of numbers: all numbers are double-precision floating-point. The effective range of double-precision real numbers on most computers (17 decimal digits) is larger than the range of long integers (10 digits), so the user will never notice that integer calculations are being done in floating-point. Any additional overhead caused by the floating-point arithmetic is buried in the other actions of the compiler.

3.2. Strings

The definition of strings is harder to decide. Should strings be arrays of characters (as is done in APL, "C", and Pascal)? Should strings be a basic data type (completely replacing the concept of an individual character)? Are bit strings different than ordinary strings? These questions (and more) arose before a rather "obvious" choice was made.

Strings need to be built up from characters when it is important to manipulate individual characters. This would happen, for example, if a string was a card image where each column had a separate meaning. In the classifier system, messages are assumed to be composed of fields. Is it reasonable to assume that fields will be one character long? That is, should we assume that the user will want to manipulate messages by changing individual bits? (Remember, one character in a message is one "bit" in a bit string.) Implied is a further assumption that all fields can be encoded as one bit — which has already been contradicted by a previous "yes", "no", or "maybe" example. Hence, it is unlikely that the user will work only with individual characters. Manipulating fields in a bit string will involve groups of characters. Groups of characters are otherwise known as strings!

Now, if strings are a basic data type, is there a difference between "regular" strings used for text (as in output to the user) and "bit" strings used for binary patterns? We could have separate definitions for text strings versus bit strings by putting text strings in double quotes (") and bit strings in single quotes ('):

```
"this is a text string"
'00100#111'
```

Would this buy us anything? Are the operators applied to bit strings completely different than the operators applied to normal strings? For example, will we want to subscript ("index") bit strings but never regular strings? Will we only want to write out regular strings as text to the user, but never bit strings? Is there any situation where bit strings and regular strings will need to be combined? Better yet, is there any situation where we will have a string and not know what kind of string it is?

Too many questions like these were asked, and too much code was duplicated, before the observation at the beginning of this chapter was formalized: doubling the number of data types doubles the amount of work during implementation. Bit strings and text strings are both implemented as strings. For most operators, bit strings are treated no differently than any other string. Only when an operator requires interpretation are bit strings treated in special ways. (Example bit string operators are *and*, *or*, and *not*. Please refer to Appendix A for a complete description of the string operators.)

3.3. Lists

Numbers and strings need to be collected together for at least one obvious reason and one not-so-obvious reason. The obvious reason is the representation of rules: a classifier rule has a condition part, an action part, and a strength number. We could force the condition part to consist of exactly one bit string. Then to create a new rule, we would need to send a condition string, an action string, and a strength. To print a rule, we could print the condition string followed by the action string followed by the strength. However, if we return the rule so that the user can manipulate it, then we must have a data object which is capable of holding two strings and a number.

What kind of object can hold two strings and a number? An array that allows elements to be of different types (which is not legal in Pascal or "C"). Or a record with one string field for the condition, another string field for the action, and a numeric field for the strength.

A less obvious problem is the representation of fields within a message. Ideally, we would like to name the fields in the Pascal style of records. If the data in our language was strongly typed (as in Pascal), then we would know what values were legal in every part of each piece of data. Creating a message would then be a matter of declaring a variable of the appropriate type and assigning values to each field. This can be done (and is done) in many compiled languages where the user types his program into a file, compiles it with a compiler to produce object code, and then runs the object code. This suffers from a lack of interaction. Why should a user type:

```
m : message;
m.first := yes;
m.middle := no;
m.end := maybe;
```

just to create a message which will be converted into the bit string:

```
"000110"
```

Given the choice between typing six bits in quotes and four lines of Pascal, the user will slowly type the bits (and swear about how obscure they are). Of course, there is no need to use pure Pascal syntax. We could introduce new delimiters to create an object with an assumed data type. For example:

```
< yes , no , maybe >
```

might be a shorthand way of creating something of type "message". This works well if there are only a few data types with special syntax, such as classifier messages and rules. Unfortunately, big messages and rules are composed from smaller less-complicated pieces which still may be big enough to need their own special syntax. Changing the language to allow for any number of special cases gets ridiculous.

The syntax used in the previous "<>" example looks suspiciously like LISP lists. The only difference is that we are assuming a fixed interpretation of the data types. Going back to the more obvious need for collections of data, a point was ignored which strongly implies a list structure. The condition part of a rule consists of one *or more* condition bit strings. Representing exactly one condition was shown to be easy. Having two conditions is equally easy, with both records and arrays, because conditions have the same "type". Exactly three conditions causes no new problems. In fact, any exact number of conditions can always be represented with either records or arrays. The same is true if a maximum number of conditions can be assumed (by replacing unused conditions with some "null" value). The phrase "one or more" does not specify a limit. If we assume a limit, then the programming language will be unable to support general classifier systems. Hence, we must not assume a maximum number of conditions. Arrays that are declared with fixed sizes can not be used here. Records that are declared with a fixed number of fields (even "variant" records in Pascal) can not be used. Some structure that has a variable number of entries must be used.

LISP collects data together in lists. A list is either empty, or contains elements. If it is empty, it looks like this:

( )

If the list is not empty, then it contains one or more elements separated by spaces. Each element is either an atom or a list. The following are examples of legal lists (with simple values):

```
(3)
(dog)
(3 dogs (sat in) a lake)
```

Even programs in LISP are lists!

Data in classifier systems is represented by lists in the programming language. No data type checking is performed on the list elements, because classifier data is sufficiently simple that type checking is unnecessary. Without strong typing, there is no need for type declarations. Without type declarations, variables are whatever they are assigned to be, and do not need to be declared in advance. Type and variable declarations are a major part of traditional language grammars. Thus, a non-traditional grammar will be required. The only concession is purely syntactical: "(" and ")" are commonly used as parentheses in algebraic expressions, so different characters should be used for delimiting lists. "{" and "}" have been chosen because they are familiar as set notation.

Please refer to Appendix A for a complete description of the list operators, and for data objects in general.

3.4. Internal Representation

The classifier programming language has numbers, strings, and lists. Numbers are always double-precision floating-point. Strings consist of zero or more characters. Lists have zero or more elements, each of which may be a number, a string, or another list.

To represent a number, we must tell "C" to declare space for the number:

```
NUMBER number;
```

where *NUMBER* is *#define*'d by the "fainc.h" module to be the "C" *double* type. To represent a string, we must declare space for an array of characters:

```
char string[MAXSTRING];
```

says that *string* is an array of characters indexed from zero to *MAXSTRING*-1. Of course, this assumes that all strings have the same length (*MAXSTRING*), and that no space is wasted by padding trivial strings to the maximum length. A more reasonable method is to dynamically allocate strings by some as yet unspecified means, and to save the address of the string:

```
char * string;
```

To allow strings to be signed (as in classifier conditions), we must save the sign of each string:

```
int sign;
```

where *sign* is either +1 for a positive string or −1 for a negated string.

Numeric and string values are stored in a "C" structure known as *ValueThing*:

```
#define ValNUMBER 703
#define ValSTRING 705

typedef struct ValueThing {
    NUMBER number;
    int sign;
    char * string;
    int type;
} ValueThing;
```

where *type* identifies the data type in the structure according to the symbols *ValNUMBER* and *ValSTRING*. These *#define*'s are prefixed with "Val" to avoid confusion with similar lexical tokens which are defined later. Although the *number* field is used only for numbers, and *sign* only for strings (and lists), these two fields are not combined — even at the expense of extra code — to keep the code readable and to provide a degree of internal error checking. (See the *CheckSign* routine.)

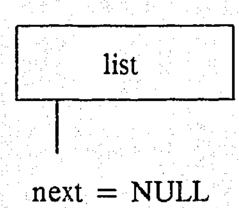
Lists are represented as linked lists. A list is a value structure with a *type* field of:

```
#define ValSET 704
```

Lists have signs (defined as for strings). Lists also point to the first element in the list:

```
struct ValueThing * next;
```

next may be the address of a *ValueThing*, which is the first element, or it may be the NULL pointer. If *next* is NULL, then the list is the empty list, because it has no elements:



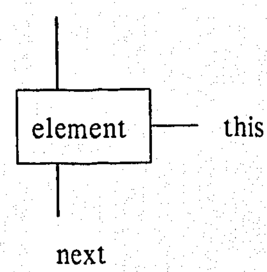
If *next* is not NULL, then it points to a value structure of type:

```
#define ValELEMENT 702
```

An element has no sign, is not visible to the user, and only serves to connect value structures containing real data. Elements have *next* pointers along with *this* pointers:

```
struct ValueThing * this;
```

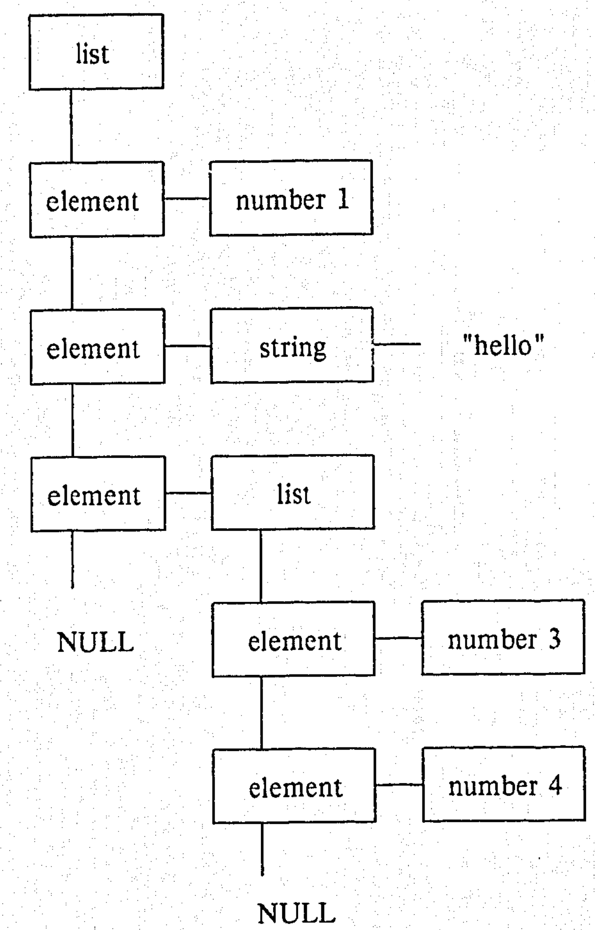
this is the address of a value structure for a number, string, or another list:



When lists and elements are linked together, data objects which look small to the user are expanded into much larger (but very regular) structures. For example, the list:

```
{1, "hello", {3, 4}}
```

has three elements: a number, a string, and a list (whose elements are two numbers). Internally, this is stored as:



This example is a visual answer to questions such as "why does a list point to its first element through *next* instead of *this*"? *this* always points to a complete data object, that is, an object which would be legal even without the presence of the list. Lists are created recursively; there is no difference between the structure of a list which appears by itself and a list which is an element in another list. (The next element in a list is always accessed through *next* independent of whether the current value structure is an element or a list.)

Lists in this language are not as compact as the lists in LISP. The basic change is the addition of a *ValSET* structure in front of what would be the LISP list. This places information about the data type in the data itself, without having to reference some external dictionary. Further, the LISP definitions allow sublists to be rooted at each element (see the *cdr* function); here elements are not legal data objects by themselves, and hence are not legal as sublists. The choice is more than just a question of representation.

LISP does not copy values when creating a list. If the same value is used more than once, then changing one reference to the value will change all other references. This is a very convenient property for experienced users, especially when combined with a copy-on-demand function, but violates an informal principle known as the "law of least astonishment" (attributed to the University of Michigan programmers who wrote the MTS operating system). This law states that when the user types a command, the system should do the simplest and most obvious action which is consistent with the phrasing of the command. The classifier language will be used in an interactive environment. The user will be typing commands, observing the results, and then typing more commands. If he assigns a value to a variable early in his session, then this variable should retain the same value throughout the session, until he explicitly changes it. LISP may turn a variable into a reference to some other variable, print the correct value now, but later indirectly change the value of the referenced variable. The frustration of the user can be extreme, especially when the original reference has long since vanished from the screen and the user's memory.

To keep users happy, and to reduce the author's work, unique copies of all elements are made when a list is created.

Why are elements forced to point to complete data objects? Elements could themselves contain the value, along with a link to the next element. The reason has to do with assignment. Lists replace arrays and records. The user will want to change values within a list. If elements contained the values, then the elements would have to be overwritten to assign a new value (to avoid damaging forward and backward pointers). Having a *this* pointer to the element's value allows the assignment to be done by replacing one pointer — much faster.

Here is the full definition of a value structure, as taken from the "fainc.h" module. Comments have been removed, due to the limited width of this formatted page:

```
#define ValDUMMY 701
#define ValELEMENT 702
#define ValNUMBER 703
#define ValSET 704
#define ValSTRING 705

typedef struct ValueThing {
    struct ValueThing * next;
    NUMBER number;
    int sign;
    char * string;
    struct ValueThing * this;
    int type;
} ValueThing;
```

(Dummy values are explained later — much later.)

One final note: The "C" programming language allows pointers to be NULL. Internally, the classifier language must check *next* pointers to see if they are NULL. Checking *this* pointers is not much more difficult. If we allow NULL values for both of these pointers, a list may have elements which point to NULL. We could hide this fact from the user, or we can make it visible. By creating a special symbol (called "NULL"), the user may have NULL values in his lists. Lists are recursively defined. If lists can have NULL elements, then NULL values must be legal by themselves. NULL values hence become a fourth type of data object. This may seem a minor point now, but having explicit NULL values makes other parts of the language much easier. (For example, what is the value of a global or local variable which has not been assigned a value? Answer: NULL!)

4. Lexical Analysis

Input from the user is structured as a small programming language. Programming languages are parsed (understood) by recognizing pieces called "tokens", combining tokens into statements according to syntax rules, and then executing semantic actions associated with the rules. The work of recognizing tokens in the input language is done by the lexical routines.

4.1. Overview of Lexical Analysis

An example will help to explain what a token is. Consider the following expression:

```
(old + new) > 25
```

This expression adds the value of the variable *old* to the value of the variable *new* and compares the sum against the number 25. There are seven tokens in this expression (ignoring the spaces):

token number	token string
1	`(`
2	`old`
3	`+`
4	`new`
5	`)`
6	`>`
7	`25`

Informally, tokens are input words separated by punctuation.

Different languages place different demands upon the lexical routines. In some languages, the recognition of tokens depends upon the current context of the parser. In an extreme case, the same input may be interpreted in different ways depending upon where it appears in a program:

```
elseif
```

might be recognized as the token "else" followed by the token "if" in a conditional statement, but as the variable name "elseif" in an assignment statement. Rules like this are hard to implement, because the same sequence of input characters can generate a variable number of tokens.

Less extreme is the case where input is always broken into the same tokens, but the meaning of the tokens may change. A confusing but legal example comes from the PL/I language (page 87 of [Ah86]):

```
IF THEN THEN THEN = ELSE; ELSE ELSE = THEN;
```

To quote [Ah86]: "In PL/I, keywords are not reserved; thus, the rules for distinguishing keywords from identifiers are quite complicated". The lexical routines must have full knowledge of the parser's context before they can break input characters into tokens. Sharing this much information makes a compiler more difficult to write, because the lexical analysis can not be cleanly separated from syntax and semantic analysis.

To avoid these problems, languages such as Pascal reserve some of the possible input words, and force reserved words to have a fixed meaning. Examples are words like "if", "begin", and "end" in Pascal. "if" is always the first part of a conditional statement; "begin" always introduces a compound statement which must terminate with "end". None of these reserved words may be used as variable names. Using "if" where a variable name is required will generate a syntax error.

Removing context sensitivity from the lexical level allows the lexical routines to run almost independently of the syntax or semantic routines. The lexical routines are given complete control over the input stream. When called, they look for the next token, and return this token to the caller. Between calls, they do nothing. In fact, the lexical routines act as a subroutine for the syntax routines. While the syntax routines are putting together an expression, they call the lexical routines whenever the syntax rules say that there may be another token in the expression. When the syntax routines have enough tokens, they reduce rules, execute semantics, etc. The syntax routines never try to read the input themselves.

With the UNIX operating system, lexical analysis can be done by writing your own program, or by using LEX to generate a lexical function called *yylex* from a table of rules written as regular expressions. LEX encourages you to create simple tokens via reserved words and operators, but allows action statements written in "C" which can potentially add as much context sensitivity as required.

4.2. Classifier Lexical Tokens

You have already seen two of the input tokens in the classifier language: numbers and strings. A number is a sequence of digits, possibly containing a decimal point, and optionally followed by an exponent. A much-simplified version of the LEX rule for a number is:

```
[0-9]+
```

which says "a character from 0 to 9 repeated one or more times". The sign of a number ("+" or "-") is not part of the token, or else expressions like:

```
5-3
```

would be treated as two number tokens, with the minus sign as part of the second number, when it should remain as a subtraction operator. Associated with the number rule is an action:

```
{ yylval.number = atof(yytext);
return(TokNUMBER); }
```

Unless you have used LEX before, or are reading the manual now, this explanation will not be very helpful. First, *yylval* is a special place where information is returned to the syntax routines (YACC). Second, *number* is a field within the structure or union for the YACC stack type. Third, *atof* is a "C" function to convert an ASCII character string to a double-precision floating-point number. Fourth, *yytext* is where LEX stores the current input token. Fifth, *TokNUMBER* is a flag returned to the syntax routines to indicate that a number was found. (Remember *ValNUMBER* and the comment about the "Val" prefix? Well, here is the corresponding "Tok" prefix.)

Interested readers may refer to the listing of the "falex.l" module in Appendix B. The remainder of this discussion will avoid the technical details.

Strings are recognized when LEX finds a delimiter which may begin a string: double quote ("), closing quote or acute accent ('), and opening quote or grave accent (`). LEX is not asked to find the entire string, because it is unable to properly handle "escape" sequences within a string. A function by the name of *LexString* is called instead. *LexString* builds up the string in a static buffer using input bytes supplied by LEX. The string ends when a matching delimiter is found. Between the starting and ending delimiters, there may be regular text characters and escape sequences. An escape sequence puts a special character into the string. Backslash followed by "n" (that is, "\n" without the quotes) is replaced by a single newline character; "\t" is replaced by a tab character; "\\" is replaced by a single backslash character; and backslash followed by the string delimiter puts one delimiter into the string as text. Before returning the string to the syntax routines, *LexString* allocates enough dynamic memory to hold the string (plus a null byte as a terminator) and then copies its static buffer to the dynamic memory. The caller is given a string token (*TokSTRING*) and the address of the dynamic copy.

The next most interesting tokens are comments. Comments are thrown away by the lexical routines before they can reach the syntax routines. This makes comments invisible to the syntax level. Hence, the only restriction on comments is that they must not conflict with anything used in the syntax grammar. Pascal uses "{" and "}" to delimit comments; the classifier language has already chosen these two symbols to delimit lists. We could use two other paired symbols: anything between these symbols would be a comment. LEX allows tokens to specify input that extends over multiple lines. Every part of each input line matching the token is placed into LEX's token buffer (*yytext*). *yytext* is, by default, 200 bytes long. Putting arbitrarily-sized comments into such a small buffer is almost guaranteed to crash LEX.

Alternatively, we could use the same trick as for strings and call a function to "process" the comments. The function definition would be trivial: read until a delimiter marking the end of the comment, or an end-of-file, whichever comes first. Possible delimiters are "/*" and "*/" familiar to "C" and PL/I programmers. This would work, but there is an even easier way used by the UNIX shells. Pick some character not found in the language, such as "#". (Even though # is used in bit strings, there is no conflict because strings always exist between string delimiters.) Tell the user that everything from # to the end of the line is ignored. Then define the following LEX rule:

```
"#".*
```

with no action. This rule matches # followed by any number of characters except for the newline. (And, as an added joke, comments are introduced by a character which means "don't care" in bit strings!)

In order of their appearance in "falex.l", the following tokens are also recognized: Reserved words ("and", "do", "else", etc.) are accepted in any mixture of upper- or lower-case, supposedly as a convenience to the user, but actually to prevent the user from writing a program which misuses a reserved word by slightly changing its appearance. Relational operators all return *TokRELOP* for their token number, with further information stored in *yylval*, since this removes seven operators from the syntax grammar, effectively reducing in half the size of the YACC finite state machine. Variable names are any letter followed by zero or more letters or digits; upper- and lower-case are important in variable names. Special symbols with unambiguous meanings are accepted. For example, all of the "C" relational operators are supported:

== != < <= >= >

as well as the Pascal relational operators:

= <> < <= >= >

The assignment operator is from Pascal:

:=

"%" is accepted for the reserved word "mod"; "&" and "&&" are accepted for "and"; "|" and "||" for "or"; etc. White space (blanks, tabs, and newlines) are ignored. Finally, all single-character tokens which do not require conversion (such as "*") are returned directly by a default rule.

4.3. Test for Lexical Routines

The lexical routines are called from deep within the YACC-generated parser. If the information they return is bad, then YACC will produce unexpected "syntax error" messages. YACC does not tell you why there is an error, or what the offending token is, only that the input does not agree with its rules. (You can use the *YYDEBUG* flag, if you dare.) Fighting with YACC is a needless trouble, especially when the lexical routines are perfectly happy to run without YACC.

To test the lexical routines, a program called "telex" was written. *telex* allocates space for the global variable *yylval*, where LEX expects to return information about the tokens, calls *yylex* for a token, prints the token's name and value, and then calls for another token. This procedure continues until you type an end-of-file on standard input.

telex is a quick-and-dirty test routine which took less than an hour to complete, found a number of trivial mistakes in the LEX rules, and saved numerous hours of aggravation when LEX was finally attached to the syntax routines.

5. Syntax and Semantic Analysis

The classifier programming language will be used in an interactive environment (that is, will talk to a user sitting at a terminal). Interaction is more than just typing a program into a file, running it through a compiler, and observing output on the terminal. Interaction means trying many things — some right, some wrong — all during the same programming session. After each attempt, something is learned about the problem, and this learning guides further attempts. The final product may well be a program saved in a file, but such an inflexible mode should not be forced upon the user.

As tokens are read by the lexical routines, they are put together according to syntax rules. The syntax rules are defined by a compiler-compiler language known as YACC. YACC accepts LALR(1) grammars (see [Ah86]). For our purposes, an LALR(1) grammar parses input starting with the leftmost token, reduces rules according to the rightmost derivation, and allows one character of look-ahead when deciding between two rules with the same partial derivation. Pascal is an example of an LALR(1) language.

5.1. Desirable Features

As statements are read from the user's input, how much should be compiled at each step, and what is done with the compiled result?

Recall from Chapter 3 that data in a list structure is not strongly typed. Individual elements may be numbers, strings, lists, or the NULL value. The type of the data is known only when the list is put together, and may change later if the user assigns a new value to one of the elements. In many languages, before using a variable, you must declare the variable by specifying the variable's name and an existing type. Pascal programs make declarations in the following order:

— constants
— types
— variables
— functions or procedures

Constant declarations must precede type declarations, which must precede variable declarations, etc. You are not allowed to declare a block of related constants and variables, followed by a second block of different related constants and variables. The impression is that the compiler will work only if all constants appear before all types before all variables. Is this necessary? No. Consider the legal declarations in a function or procedure:

— constants
— types
— variables
— functions or procedures

That is, *exactly* the same declarations which are legal in the main program. A variable declaration in the main program may appear before a constant declaration in a function or procedure. While these new declarations are local to the function, they still demonstrate that Pascal must be prepared to handle any declaration in the presence of all other declarations. Why, then, do declarations have a fixed order? To make the language definition more uniform. (The convenience of the user is not important.)

The classifier language does not have strong typing of data. Without type declarations, there is no need for variable declarations: to create a variable, assign it a value. The type of the variable becomes the type of the value. Without type and variable declarations, there is no need for constant declarations (if you want a constant, assign a value to a variable and then don't change it). This leaves only function or procedure declarations.

Declarations are a major part of the Pascal language: by forcing the user to say so much about his variables, the compiler can perform static checking to test the validity of expressions before they are executed. After the declarations come the executable statements. An executable statement calls a function or procedure, assigns a new value to a variable, or is a control statement which can execute any number of other statements. Given that the classifier language has function declarations and executable statements, in what order should they appear? The user is assumed to be sitting at an interactive terminal. Suppose he defines a function, and then tries to call it with an executable statement. The function may work, or it may fail. If the function works, then the user is happy; if it fails, he may redefine the function to fix a mistake. Then another executable statement will be needed to test the new function. Thus, the definition of functions and the execution of statements will be intermixed, and no specific order should be assumed.

The classifier language is an interpreter (not a true compiler). One function definition or executable statement is read at each step. Statements are converted into parse trees, and the parse trees are executed. The results of the execution (if any) are shown to the user. Statements can be simple expressions (such as adding two numbers), or complex programs making full use of the conditional and looping facilities (*for*, *if*, *repeat*, *while*). Correctly formed function definitions do not require any action; the parse tree for the function is saved so that it can be executed later via a function call.

The syntax grammar for the classifier language may resemble Pascal and other compiled languages, but the semantics of the language are closer to APL or LISP where anything and everything can be redefined at any time.

5.2. Language Grammar

Before explaining how individual parts of the language grammar were implemented, it is helpful to summarize the grammar in a more formal notation. Backus-Naur forms (BNF) are commonly used (see [Bo81] for Waterloo Pascal); here it is more natural to extract the YACC definitions from the "fayac.y" module in Appendix B. By annotating the YACC code, the curious reader may review the more detailed grammar.

At each call to the parser, one production of "program" is parsed:

```
program
        :
        | function ';'
        | statement ';'
```

says that a "program" can be empty (end-of-file), a function definition followed by a semicolon, or a statement followed by a semicolon. A function definition is:

```
function
        : TokFUNCTION TokNAME func_head func_body TokEND
```

That is, a function definition consists of the token "function" (*TokFUNCTION*) followed by a name (*TokNAME*), a function header, a function body, and the token "end" (*TokEND*). A function header is:

```
func_head
        :
        | '(' ')'
        | '(' func_pars ')'
```

allowing for a function header which is missing (the first rule), has only the left and right parentheses (second rule), or has function parameters between the parentheses. Parameters are a list of names:

```
func_pars
        : TokNAME
        | func_pars ',' TokNAME
```

Lists in YACC are defined by their first instance (first rule) along with their repeated instances (second rule). The function body is:

```
func_body
        :
        | TokDO stmt_list
```

Again, the first rule allows a null body. The second rule is for the normal case when the token "do" precedes a list of statements:

```
stmt_list
        : statement
        | stmt_list ';' statement
```

which is a left-recursive production like the function parameters. Statements can be:

```
statement
        :
        | expr
        | for_stmt
        | if_stmt
        | repeat_stmt
        | while_stmt
```

null (empty), expressions, *for* statements, *if* statements, *repeat* statements, or *while* statements. Expressions are simple terms or operators applied to other expressions:

```
expr
        : expr_term
        | TokNOT expr
        | '+' expr
        | '-' expr
        | expr '*' expr
        | expr '+' expr
        | expr '-' expr
        | expr '/' expr
        | expr TokAND expr
        | expr TokASSIGN expr
        | expr TokDIV expr
        | expr TokMOD expr
        | expr TokOR expr
        | expr TokPOWER expr
        | expr TokRELOP expr
        | '(' expr ')'
        | '{' expr_set '}'
        | expr '[' expr_index ']'
```

Lists are right-recursive productions which may be empty:

```
expr_set
        :
        | expr_slist
expr_slist
        : expr
        | expr ',' expr_slist
```

Index expressions are left-recursive, but allow two different forms (with or without the colon for a subrange):

```
expr_index
        : expr
        | expr ':' expr
        | expr_index ',' expr
        | expr_index ',' expr ':' expr
```

All expressions are eventually reduced to simple terms:

```
expr_term
        : TokNAME
        | TokNAME '(' expr_pars ')'
        | TokNULL
        | TokNUMBER
        | TokSTRING
```

where the second production is a function call with parameters:

```
expr_pars
        :
        | expr_plist
expr_plist
        : expr
        | expr_plist ',' expr
```

Compound statements group zero or more simple statements together. In the case of a *for* statement, the syntax is:

```
for_stmt
        : TokFOR TokNAME for_from for_to for_by for_do TokEND
for_from
        :
        | TokFROM expr
for_to
        :
        | TokTO expr
for_by
        :
        | TokBY expr
for_do
        :
        | TokDO stmt_list
```

making all parts of a *for* loop optional except for the name of the index variable. *if* statements select between two different statement lists depending upon the value of a conditional expression:

```
if_stmt
        : TokIF expr if_then if_else TokEND
if_then
        :
        | TokTHEN stmt_list
if_else
        :
        | TokELSE stmt_list
```

(Both the *then* and the *else* clauses are optional in *if* statements.) Finally, *repeat* and *while* statements combine a conditional expression with a statement list:

```
repeat_stmt
        : TokREPEAT stmt_list TokUNTIL expr
while_stmt
        : TokWHILE expr while_do TokEND
while_do
        :
        | TokDO stmt_list
```

The justification for this grammar follows.

5.3. Simple Executable Statements

A simple statement is an assignment or an expression. An assignment computes the value of an expression and assigns the result to a variable. An expression by itself is evaluated, and the result is printed. (This differs from Pascal where you must explicitly write out the value of an expression before you can see the result.) For example:

```
5 - 3
```

is an expression. When evaluated, the result of this expression is the number 2. We would like the language to immediately print the result of any simple expression:

```
2
```

Should the language assume that an expression ends when it sees a newline character (carriage return)? This would be convenient for a program pretending to be a desk calculator, and may be acceptable in a program with simple input, but is not acceptable to a user who is trying to assign a 100-element list to a variable. (The screens on most terminals are only 80 characters wide.) If newline characters can not be treated as special delimiters, then they should be ignored like other white space (blanks and tabs). Some other character must be used to terminate statements. A semicolon was chosen:

```
5 - 3 ;
```

immediately prints the result:

```
2
```

The similar-looking assignment:

```
a := 5 - 3;
```

prints nothing. Why? When you assign a value to a variable, you can always see the result later by printing the variable. (This makes the assignment operator different than other operators, so that the user does not see the result of every assignment in a function.) Typing:

```
a;
```

evaluates *a* as an expression, printing the result:

```
2
```

Hence, a simple statement ends with a semicolon, and prints its result if the statement is not an assignment.

Appendix A lists all of the operators which are legal in an expression. The meaning of the operators closely follows the Pascal language. The operator priorities are determined by YACC %left and %right declaratior instead of the long unambiguous productions which are typical in Pascal language definitions. One notable change is that assignment is done by an operator, not as a distinct assignment statement. This is unusual because it allows assignments to occur in the middle of an expression — the semantics of which are questionable. The semantics are clear when an assignment is used in the traditional form:

variable := *expression* ;

or the multiple-assignment form:

variable1 := *variable2* := *expression* ;

Inside an expression, the meaning depends upon the order of execution for the various parts of the expression.

YACC does not care about the order of embedded assignments, nor does the classifier language (that is the user's problem). What YACC does care about is ambiguity. For the sake of an example, assume that the only legal production on the left side of an assignment operator (which is a token of type *TokASSIGN*) is a variable, where a variable is defined elsewhere to be either a variable name (*TokNAME*) or a subscripted variable. Assume also that an expression is defined elsewhere by a production called *expression*. Then the following two rules decide if the input is an assignment or another expression:

```
statement     : variable TokASSIGN expression
                    { /* do assignment */ }
              | expression
                    { /* print expression */ }
              ;
```

We need an address on the left side of an assignment so that we know where to assign the value on the right side. For the expression, we need a value. This creates an ambiguity for YACC. Variable names are legal expressions (otherwise, how would you add variable *a* to variable *b*?). When YACC sees a variable name, it must decide whether this name matches *variable* in the first rule, and is reduced as an address, or matches *expression* in the second rule, and is reduced as a value. Being unable to decide, YACC reports a "reduce/reduce" conflict. To avoid the conflict, the production on the left side of an assignment operator must be the same as the production on the left side of all other operators: an expression.

YACC has a look-ahead token which could decide between the two rules if the only legal left side of an assignment was a variable name (*TokNAME*), because for all legal assignments, *TokASSIGN* would be the next token. If the next token was not *TokASSIGN*, then the input could not be an assignment, and must be an expression.

Pascal avoids this issue by not allowing expressions as complete statements. The convenience of an immediate reply to any expression is too much to give up in an interactive environment. Hence, assignments become just another operator, with an expression on the left side and an expression on the right side. The responsibility of maintaining enough address information to perform the assignment is postponed to the interpreter.

5.4. Creating a Parse Tree

Expressions may be executed while they are being compiled, or they may be saved in a compiled form known as a parse tree and executed later.

Combining execution and compilation has the advantage of removing one layer of software, but has two strong disadvantages: First, the expression must be recompiled every time it is used, even if this occurs inside a *for* loop! Second, YACC does not look for a complete expression, and then compile the individual pieces; YACC reduces pieces ("productions") each time the right tail of the input matches one of its syntax rules. This right derivation can be confusing to the user, especially when there are errors or other implicit I/O. For example, suppose that the user wants to call the function *f*, which asks for a number and returns this number as a result, and then add one to the function's result:

```
f() + 1;
```

If compilation and execution are combined, then after the user types:

```
f()
```

YACC recognizes the function call, and invokes the function:

```
enter a number:
```

where this prompt is written by the function. The user types a number:

which is returned as the function's value. Then YACC starts looking for the rest of the expression, and the user must type:

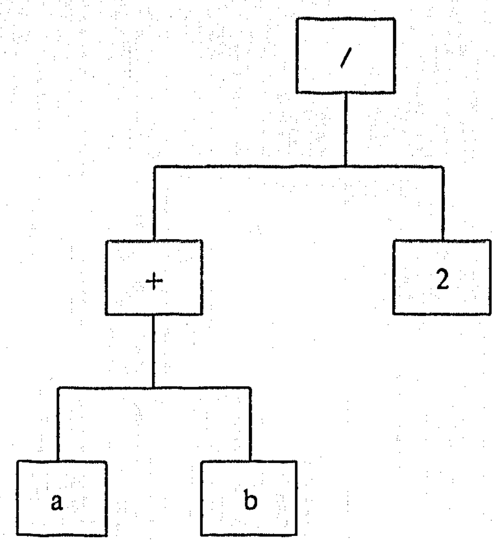
```
+ 1;
```

and the result 6 is printed. This example may seem contrived (it is), but it demonstrates how confusing execution and I/O can be if compilation and execution are combined. Techniques such as buffering a complete input line help alleviate the problem, but still fail to solve truly abnormal situations where the only appropriate response is to print an error message after only part of the input has been parsed. (Buffering the text for an entire statement looks like an easy solution, but would require a very large buffer because a program can be a single compound statement extending for hundreds or thousands of input lines. Unless the entire statement can be parsed before any part is executed -- which would have to be done by the method in the following paragraphs — parsing must be done in pieces, which can cause statements inside a loop to be parsed many times.)

The alternative is to convert the user's input into a compiled version known as a parse tree. A parse tree has a node for each operator. The children of a node are parse trees for computing the values of the operands. For example, the expression:

```
(a + b) / 2;
```

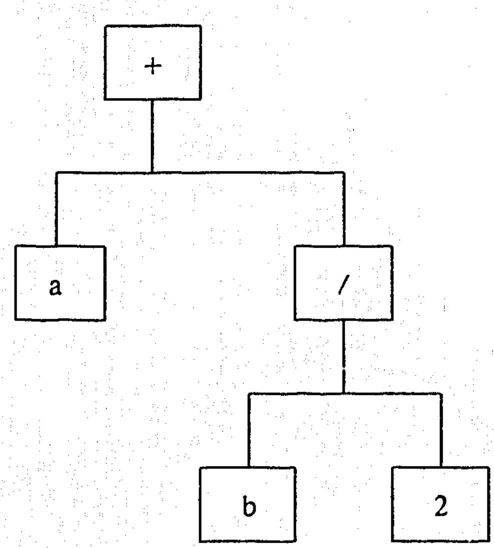
adds the value of *a* to the value of *b*, and divides the total by two. As a parse tree, this is represented as:



Of course, if you forget the parentheses:

```
a + b / 2;
```

you get an entirely different parse tree:

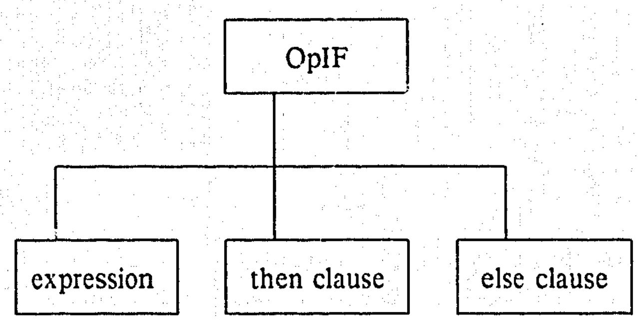


The disadvantages to parse trees are that they take time to create, and that they use extra space. The advantages are that a complete statement is parsed before any part is executed (allowing syntax errors to be handled cleanly), and when the same expression is executed many times, a saved tree can be quickly traversed by having a section of code dedicated to each type of operator node.

The semantic actions in the YACC grammar create parse trees. On each call to the YACC function *yyparse*, exactly one function definition or executable statement is parsed. For functions, the body of the function is a parse tree, which is saved as the function's definition. For executable statements, the address of the parse tree is returned to the caller; it is the caller's responsibility to execute the tree. When a syntax error occurs, a NULL pointer is returned instead.

5.5. Compound Executable Statements

The compound statements in this language are *for*, *if*, *repeat*, and *while*. A compound statement groups zero or more simple statements together. The *if* statement is a parse tree node (of type *OpIF*) with three children: an expression, a "then" clause, and an "else" clause:

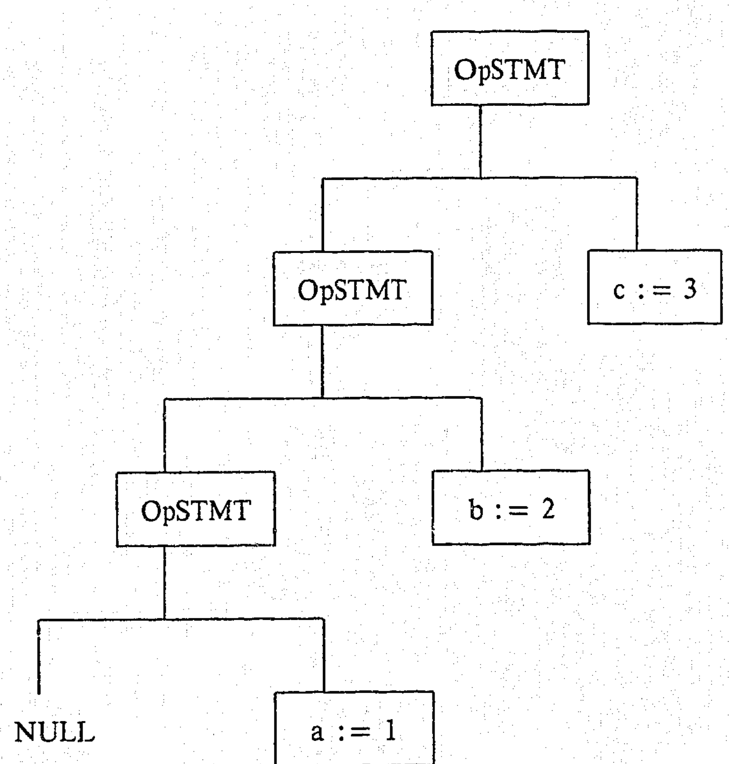


When executed, *expression* must evaluate to either *false* (defined as the number 0) or *true* (the number 1). If *true*, the statements in the *then* clause are executed; otherwise, the statements in the *else* clause are executed. Both clauses are parse trees; their addresses are saved in the *if* operator node. Either clause may be empty, in which case the NULL pointer replaces the parse tree address.

Multiple statements are grouped together as one parse tree by an *OpSTMT* operator node. Given the trivial *if* statement:

```
if true
then
    a := 1;
    b := 2;
    c := 3;
end;
```

the *then* clause is stored as a left-recursive structure:



(The complete parse trees for the assignments are not shown.) This is known as a left-recursive production because the leftmost child of each node must be executed first, or else the statements will not be executed in the same order as they were given by the user. YACC — and all other LR parsers — encourage left recursion, as this limits the maximum depth of their internal stack.

Right recursion is used to create lists. Lists are built up in the same way as compound statements; only the name of the operator node changes. When the user types:

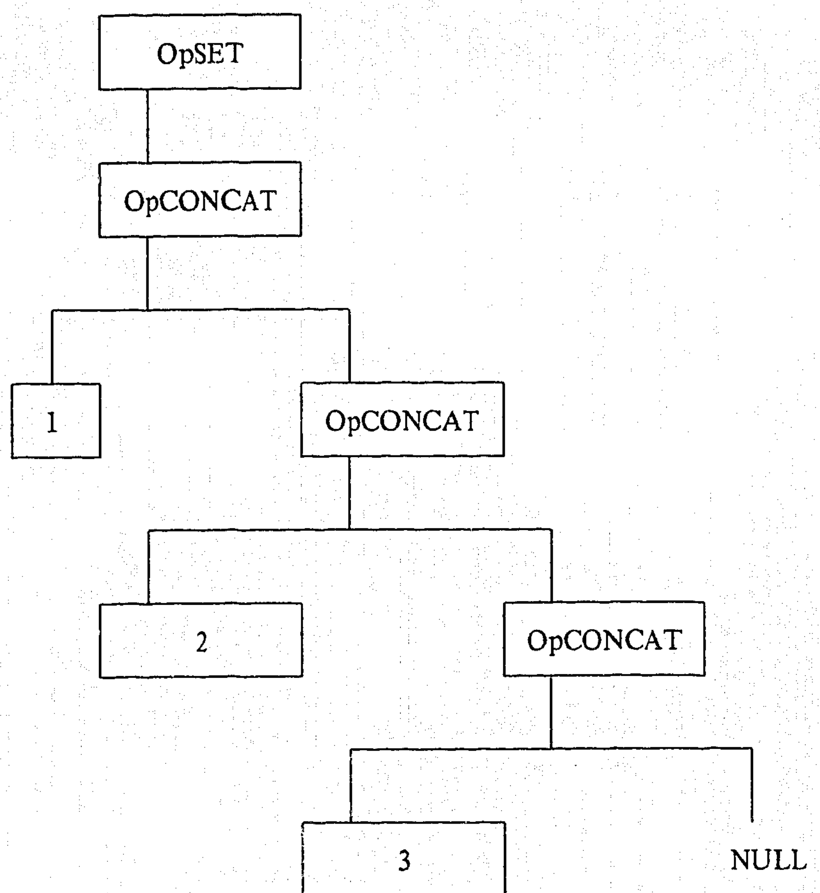
{1, 2, 3}

for a simple list with three numbers as elements, this list can be constructed in one of two ways:

(1) Start with an empty list. For each new element, find the end of the list, and append the new element.

(2) Start with an empty list. Working backwards, from the last element to the first, put each element at the front of the list. When all elements are on the list, add a list header.

The first method is left-recursive, and involves repeatedly finding the end of the created list (unless extra tail pointers are maintained). The second method is right-recursive, and allows most of the work to be done by the structure of the parse tree:



While the list is built from right-to-left, the user may have elements which need to be executed left-to-right (i.e. function calls or assignments). This is handled by evaluating the left side of a node, recursively evaluating the right side, and then joining the elements together as the recursion unwinds. Right recursion joins elements by setting one pointer and returning; see Chapter 3. (The interpreter has a huge stack for intermediate values, so this doesn't cause a problem.)

5.6. Functions and Procedures

Function definitions are nothing more than named parse trees with local variables. Functions have zero or more parameters. Each parameter is a "local variable" in the sense that the function may change the value of the variable, but this change does not affect the caller — with one exception. The first local variable always has the same name as the function. If a value is assigned to the function's name, then this value is returned to the caller as the function's result.

The names of the local variables are stored in a local symbol table, along with offsets into the parameter list. When a variable is referenced in a function, the local symbol table is searched before the global symbol table. If a local symbol is found, then a parse tree node of type *OpNAME* will point to the local symbol table entry. If a global symbol is found, then the node will point to the global entry. If no symbol is found, then a new global symbol is created with the default value of NULL.

When a function calls another function, the number and the type of the parameters are *not* checked at compile time: there is no way of knowing whether the new function will be redefined before the function call is actually executed. Similarly, a function may or may not return a result. Pascal distinguishes (at compile time) between functions and procedures by whether a result is returned. Arbitrary redefinition allows what is now a function to become a procedure, and vice versa. Mistakes allow a function to sometimes return a result and sometimes not. Hence, functions and procedures can not be distinguished in this language at compile time, and should be treated as the same class of objects: functions.

5.7. Error Productions

There are no semantic errors in the grammar: any statement which is syntactically correct has semantic meaning. Because almost every variable and function can be redefined at any time, there is no point in checking the "types" in an expression against the current values assigned to variables. This is particularly true in functions where a global variable may be referenced (say, in an division operator) with a current value which is inappropriate (say, a list). To generate a semantic error message would be to assume that the global variable will not be changed to a more appropriate value before the function is executed.

Syntax errors are handled by YACC error productions. YACC is notoriously poor at handling errors in arbitrary grammars, so the secret is to design a grammar which agrees with YACC's limited abilities. All statements end with a semicolon (;), including function definitions. Extra semicolons are allowed (that is, null statements are ignored). Semicolons are not used anywhere else in the grammar. Only one statement or function definition is parsed on each call to *yyparse*. Hence, skipping past a semicolon will lose at most one statement. At this point, it should be safe to call the parser for the next statement.

(By making the semicolon into a unique token with only one purpose, YACC does not need a look-ahead token to recognize the end of a statement. Further, since no other token begins with a semicolon, LEX does not need a look-ahead character to recognize the semicolon as a token. Hence, at the end of each statement, neither LEX nor YACC have buffered input. This is what makes it safe to return from YACC's *yyparse* after every statement. Nothing can be lost before the next statement is begun. The same safety applies to redirecting LEX's input when the *load* function starts reading from a new file: LEX's input file pointer *yyin* is replaced by a new file pointer, and LEX never notices!)

A sample error production for the addition operator is:

```
| expr '+' expr
    { $$ = MakeParse(OpPLUS, $1, $3, NULL, NULL); }
| expr '+' error err_expr_plus ';'
    { YYABORT; }
```

If the addition has proper syntax, then a parse tree node is created. Otherwise, after an expression and the definite token "+" have been found, the error production "err_expr_plus" is called, and YACC starts to look for the next semicolon. The error production looks like this:

```
err_expr_plus :
        { skippy("error after '+' in expression"); }
    ;
```

This production has a null rule, which means that it always gets reduced. Upon finding a syntax error, YACC reduces this rule, causing the function *skippy* to be called with the string "error after '+' in expression". *skippy* is the brand name of a peanut butter; it is also the name of a function which prints a caller's error message followed by the warning "skipping to next semicolon ';'". Then YACC goes back to the previous rule, which now looks like this:

```
| expr '+' error ';'
    { YYABORT; }
```

Whenever the special token *error* is followed by a character, YACC throws away all input until it finds that character, and then reduces the production. Reducing this error production forces the parser to abort with an error code returned to the caller (YYABORT). The calling program must check the error code, ignore the incomplete parse tree, and then call for the next statement.

The null rule is a clever way of printing an error message and warning the user to type a semicolon *before* YACC starts throwing away input. The user may find this funny, in that the compiler is telling him how to fix a problem already known to the compiler, but this is a helpful way to produce YACC error messages in an interactive environment.

Syntax error productions were added to the YACC grammar as follows: First, an error production was added to the initial production ("program") as a last chance error recovery when the input fits no rules. Second, after a definite token (not an optional token) is found, then all productions which use the same token in the same place are given the same error production. Third, no error productions are placed after a definite token which is followed by an optional clause (such as the *then* clause in an *if* statement) — because YACC's default action in optional productions is to reduce the production and to postpone recognizing errors until a required production fails.

5.8. Other Features

The YACC grammar contains some productions which are not documented for the users, because they are mostly of interest to the author:

(1) Null statements are ignored when possible (see the "stmt_list" production), since statements are a relatively expensive node in the interpreter.

(2) Missing elements in a list are assumed to be NULL values (see "expr_slist"). For example:

```
{1, , 3}
```

is equivalent to the list:

```
{1, NULL, 3}
```

This introduces a minor quirk into the language where:

```
{ , , } eq {NULL, NULL, NULL}
  { , } eq {NULL, NULL}
    { } ne {NULL}
```

That is, a list consisting of two commas is equivalent to a list with three NULL elements, a list with one comma is equivalent to a list with two NULL elements, but a list with zero commas is still the empty list (zero NULL elements!).

(3) Missing parameters in function calls are treated in the same way as missing elements in a list: they are assumed to be the NULL value. This mirrors the treatment of function parameters by the interpreter.

(4) The Pascal style of begin/end blocks for compound statements is not used. *for* statements, *while* statements, and function definitions must have an explicit *do* before the compound statement, and an explicit *end* at the end. *if* statements have optional *then* and *else* clauses, but must be terminated by an explicit *end*. This leads to less confusion about what a piece of code means, and forever removes the "dangling else" problem common to many languages.

5.9. Test for Parse Tree Routines

The grammar for this classifier language does essentially one job: create parse trees. Even functions are mostly defined in terms of their parse trees. To test the grammar means to test the creation of parse trees. Another quick-and-dirty test program called "tepar" was written.

tepar repeatedly calls the YACC-generated function *yyparse*. The returned status is printed (zero for YYACCEPT and one for YYABORT). If a parse tree is returned, then it is dumped in a crude indented format. Each node in the parse tree is a structure of type *ParseThing*. Pointers are shown in hexadecimal and values in decimal or as strings (where possible). The following is an example for the list {1,2,3} shown in a previous diagram:

```
calling yyparse()
{1,2,3};
yyparse() returns 0

at e9c0 STMT two = e980
  at e980 SET one = e940
    at e940 CONCAT one = e800 two = e900
      at e800 NUMBER number = 1
      at e900 CONCAT one = e840 two = e8c0
        at e840 NUMBER number = 2
        at e8c0 CONCAT one = e880
          at e880 NUMBER number = 3
```

Here, *one* is the "left" child and *two* is the "right" child. NULL pointers are not shown, and must be assumed by their absence.

Like *telex*, *tepar* found mistakes in the grammar. Many were non-trivial. Early versions of the compound statements had more *OpSTMT* nodes than were necessary. Numerous operators had their child pointers in the wrong place. Some productions were not being reduced as expected. Many error productions didn't work, or were positioned incorrectly. All of these mistakes were found and corrected before the remainder of the compiler was written. Thus, the syntax grammar was a final feasibility test for the entire language before too much time and effort was expended on a design that was impractical.

6. The Assignment Operator

The design of the assignment operator comes before the design of the parse tree interpreter, because assignment affects the structure of the execution data.

6.1. The Assignment Problem

To allow expressions as complete statements, assignment is an operator, not a statement. Most operators want values for their operands (left side and right side); assignment needs an address on the left and a value on the right. To avoid a YACC conflict, both sides are reduced as expressions. Expressions in a compiled language usually produce values. However, this language is not compiled: it is interpreted, and is not restricted by what most compilers do. If there is a way of keeping address information associated with values so that assignment works correctly, then we can implement expressions as complete statements.

What objects are legal on the left side of an assignment? First, variable names are legal; we always want to be able to assign a new value to a variable:

name **:=** *expression* **;**

where *expression* may return any value (including the NULL value). Second, the elements in a named list are assignable:

name **[** *index* **]** **:=** *expression* **;**

Third, if the indexed element in a named list is another list, then its elements are also assignable. (That is, the list indexing operator recursively preserves the assignability of the left side.) Fourth, are indexed characters in a string assignable? This would be a reasonable definition. Unfortunately, the way strings are stored makes substring assignments difficult: strings are not sets of individual characters which can be easily changed; they are packed sequences of bytes which must be modified in place. (This is a consequence of having strings as a basic object rather than characters. Please note that strings can be modified by subscripting the beginning of the string, concatenating a new middle with "+", and concatenating the old subscripted end.)

Thus, we have the following rules for deciding which expressions are assignable:

— named variables are assignable,

— the list indexing operator preserves assignability.

All other operators cancel the assignability of the operands. Hence, we need an assignment strategy which allows a relatively small number of operators to explicitly create or preserve assignability, while having a default action which prevents assignment.

6.2. General Solution

Global variables are stored in *ValueThing* structures. One *ValueThing* holds one number, one pointer to a string, or one pointer to the first element in a list (see Chapter 3). Value structures are linked together to form complete lists. The names of the global variables are stored in a global symbol table. Symbol table entries are of type *SymbolThing*. Each entry points to its name, its value, and the next symbol table entry. Assume, for the moment, that a symbol table entry points to its value via a field declared as:

```
ValueThing * value;
```

Further, assume that *sym* is the address of a *SymbolThing* and that *val* is the address of a new *ValueThing*. Then to assign a new value to the global variable, the following steps are required:

```
FreeValue ( sym->value ) ;
```

That is, free the space allocated to the old value, and then:

```
sym->value = val ;
```

This works for all data objects, even the NULL value when it is represented by the NULL pointer.

If all assignments were to global variables, then the assignment problem would be solved. Expressions are evaluated on a stack (described later, for now assume that the stack *points* to the value, as compared to containing the value). Stack entries of type *StackThing* could point to the current value and a symbol table entry. By default, the symbol table pointer would be NULL, meaning that the stack entry is not assignable. When a global variable was used in an expression, its current value would be placed on the execution stack along with the address of its symbol table entry. If the next operator in the expression was an assignment, then it would evaluate the right side, leave this result on the stack, and change the symbol table entry to point to the new value. All other operators would ignore the symbol table entry.

Complications arise when trying to assign values to elements in a list. A *ValueThing* of type *ValELEMENT* is not valid data object by itself. Symbol table entries point to valid data objects. Hence, some other method must be used to assign elements.

6.3. Specific Solution #1

To assign global variables, list elements, and later local variables, one solution is as follows:

Give each entry in the execution stack a value pointer, a flag, and an assignment pointer. If the flag is zero, then the entry is not assignable. If the flag is marked "global", then the entry is assignable, and the assignment pointer is the address of a symbol table entry. (Assignment would be done by changing the *value* field in the symbol table entry to point to the new value.) If the flag is marked "local", then the entry is assignable, and the assignment pointer is the address of a variable local to a function call (the fields of which have not been explained yet). If the flag is marked "element", then the entry is assignable, and the assignment pointer is the address of a *ValueThing* of type *ValELEMENT* (change the *this* field).

This solution looks messy, but all of the tricky code appears only once in the assignment operator.

6.4. Specific Solution #2

A better-looking solution can be found by considering changes to the data structures:

Force all stack entries, symbol table entries, and value structures to have the same format (say, something called *ThingThing*). Then no assignment flag is necessary on the stack. If the assignment pointer is NULL, then the stack entry is not assignable. If the assignment pointer is not NULL, then it is the address of a *ThingThing* which can be assigned a new value by changing a pointer field with a common name (say, *value*).

This solution works by virtue of its brute-force approach to memory management. Combining the fields of many structures into one common structure does one of two things: (a) greatly increases the amount of memory required; or (b) creates a confusing number of "C" *union* declarations to annoy the programmer who has to type in and debug the code. Both side effects are unacceptable. (This is the programmer speaking!)

6.5. Specific Solution #3

This solution is really a stepping stone to the final solution:

Instead of assigning values by exchanging pointers, overwrite the contents of the existing value structure. This appears to reduce the number of dynamic memory allocations and de-allocations, and to be safe in the sense that any list currently pointing to an element will continue to point to the same element after assignment. (That is, there is no assumption of data being used uniquely!) Stack entries will need two fields: a value pointer and an assignment flag. Assignment is legal if the flag is non-zero, and proceeds by releasing any memory pointed to by the current value (if a list or string), and then replacing all fields in the value structure with fields from the new value.

The NULL value will cause problems. It is no longer acceptable to use a NULL pointer to indicate that a value is the NULL value, because this NULL pointer would be pushed onto the stack where a value structure pointer is required. Attempting to reassign a value from NULL to anything else would fail because there is no old value structure to overwrite. The NULL value could be removed from the language; then some other default value would have to be given to variables which are used before being assigned (the number zero is reasonable). Keeping the NULL value means changing its definition. A new type of *ValueThing* must be created called *ValNULL*, which does not use the information in any of the fields, but is processed in the same way as the other data types. This causes two new problems:

(1) The language now has an external NULL value which differs from the implementation's internal ("C") NULL pointer. The confusion does not affect the user, but may create numerous obscure bugs in the compiler when the author forgets which value is which. (Of course, a much different name could be chosen for the external NULL value — like "nil"!)

(2) All variables must be initialized to point to a *ValNULL* value structure. This structure must be distinct, since it may get overwritten. The initialization must always be done, even if the next operand is an assignment to a new value. The same applies to all entries on the execution stack: they must be defaulted to a *ValNULL* value. Hence, many *ValNULL* values will be created for the sole purpose of being destroyed.

Compared to the next method, this solution confuses the programmer, uses CPU time to move fields from one structure to another during assignment, and still manages to create and destroy just as many dynamic value structures.

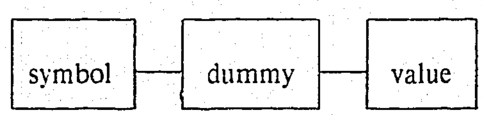
6.6. Final Solution #4

Solution #1 proposed that the execution stack should have a different pointer for each type of object that can be assigned. That is, if the current value is the "child" record, then the stack should also point to the "parent" record.

Solution #2 proposed that all structures should have a common format. This wasted space, but introduced an idea which will be used here: Assignments are easy if the execution stack points to a structure which has a consistent format.

Solution #3 proposed that the NULL value is most useful when it is the same as the internal NULL pointer.

Consider the following idea: Put a dummy structure between the global symbol table entry and the global variable's actual value:

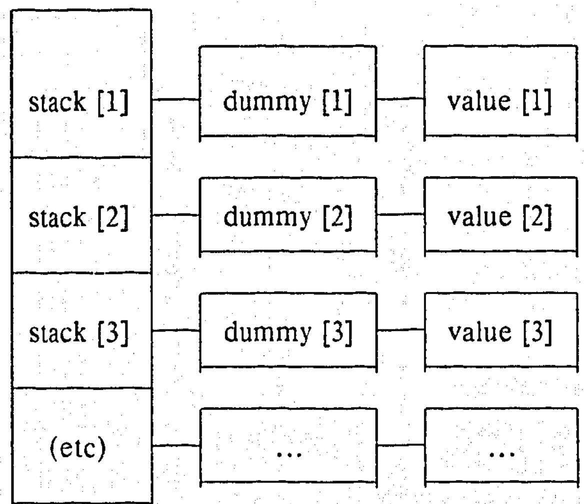


Now make this dummy structure to be something of type *ValueThing*. We already know that list elements are of type *ValueThing*, and point to the element's value through the *this* field. If the dummy structure points to the variable's value through its *this* field, then both elements and global variables will now point to their values through the same field name in structures of the same type. Assigning either one becomes an identical operation (removing at least one messy step from solution #1):

```
owner->this = val ;
```

where *owner* is the address of something of type *ValueThing*. This keeps the property that assignments are done by the quick operation of replacing one pointer (after freeing any old value, of course).

The execution stack is basically a big array of local variables. If we use the same dummy value trick, then the stack will look like:



Stack entries (the left column above) have fields for:

```
struct ValueThing * dummy;
struct ValueThing * owner;
```

dummy is the address of the dummy structure which points to the actual value for this stack entry. *owner* is the address of the value structure which "owns" the value pointed to by this dummy structure. Normally, *owner* points to another dummy structure either on the stack or attached to the global symbol table.

Assigning a local variable becomes one more case of the same thing: When a variable is referenced, the address of the variable's value is attached to the *this* field in the stack's dummy structure, and the address of the variable's dummy structure is put in the stack's *owner* field. All operators which expect to see a value can get the value from the stack via the *dummy* field. The assignment operator ignores the value; instead it checks the *owner* field. If non-zero, the assignment frees the old value pointed to by the owner's *this* field and attaches a new value.

This method of assignment has three advantages: First, the operands are general expressions (not special grammar productions). Second, assignment is done by replacing pointers and not by moving data. Third, the NULL value retains its useful properties. The major disadvantage is that a dummy structure is inserted between many objects and their real data, which uses more space, and also causes a lot of repeated "dummy->this" typing in the "C" code.

6.7. Assignment Example

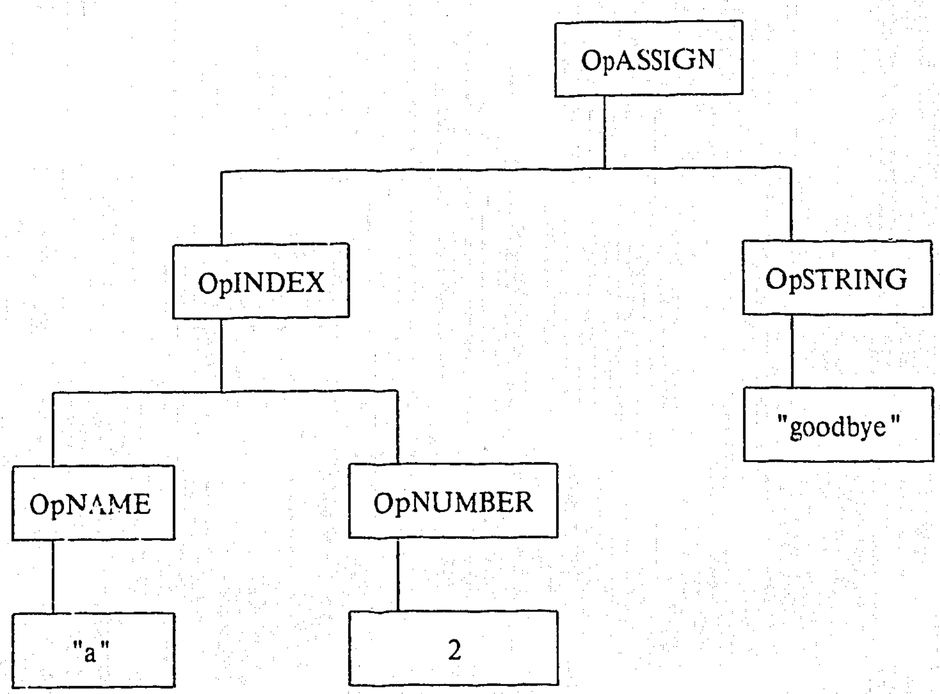
Let *a* be a global variable that is a list, such as the example used in Chapter 3:

```
{1, "hello", {3, 4}}
```

The parse tree for the statement:

```
a [ 2 ] := "goodbye" ;
```

looks like this:



To execute the assignment, the following steps are performed: The assignment operator (*OpASSIGN*) is called. The first action of *OpASSIGN* is to recursively evaluate the left side. This calls the index or subscript operator (*OpINDEX*). *OpINDEX* pushes a NULL value onto the execution stack to reserve an entry for its result. Assume that this is stack entry number 24:

```
Stack[24]
```

Then stack entry #24 becomes:

```
Stack[24].dummy->this = NULL;
Stack[24].owner = NULL;
```

OpINDEX recursively evaluates its left side, which invokes the name operator. *OpNAME* points to the symbol table entry for the global variable *a*. Let *sym* be the address of this symbol table entry. A new stack entry is created (#25) and changed so that the dummy structure for #25 points to the value of *a*:

```
Stack[25].dummy->this = sym->dummy->this;
```

and so that the owner field for #25 points to the dummy structure for *a*:

```
Stack[25].owner = sym->dummy;
```

This completes the execution of *OpNAME*. Going back to *OpINDEX*, the right side is now recursively evaluated, invoking the number operator. *OpNUMBER* creates a new stack entry (#26) to point to a value structure containing the number 2. *OpINDEX* pops this subscript off the stack, saves it in an internal variable, and finds the address of the second *ValELEMENT* structure pointed to by #25 (the list *a*). Let *val* be the address of this element structure. Because stack entry #25 is assignable, the result of *OpINDEX* at #24 is also assignable:

```
Stack[24].dummy->this = val->this;
Stack[24].owner = val;
```

(Remember that list elements point to their values through the *this* field.) Entry #25 for *a* is popped off the stack, and *OpINDEX* returns. *OpASSIGN* evaluates its right side, calling *OpSTRING* to push a string value structure ("goodbye") onto the stack at entry #25. The assignment operator now has a left side and a right side. The left side (#24) is checked to make sure that it is assignable (it is). The old value pointed to by the owner of the left side is released:

```
FreeValue ( Stack[24].owner->this );
```

The new value (at #25) is attached to the stack at #24 — allowing the result of the assignment to be used again in an expression:

```
Stack[24].dummy->this = Stack[25].dummy->this;
```

The owner of the left side (the second *ValELEMENT* in *a*) is also given the new value:

```
Stack[24].owner->this = Stack[25].dummy->this;
```

This completes the assignment. The new value of the global variable *a* is:

```
{1, "goodbye", {3, 4}}
```

A few details have been omitted here, most of which involve dynamic stack data.

7. Interpretation and Execution

Parse trees are executed by an interpreter; nodes in the parse tree become executable functions. The choice of the nodes, and the functions they represent, is based on an assumed execution model. This model limits the size of the implementation, while at the same time providing an acceptable level of service to the user.

7.1. The Execution Stack

Deep inside the interpreter are some thirty different operators. Each operator can perform more than one function, depending upon the types of the operands. Individual operators ask three questions:

— where are the operands (left and right sides)?
— how are they processed?
— where does the result go?

The first and third questions are really the same question, because the result from one operator may be the input to another operator.

Consider the case of the addition operator ("+") for numbers. It has one operand on the left side of the "+", one operand on the right side, and returns one number as its result. Before the addition can proceed, the left and right sides must be evaluated and put in some location known to the operator, and a location must be allocated to hold the result. We could have the caller (parent node) perform the evaluation and allocation — but this would require every parent node to know the number of operands for each child node. That is, all operators must know about all other operators. This is hardly reasonable in a language where changes should be possible without a major reprogramming effort.

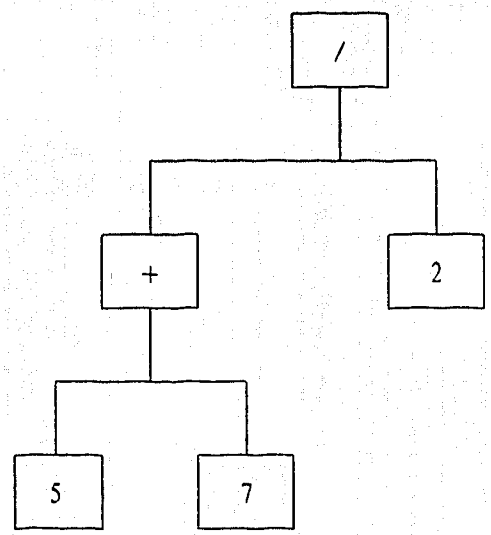
If the parent of a node can not be asked to evaluate operands or allocate results, and the children of a node are clearly in no position to do this, then the node itself must take care of all operand and result related details. Allocating space for a result is easy: use the same facilities that already exist for creating dynamic data objects. Returning the result to the calling node is more difficult, since the result must be placed where the caller can find it.

Look at the problem from the viewpoint of a node as it evaluates its operands. The left side of an addition may be a simple number, in which case the evaluation is trivial. The left side may also be an expression, in which case the evaluation is best done by calling the appropriate operator to evaluate the expression. Where does this new operator node return its result? We could supply the address of a static location, if we were willing to accept that an addition operator could not have another addition in its operands. (Otherwise, the static location would be overwritten.) We could supply the address of a dynamic location, if we were willing to spend a lot of time creating and destroying dynamic data (which a previous chapter cautioned against). Not being able to supply either a static or dynamic location leads to the interesting conclusion that we should not be supplying an address for the result!

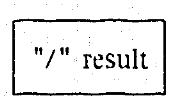
Operator nodes must know where to find their operands and where to put their results without being told. This rules out any sort of register or other fixed-address allocation scheme. We need something where operands have variable addresses, but still can be accessed in a predictable manner. One such method is a stack. Consider the expression:

```
(5 + 7) / 2;
```

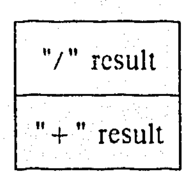
which has the parse tree:



Assume that the stack is initially empty. When the division operator ("/", alias *OpSLASH*) is called, its first action is to allocate a stack entry for its result. This gives the operator a convenient place to build up the resulting data structures, and makes sure that there is no unattached dynamic memory in the event of an error:



where "result" means that this part of the stack is occupied, but currently has no value. Then the addition operator ("+", alias *OpPLUS*) is called. Like the division, space is allocated for a result:



Next, the left operand of "+" is evaluated. This is trivial, because it is an explicit number:

"/" result
"+" result
5

Now the right operand is evaluated:

"/" result
"+" result
5
7

The addition operator now has both of its operands. They are combined into a result (12), which replaces the dummy "+" result already on the stack:

"/" result
12

Control returns to the division operator, which evaluates its right side:

"/" result
12
2

Division may now be performed, and the dummy "/" result is replaced by the calculated result:

6

While this may look simple, there are some serious considerations implied by the use of stacks in an execution model.

First, a great benefit is achieved by allocating space for a result and then sequentially (recursively) evaluating each of the operands. When an operator is called, the stack will have a certain position. The operator does not need to know in advance what this position will be. Its result will go into the next position (that is, at an offset of zero from the position of the stack as of when the operator was called). Its first operand is evaluated on the stack, and leaves a result at the next stack position (no matter how complicated this evaluation may be). The second operand similarly goes in the stack position which has an offset of two relative to the beginning of the operator. At no time does the operator need to know how big the stack is, or what other results are already on the stack.

Second, all operators must conform to this stack model of execution. The language must be designed for recursive evaluation. The grammar must ensure that parse tree nodes requiring values for their operands have valid expressions as the child subtrees. No attempt should be made to avoid complete evaluation of expressions, such as in a "short circuit" mode where logical AND is assumed to be false if the left side is false (without looking at the right side). Operators are not allowed to leave additional information on the stack, only their result. Child operators may not question the actions of their parent operators; however, they may be selective in invoking their own children.

These restrictions do not appear to be too severe in an interactive programming language where the user will be writing simple programs. They might become annoying in a more general case. Think about the rule which says (paraphrased), "children must only speak when spoken to". A child node is not allowed to change the actions of its parent node. In the "C" language, *while* statements may contain *break* statements. The effect of a *break* is to immediately exit from the enclosing *while* loop. This is an example of a child (*break*) affecting the actions of its parent (*while*), and is prohibited here.

7.2. The Stack Entries

The stack is the basic execution model. It was chosen so that operators do not have to perform their own allocation and evaluation. The entries in the stack are assumed to be addressed starting from some relative stack pointer (SP). How big each entry is, and what information it contains, has not been explained.

All values pass through the stack at some point. Each stack entry should be capable of holding one value. Values in this language are numbers, strings, lists, and the NULL value. Numbers have a fixed size, and are easy to accommodate. Strings vary in length, and must be pointed to. Lists also vary in total size, and must be pointed to. The NULL value requires no space, only a NULL pointer. If stack entries were required to contain an entire value, then they would have to be of variable size. This would make addressing difficult, since the second (evaluated) operand for an operator could not be accessed unless you knew the size of the first operand. (Further, stack entries would not have the same structure as dynamic data objects, which would be messy.) Hence, stack entries have a fixed size, and point to their values.

One stack entry is defined by a structure called *StackThing*:

```
typedef struct StackThing {
    struct ValueThing * dummy;
    int free;
    struct ValueThing * owner;
} StackThing;
```

dummy points to a dummy value structure, which in turn points to the real value for this stack entry. *free* is a dynamic data flag which is either *YES* or *NO*. *owner* is the address of a value structure which "owns" the value pointed to by *dummy*, or else NULL if there is no owner. *dummy* and *owner* are fully explained in the previous chapter.

The whole stack is an array of *StackThing*'s:

```
StackThing Stack [ STACKSIZE ];
```

where *STACKSIZE* is some large number (currently 999). The current entry in the stack is indexed by a variable called "SP":

```
int SP;
```

All access to the stack should be relative to the current stack pointer.

Two common routines for stack management are *PushStack* and *PopStack*. *PushStack* "pushes" the stack down to create a new entry. This entry is given a dummy structure which points to a NULL value, and has no owner. The stack pointer (SP) is changed to this entry number, so that:

```
Stack [ SP ]
```

refers to the new entry. *PopStack* "pops" the last stack entry, releasing any dynamic data, and decrements the stack pointer to be the index of the previous stack entry.

Two other stack management routines are *CheckStack* and *ClearStack*. *CheckStack* is called by *PopStack* and *PushStack* to check that the current stack pointer is within a legal range. *ClearStack* is called once to initialize all stack entries to a consistent state, which ensures that all pointers are either NULL or point to something legitimate.

7.3. Dynamic Data

As entries are pushed and popped from the stack, it is necessary to allocate and de-allocate dynamic storage.

Once again, consider the addition operator ("+"). When adding numbers, the operands may point to value structures which are owned by another value structure (a global variable or a list), or are dynamic (not owned). For the operands in an addition, this is not important: *OpPLUS* picks out the numbers, adds them, and then returns a result. The result is dynamic ("free"), because nobody owns it. If the result is used in a further calculation, such as another addition, then it is no longer needed, and must be de-allocated when it is popped from the stack. If the result is assigned somewhere, then it ceases to be "free", and must not be de-allocated.

Stack entries contain a *free* field which is *YES* if the value pointed to by the stack entry is dynamic, and *NO* if the value is attached somewhere. *PushStack* defaults this field to *YES*; operators which retrieve named variables (*OpNAME*) set this to *NO*; operators which produce new results reset this to *YES*. *PopStack* checks this field before popping a stack entry. If it is *YES*, then the value pointed to by the stack *dummy* field is released. In all cases, the dummy value pointer is then set to NULL, and the stack entry ceases to have any connection with the value.

There is a difference between the *free* and *owner* fields. *free* decides if a stack entry must be de-allocated after use; *owner* decides if a stack entry can be assigned. The two perform similar functions, except in the case of special user classifier variables which are read-only: *free* is *NO* and *owner* is NULL. This allows the variables to be used by any operator, except assignment.

7.4. More General Execution

The module responsible for interpreting and executing a parse tree is "faexe.c" (see Appendix B). The main function is *ExecParse*. Given a pointer to a parse tree, *ExecParse* performs a few administrative details, and then passes control to the individual operators via a "C" *switch* statement. (Each node in the parse tree has an operator type as one of its fields.) Most of the code is for the operators, and they all work relative to the current stack position. When an operator needs to evaluate one of its child nodes, it calls *ExecParse* again. Hence, *ExecParse* is also stack-relative, and is capable of recursively evaluating complete parse trees. Eventually, all parse trees end in leaf nodes which have no children: nodes for pushing named variables, strings, numeric constants, etc.

Operator nodes of special interest are:

(1) The assignment operator (*OpASSIGN*) evaluates its left side — as a value — and then its right side. If the stack entry for the left side still has an *owner* pointer, then the assignment is performed. Otherwise, an error occurs.

(2) The *for* statement (*OpFOR*) evaluates its loop expressions (*from*, *to*, and *by* clauses), checks that they are acceptable numbers, and then performs an internal "for" loop with this information. Moving the limits inside the interpreter saves the overhead of re-evaluating them at each repetition of the loop.

(3) The *if* statement (*OpIF*) evaluates its conditional expression (first child). If the result is 1, then *ExecParse* is called to execute the *then* clause (second child). If the result is 0, *ExecParse* is called to execute the *else* clause (third child). Otherwise, an error occurs.

(4) Unary negation (minus or *OpNEGATE*) tries to avoid copying its operand, since this may be a large list. Instead, after evaluating its right side, it checks if the operand is "free". If not, a dynamic copy is created with the *MakeDynamic* routine. Then the sign is negated in place.

(5) The statement operator (*OpSTMT*) saves the current stack pointer before executing its child. When the child returns, the new stack pointer is compared against the old. If one value is left on the stack, it is printed (if the child is not an assignment) and popped from the stack. Then if the two pointers are not equal, an error is detected. Since almost all operators are descendants of a statement node, this catches operators which are using the stack incorrectly.

(6) *while* statements (*OpWHILE*) are similar to *if* statements, except that the body of the loop is executed as long as the conditional expression evaluates to 1.

Finally, there is a function called *ExecFile* which is responsible for executing an entire file. As complete statements are parsed (with *yyparse*), they are passed to *ExecParse* for execution, and then de-allocated. Error conditions are caught, in a way which will be described later, and allowance is made for *ExecFile* to be called recursively. That is, a statement in the current file may call the *load* function, which causes a new copy of *ExecFile* to start reading from a new file. When this new file is finished, the new copy of *ExecFile* must restore all necessary internal variables to their previous values.

8. Functions and Local Variables

Functions are operators with any number of children. These children are called "parameters", and appear in a function "parameter list". The implementation of functions corresponds so closely to the regular operators that functions are in fact a trivial operator known as *OpFUNCTION*.

8.1. Function Example

An example of a function call is:

```
a := 5;
write(a, a+2, a*a, sign(a));
```

which calls the pre-defined *write* function to write out four parameters: the value of the variable *a*, the value of *a* plus two, the square of *a*, and the sign of *a*. The sign is obtained by calling the pre-defined *sign* function and using the result as a parameter to the *write* function.

The child of an *OpFUNCTION* node is a left-recursive tree of parameter nodes (*OpPAR*). Any expression is a legal parameter. *OpFUNCTION* allocates space for a function result (which defaults to the NULL value), generates the parameters from left to right, calls the function, and then pops the parameters off the stack. For the *write* example above, the following steps occur. A stack entry for the result is created:

"write" result

write does not return a result, but *OpFUNCTION* does not know this, so it still allocates an empty stack entry. Next, the value of *a* is pushed onto the stack (the number 5). Ignoring the intermediate call to the *OpNAME* operator, the stack now looks like this:

"write" result
5

The second parameter is an expression. The addition operator is called, allocates space for a result, and pushes its operands onto the stack:

"write" result
5
"+" result
5
2

After the addition is complete, only the result is left:

"write" result
5
7

Similarly, the multiplication operator is called to produce:

"write" result
5
7
25

Calling *sign* involves a function call inside the incomplete function call to *write*. As before, allocate space for a result and push the parameter *a*:

"write" result
5
7
25
"sign" result
5

sign returns 1 for the sign of a positive number. Immediately before *sign* returns, the stack looks like:

"write" result
5
7
25
1
5

After returning, the parameter to *sign* ($a = 5$) is popped off the stack:

"write" result
5
7
25
1

Now the parameters to *write* are ready, and *write* is called to write the values on standard output:

```
57251
```

which is cryptic, since we forgot to include spaces between each number! *write* does not return a result, so the stack entry for its result should really be shown as the NULL value (which means that the dummy structure for this stack entry has a NULL value pointer):

NULL
5
7
25
1

After *write* returns, *OpFUNCTION* pops the parameters off the stack, and returns to its parent, leaving the NULL function result on the stack:

NULL

If you try this example, you will find that "NULL" is not printed on your terminal, because the parent node of *OpFUNCTION* is a statement node (*OpSTMT*), which throws away NULL function results — even if NULL is the value you want to see! Functions without a result are more properly known as procedures (in the Pascal style of naming); by defaulting the result to NULL and then throwing NULL away, procedures are made to look like functions, and the language has one less object.

8.2. Parameter Lists

Function parameters are expressions and can take on any value. No data type checking is performed, because this language does not support explicit type declarations. The called function is expected to test the validity of its parameters, and to generate an error message if they are unacceptable. This testing may be done by implicitly assuming that the parameters are legal, and letting the language force an error condition if an illegal operation is attempted.

The language does support a minimum number of parameters. If you call a function that has four parameters, and you supply only three parameters in your parameter list, then an extra NULL parameter is automatically created for the fourth parameter. No error message is generated. The reason is as follows: The language does not know what the function will do with the missing parameter. It may be optional, in the sense that it only gets used when certain combinations of the first three parameters appear. It may be required, in which case supplying a NULL value makes it safe to reference the stack at this point; however, the NULL value will probably generate an error if it is used.

Extra parameters are also legal: calling a four-parameter function with five parameters is allowed. All parameters are pushed onto the stack, the function is called, and then all parameters are popped from the stack. The extra parameters may serve some purpose as they are being evaluated, but the called function will have no way of referring to them.

Why should this be allowed? There are many pre-defined functions which have a variable number of parameters. Most of these functions need at least one or two parameters (such as the format string in *printf*), but accept more parameters. For example, *write* may have zero or more parameters. Hence, the parameter count stored in the definition of a function is treated as a minimum number of parameters. No maximum is enforced. It is the user's problem if he supplies too many parameters. Far from being unfriendly, this allows the user to do pretty much as he wishes.

8.3. Local Variables

A local variable inside a function definition is a way of naming a parameter in the parameter list. (All local variables or parameters are stored on the stack so that functions may be recursive.) Consider the following function definition:

```
function plus ( left , right )
do
    plus := left + right ;
end ;
```

which is a named version of the addition operator ("+"). If we call *plus* to add the numbers 2 and 3:

```
plus ( 2 , 3 ) ;
```

then the stack will look like this after the parameters have been pushed:

"plus" result
2
3

We want the local variable *left* to refer to the number 2, and the local variable *right* to refer to the number 3. *plus* is also a local variable, and refers to the result. In this example, the stack pointer (SP) is pointing to the entry for the number 3. We could make *right* equivalent to:

```
Stack [ SP ]
```

make *left* equivalent to:

```
Stack [ SP-1 ]
```

and make *plus* equivalent to:

```
Stack [ SP-2 ]
```

(This is the negative indexing trick used by YACC for its stack.) Unfortunately, the language does not guarantee that there are only three values on the stack, because extra parameters may have been pushed, and operations inside the function may be using stack space. We need a parameter reference mechanism which does not depend upon the relatively fickle value of the stack pointer.

A new stack pointer is created: the "frame pointer" (FP). A "frame" is a complete set of function parameters. When a function is called, the frame pointer is the index of the stack entry for the result, the frame pointer plus one is the index of the first parameter, etc. For the previous example, the stack would look like:

FP+0 =	"plus" result
FP+1 =	2
FP+2 =	3

If unnecessary parameters are pushed onto the stack, they appear at index FP+3, FP+4, and so on. Intermediate values from expressions inside the function also appear at later stack entries, but do not affect the frame pointers. Hence, this is a fixed way of using parameters inside a function, no matter where the references may appear. The symbol table entry for a local variable contains its frame pointer offset.

Sometimes it is desirable to have local variables which are not parameters, so that these local variables may be used for temporary storage. Once again, this language does not have variable declarations. How should local variables be declared? Remember that the language will supply NULL values for any parameters which are omitted by the caller. The obvious solution is to declare all temporary variables as parameters, and to not tell the caller that they are in the parameter list. This requires absolutely no additional work in the implementation, and removes any need for a special way of declaring local variables.

8.4. Passing Parameters

To this point, the classifier language is capable of passing parameters both by value and by address (since the stack contains the address of the value!). If passing by address is supported, then a function will be able to change the caller's parameters when the normal conditions for assignment exist. The grammar does not presently have a way of identifying which parameters should be passed by value, and which should be passed by address. If someone finds a legitimate use for changing a caller's parameter, then the grammar can be modified and *OpFUNCTION* changed so that it no longer assumes passing by value. (All references to *SymLOCAL* should also be checked.) Until then, parameters to user-defined functions are made "free" with the *MakeDynamic* routine. When a function changes a parameter, it is changing the "free" copy, which gets de-allocated when the function returns.

Parameters to pre-defined functions such as *write* are passed in whatever form they get pushed onto the stack. Almost all pre-defined functions return information only through their result, and have no need to modify their parameters. Not enforcing call-by-value saves a lot of time, especially for the arithmetic functions (*abs*, *random*, *sign*, *size*), and leaves enough address information on the stack so that the formatted I/O routines can pass back data through their parameters. Of course, pre-defined functions are similar to operators in that they must not violate the dynamic allocation status of the stack entries.

9. Error Recovery

Errors are detected at five levels: lexical, syntax, semantic, execution, and internal. Each level provides a different degree of explanation and recovery.

9.1. Lexical Errors

Lexical errors occur while trying to break the input stream into tokens. Most tokens are clearly defined by LEX rules: either the input matches a meaningful token, or it matches the default rule and gets returned character by character. Only one token is capable of generating error messages: strings.

A string is a sequence of characters enclosed in quotation marks. To prevent a single typing mistake from swallowing up an entire program, strings are not allowed to include explicit newline characters. (If you want a newline inside a string, use the "\n" escape sequence.) Omitting the closing quotation mark will print the error message:

```
newline in quoted string (")
```

and the string will include only those characters appearing before the newline. The parser is not told about this change to the input; the altered token sequence may be a perfectly legal statement, and the user may get something other than what he was expecting. (Lexical scanning is below the level of the more sophisticated syntax or execution error recovery. The altered input will probably cause a syntax error. A syntax error could be forced by returning a token not found in the grammar, such as *TokERROR*.)

Similarly, the lexical end-of-file character is not allowed in strings, and generates the error message:

```
end-of-file in quoted string (")
```

Corrective action is the same as for newlines.

While looking for a string, a large buffer of fixed size is used to hold the characters before they are copied into a variable-length dynamic buffer. If this buffer gets full, the user is told:

```
string longer than 999 characters, begins with "..."
```

where "..." is the beginning of the string. The string ends with the maximum number of characters, and any further input is treated as text to be broken into more tokens.

Associated with the lexical routines are a global variable called *LineNumber* and a function called *PrintLine*. For standard input (the terminal), *LineNumber* is negative to signify that no line number is meaningful. For files, *LineNumber* is the number of the current line in the current input file. The LEX rule for the newline character increments *LineNumber*:

```
\n          { /* newline */ LineNumber++; }
```

The function *PrintLine* prints a message saying:

```
at line 999:
```

where "999" represents the current line number, if input is from a file. Every error message generated by the compiler is preceded by a call to *PrintLine*. When combined with a subversion trick in *ExecParse*, this is an accurate way of pointing to the cause of an error — at any level.

9.2. Syntax Errors

Chapter 5 explains YACC syntax error recovery in great detail. Upon finding a syntax error, a message identifying the last legal token is printed, the user is warned, and then YACC is directed to throw away all input until the next semicolon. The line numbers in the error messages are obtained from the lexical level: there is no fiddling with the *LineNumber* global variable.

9.3. Semantic Errors

While reducing the YACC syntax productions, no semantic errors are recognized. If a statement has correct syntax, then a parse tree is successfully built, or a function is successfully defined.

9.4. Execution Errors

As a parse tree is being executed by *ExecParse*, the user may ask for something that can not be done (or is not implemented): division by zero, incompatible operands, bad parameters to a pre-defined function, etc. The only meaningful action is to abort the current parse tree.

One approach is to print a warning message, return an "undefined" value, and continue execution. If all operators recognized the "undefined" value as a message to quit, then this method would cleanly return back up the parse tree until the root was reached. Compared with the next method, the cost would be in having every parent node check the result of its child nodes.

Many operating systems provide a way of saving the current state of a program, and later returning to this state. The state is usually defined to be the general registers, processor status, and stack pointer. In UNIX, calling the "setjmp" routine saves the state and "longjmp" returns to this state. A sudden jump to a previous state only makes sense when the saved state occurs in a routine which is an ancestor of the routine asking for the jump (otherwise, the contents of the machine stack are garbage). If we want to be able to abort the execution of an arbitrary parse tree with a call to:

```
ExecAbort();
```

then there must be a routine which is always the eventual parent of any parse tree. *ExecParse* is not acceptable, because it is called recursively. Saving the state upon entry to *ExecParse* and then jumping back to this state would abort only the current node in the parse tree — not the entire tree. The routine we jump to should be the same as the routine which calls *yyparse* to create the parse tree. The name of this routine is *ExecFile*.

The basic control loop in *ExecFile* is this: Save the current frame and stack pointers. Call "setjmp" to save the current state. If "setjmp" returns a non-zero status, then "longjmp" has been called by *ExecAbort* to return to this state. If "setjmp" returns a zero status, then call *yyparse* for a parse tree and *ExecParse* to execute this tree. Should an error occur, the stack may need to be cleaned up, and the current input file may need to be closed. Otherwise, repeat. This guarantees that control will return to a point which is capable of correctly handling the error condition. (The same strategy could be used to trap an interrupt signal from the ATTN, BREAK, or control-C keys.)

One final trick is played by *ExecParse* to improve the line numbers reported by *PrintLine*. As parse tree nodes are created, the current *LineNumber* becomes part of each node. Before executing a node, *ExecParse* saves the value of *LineNumber*, subverts this to be the line number in the parse node, and then executes the operator for this node. After the operator completes, the old value of *LineNumber* is restored. *ExecFile* cooperates by adjusting *LineNumber* should an error occur. The effect is like having a clock which constantly changes to show the correct time for events that are being discussed.

9.5. Internal Errors

An internal error in the compiler occurs when an assumed condition is checked, and the check fails. Examples are the sign of a data structure (*CheckSign* routine) or the range of the stack pointer (*CheckStack*). Most internal errors are caught by "switch" statements which test all legal values for an identifier (such as the *ValXXX* data types), and the "default" case is invoked because of an illegal value. This happens quite frequently during development, but should never happen in a production version.

After an internal error is detected, the usual line number message is printed followed by "internal error", the name of the current function, and the value which caused the error. This may be enough information to duplicate the problem. No corrective action is taken, as none is known. The current function returns to its caller without doing what was requested. Some parse tree routines call *ExecAbort*. The compiler may continue to run, but should be considered damaged.

10. Pre-Defined Functions

Pre-defined functions provide the user with facilities beyond the basic expression operators. The names of these functions, their parameters, and their actions are fully documented in Appendix A; the discussion here is limited to explaining why the functions are included in this language.

New pre-defined functions are relatively easy to add, even if they are written in "C" and cause the compiler to be rebuilt, so the list here is a minimal collection which may be extended from time to time. A suggestion is to first write a new function in the `face` language. If the execution time is too slow, or if the function is used often enough, then rewrite it in "C".

10.1. Control Functions

The control functions are *exit* (alias *quit*), *load*, *save*, and *stop*.

exit performs the rather obvious task of exiting from the `face` program and returning to the parent process, which is usually the UNIX shell. While a user can achieve the same result by typing the end-of-file character (control-D) on standard input, this is the only way for a user-defined function to force an exit.

load starts reading statements from a named file. These statements can be assignments to global variables saved with the *save* function, or they can be programs written by the user. The file being loaded may contain further *load* requests. File names on the `face` command line and *load* are both implicit calls to the *ExecFile* internal routine.

save creates a file with assignment statements for all global variables. Special user classifier variables (*messlist* and *rulelist*) are saved as calls to assumed classifier functions (*message* and *rule*). This is a quick way to save the state of a classifier system, so that it can be restored later. Local variables are considered transient and are not saved. Functions are not saved, because an external definition recreated from the internal representation would differ too greatly from what the user originally entered. Users are advised to *load* their functions from a text file.

stop is an implicit call to the *ExecAbort* internal routine. By calling *stop*, user-defined functions can treat a program-detected condition as a fatal execution error. A message explaining the error should be printed before calling *stop*.

10.2. Formatted I/O Functions

Implicit I/O is done when a statement consists of a simple expression: the *PrintValue* internal routine is called to write out the value of the expression. If the expression contains a string, then the string is quoted. A newline is printed after the complete expression. This differs from writing the same expression with *write*, since *write* does not add quotes or newlines. (Perhaps there should be a *writeq* function for explicit quoted output of strings.)

There is no *read* function for two reasons: First, it is unnecessary with *scanf*. Second, in a language without data typing, *read* would have to be told what kind of value to read, which amounts to the same information given to *scanf* anyway. Specialized read functions (*getnumber*, *getstring*, etc.) could remove the second reason, but not the first.

Formatted I/O is the ability to read and write data according to a "picture" of the expected data. Formatted I/O can be done by creating new pre-defined functions, or by calling existing system routines. Writing new functions ensures that the language will retain complete control during I/O. Using existing system routines saves a lot of time by limiting the amount of new code and by using standard documentation. This language has an interface to the UNIX routines *printf*, *scanf*, *sprintf*, and *sscanf*. *printf* does formatted writes onto standard output; *scanf* does formatted reads from standard input (or the current *load* file if there is one). *sprintf* and *sscanf* manipulate strings, in an attempt to move more of the work load into user-defined functions. (If a user can implement a new feature by writing a function, then that is one less function which needs to be implemented in "C".)

The pre-defined functions for formatted I/O do a reasonable amount of error checking before calling the system routines: the format parameter must be a string, other parameters must not be NULL, etc. However, they do not look at the codes in the format string, and do not know what the user is doing. If the user violates the guidelines laid out in the documentation, then the system routine may damage the compiler, the effect of which is unpredictable. A "core dump" is likely.

10.3. General Information Functions

Non-trivial user-defined functions occasionally need more information about an expression other than its nominal value. *abs* returns the absolute value of an expression, saving the user the trouble of checking the sign and possibly negating the value. *random* returns a random integer given a modulus, allowing the user to randomly pick apart messages, rules, or other data objects. *round* rounds a number to the closest integer. *sign* returns the sign of an expression: -1, 0, or +1. *size* returns the number of elements in a list or string. *trunc* truncates a number to its integral value (that is, throws away the fractional part). *type* returns the type of an expression, so that most user classifier support functions can be written in the `face` language (easy to change) instead of as pre-defined pieces of "C" code (more work for the author).

10.4. String Manipulation Functions

Classifier messages and rules are composed from bit strings. These bit strings contain fields of one or more bits, where each field serves some feature in the classifier's application. The programming language prefers to work with lists, where each element in a list corresponds to one "field" in a message. Converting from lists to bit strings would be tedious if nothing more was known in advance about the conversion. Fortunately, the assignment of bits to fields is generally fixed throughout an entire classifier application. Hence, the same field in different messages always has the same size (both in a bit string and as an element in a list). Messages can be created from lists by packing the elements together, and assuming that a user will supply elements appropriate for his application. Messages can be unpacked via a pattern list whose elements are the correct length for the bit fields.

The function for packing a list into a string is called *pack*; the function for unpacking a string into a list is called *unpack*. Both routines are sufficiently general that the user can supply a "value" parameter with NULL values for unspecified fields, along with a standardized "pattern" parameter for missing fields. For example:

```
a := { , "01" , } ;
b := { "##" , "##" , "##" } ;
pack ( a , b ) ;
```

will print:

```
"##01##"
```

since the first and third elements of *a* are NULL, and get replaced from the pattern *b*. Recognizing NULL elements allows the user to manipulate part of a message, then later merge this partial message back into a complete message.

A third function called *pretty* is supplied as a user-defined function in the "pretty.f" file. *pretty* shows how an obscure bit string can be printed in a reasonably intelligent format with names instead of bits. Being user-defined, *pretty* can be copied and changed to fit another application.

The manipulation of strings in this manner is sufficient only when message and rule strings have a fixed format. Should anything more sophisticated be necessary, then SNOBOL pattern matching or UNIX regular expressions may be required.

11. User Classifier Support

Large portions of this language are designed on the principle that the language should know very little about classifier systems. Support for user classifier systems is no different. The less the compiler knows about classifiers, the fewer the assumptions that are made, the easier it is to change the classifier without changing the language. In the ideal scenario, it should be possible to completely replace the classifier without making any changes whatsoever to the language. If this objective can be met, then classifier systems become "users" of the programming language, and it is appropriate to talk about "*user* classifier systems".

To change classifiers without changing the programming language, the classifier should be a separate program. Otherwise:

(1) Every change to the classifier would require the combined object module to be rebuilt.

(2) Both parts would have to be implemented in the same language, or at least in compatible languages.

(3) Similar programming styles would be required to avoid naming conflicts or incorrect function arguments.

(4) Getting 6,000+ lines of compiler working is hard enough without having to worry about side effects on thousands of additional lines.

Hence, classifiers execute as separate UNIX processes and communicate with the programming language through a UNIX pipe.

11.1. Open and Close

The first restriction placed on the ideal situation is that the classifier system must be able to communicate with the programming language. With the UNIX operating system, the best way for two cooperating processes to communicate is with a pipe. (On BSD versions of UNIX, pipes are special cases of sockets, but that is not important here.) A pipe is a buffer stored in kernel memory, giving it a distinct speed advantage over file-based methods. A pipe has a "read" end and a "write" end. One process writes into the "write" end while the other process reads from the "read" end. This establishes a one-way communication path. To form a two-way path, a second pipe is opened in the reverse direction.

Opening pipes involves a lot of system detail which is best omitted here. Suffice to say that one process must act as the "parent"; the other process acts as the "child". The programming language is the parent. To start talking to the classifier "child", the parent opens two pipes. A duplicate copy of the parent is "forked". One copy remains as the programming language, and selects a read end from one pipe and a write end from the other. The second copy uses the opposite ends to replace its standard input and output, and then executes the real classifier program. The classifier starts running with standard input and standard output attached to the pipes from the programming language. The classifier does not know that *stdin* and *stdout* are connected to a process instead of a terminal.

This introduces the first design restriction: user classifier programs must read their input from standard input and write their output on standard output. Other units may be used, but they will not be connected to the programming language.

A second design restriction is implicit here: the user classifier is executed by the name of its executable file. At most one argument string will be supplied. (Both the file name and the argument string are options.)

Once the classifier starts running, it must tell the programming language that it is ready by sending the string:

```
ready
```

following by a newline character. Since the classifier's standard output is connected to a pipe, the following is sufficient:

```
printf("ready\n");
fflush(stdout);
```

The call to *fflush* is necessary to ensure that the *stdout* I/O buffer is forced into the pipe. This introduces a third design restriction: the classifier must flush "ready" onto standard output every time it is ready for new input.

A fourth design restriction applies to the commands sent from the programming language to the classifier. The language will send a command keyword, possibly followed by parameters, followed by a newline character. There is no guarantee that the command keyword will be valid, or that the parameters are meaningful. The classifier must be able to read complete lines from standard input (possibly with *gets*), process the command, print any requested output, and then return back to the "ready" prompt. If the input is illegal, then error messages can be written onto standard output, and will be reported back to the language user (assuming that they are not recognized as legitimate output). A command-driven approach was chosen because it works equally well for input from a process or input from a human user.

The general execution cycle of the classifier must be:

```
initialize
repeat
        write "ready" on standard output
        flush standard output
        read command from standard input
        process command
until command = "close"
```

Sending "close" followed by a newline character is a command for the classifier to finish whatever it is doing and exit.

11.2. Basic Pre-Defined Functions

To support the simple protocol explained above, four pre-defined functions are necessary: *open*, *close*, *send*, and *receive*. *open* takes care of opening the pipe, may be explicitly called by the user with the name of the classifier program, or will be implicitly invoked on the first call to *send* or *receive*. *close* sends a "close" command, does not expect a reply, and closes the pipe. *send* sends an arbitrary string of characters, and adds the trailing newline. *receive* reads a string of characters, and throws away the trailing newline. These four functions make minimal demands upon the style of a user classifier system. While they may dictate the method of communicating, they make no assumptions about what the classifier is doing or how it works.

11.3. Message and Rule Lists

Ideally, no further assumptions should be made. The `face` language is powerful enough that all other communication should be done by user-defined functions loaded for each classifier system. As usual, there is a necessary feature which warrants another assumption.

Classifier systems are based on message and rule lists. The message list corresponds to the current state of the classifier; the rule list corresponds to legal transitions to new states. Both lists are owned by the classifier system, and not by the programming language. Hence, the special language variables *messlist* and *rulelist* are read-only copies of what resides in the classifier system. The message or rule lists could be fetched after every command which changes them by reading back the entire list. This can be done by user-defined functions calling *send* and *receive*. Unfortunately, both lists may be large and change frequently. Hence, trying to keep complete up-to-date copies of both lists at all times would incur a heavy communication penalty.

It is much better to fetch the lists only upon demand: when the user references the *messlist* or *rulelist* variables. Variable references occur below the level of user-defined functions. Hence, support for "fetch on demand" must be encoded into the compiler. Encoding forces assumptions. *messlist* and *rulelist* are assumed to be lists. Each element in *messlist* is a message; each element in *rulelist* is a rule. We could make further assumptions about what messages and rules look like, but this would be unwise. During early conversations with the eventual users of this language, no consensus was reached about the size of a message, the number of conditions in a rule, or even what fields should be in a rule. (Conditions and actions were obvious; strength was reasonably certain; parent and child identifiers were suggested; etc.) If the programming language can not know what a message or rule looks like, then the classifier system must supply this information.

The protocol for fetching a message or rule list is as follows: The string "messlist" (or "rulelist") is sent to the user classifier, followed by the usual newline character. The classifier must respond with one message (or rule) per line, ending with the "ready" prompt. Lines become elements in the *messlist* (or *rulelist*) list. Elements appear in the same order as they are given by the classifier. Each line must be a valid literal data object in the `face` language consisting of NULL values, numbers, lists, or strings. (No expressions are allowed because these lines are not parsed by the regular parser. For the same reason, escape sequences are not allowed in strings, and missing elements are not allowed in lists.) This allows the user classifier system to completely determine the order and content of the message and rule lists. The programming language assumes that the lists exist as lists, but enforces no further assumptions.

Using some examples from Chapter 2, one possible sequence for requesting the rule list is as follows:

language sends	classifier replies
`rulelist`	
	`{ "01010" , "11111" }`
	`{ "0##1#" , "1####" }`
	`{ -"000##" , "00000" }`
	`{ { "0####" , -"####1" } , "0###1" }`
	`ready`

Here, the first element in each list is the condition; the second element is the action. The last rule has a multiple condition. Any spacing of the elements is acceptable, since blanks and tabs are ignored.

11.4. User-Defined Support Functions

Most communication with the classifier system should be initiated by user-defined functions written in the `face` language. These functions should be saved in a file (such as "user.f") and loaded each time `face` is run. The file may contain an explicit call to *open* a connection to the classifier, and may *send* initialization strings.

Consider the "payoff" function defined as follows:

```
function payoff ( number , buffer )
do
     flagrule();
     buffer := "payoff " + pack(number);
     send(buffer);
     receive("ready");
end;
```

which sends a command "payoff" followed by a number to the classifier system. (Pay-off tells the classifier how well it is performing.) The first parameter *number* is assumed to be the pay-off number. The second parameter *buffer* is a local variable that the caller does not know about. The first statement calls the pre-defined function *flagrule* to tell the programming language that its rule list is no longer valid, and must be refetched upon demand. (A similar function *flagmess* exists for *messlist*.) The second statement assigns *buffer* to be the string "payoff" followed by a space followed by *number* packed into a string. This *buffer* is sent to the user classifier with *send*. The response "ready" is expected by *receive*. Any output before the "ready" prompt will be treated as an error message.

User-defined functions may be slower than pre-defined functions inside the compiler, but they remove from the programming language almost all decisions about the classifier system. This comes close to the ideal of having a language which knows very little about classifiers.

12. Final Comments

During the design of this classifier programming language, very little has been said about user classifier systems. An early chapter talks about how the representation of classifier data affects the rest of the language. A later chapter begins to discuss pre-defined functions for a communication interface, but stops after concluding that most of the work can be done by user-defined functions. In between, classifier systems are virtually ignored.

The programming language is not the classifier system. There is no need to understand the classifier, only to feed it the correct commands and to read back the results. Given a minimal set of data requirements, and knowing that the language will be active during all communication with the classifier, designing the language becomes a matter of finding the smallest, most complete grammar which satisfies the requirements. The chosen language looks like the pseudo-code often used to describe the execution of algorithms. This is no coincidence. Pseudo-code is meant to be intuitive, once a few basic operators are explained. This language has a limited number of operators and pre-defined functions, all of which are implemented on the basis of what should work does work.

The user may look at this language as either the biggest desk calculator ever written (next to APL), or as a programmable tool. The connection to a user classifier system is fully functional, but is not a necessary part of the language. All concept of classifiers could be removed, and some new application added, by rewriting a few pre-defined functions. New features are limited only by the definition of functions: their parameters and results must be valid data objects in the language. This restriction is not too severe, since the representation of data has been generalized beyond the point where explicit type-checking can be performed. (That is, data has become "polymorphic".)

A few final comments are in order.

12.1. Representation of Data

Too much time is spent manipulating dynamic data (see Appendix C). While there is no obvious way of reducing this in the current version of the compiler, it is also not so obvious that the chosen data structures are the best possible. A close contender in the original design was to overwrite existing structures when doing an assignment (Chapter 6). This may have been faster. Unfortunately, a great deal of work would be involved in changing over to a different design, and curiosity alone was not enough motivation to investigate this alternative.

12.2. Questionable Semantics

The semantics of having assignment as an operator when combined with dynamically allocated data (that is, no type checking) are questionable. For example, what should the following statements mean?

```
s := {1, 2, 3} ;
s [ s := 1 ] := s ;
```

Is it legitimate to index a list with an expression that changes the meaning of the list? What happens when the inner assignment releases the old value of *s* but the outer assignment attaches a new value to a released element? There should be an example like this that blows up the compiler; so far none has been found. Some dynamic memory becomes attached to dead storage (freed memory), which is a fault, but does not violate the integrity of the compiler.

12.3. Missing Features

This classifier language does many things well. The features that have been included are carefully explained in a user's guide (Appendix A) and an internal guide (the thesis body). Features that were not included are explained only briefly, or not at all.

Control flow is limited to *for*, *if*, *repeat*, and *while* statements. Even though most structured code can be phrased in terms of these statements, it is sometimes more convenient to have other variations. For example, *break* statements in the "C" language may disturb the recursive execution of nested code, but they are also very useful. (As proof, see how often they are used in the compiler's code!)

I/O support is primitive. The user must read and write with units chosen by the language. The ability to open arbitrary files is missing. To save more than just the global variables, the UNIX *script* command must be used to record a terminal session, this file must be edited back into a suitable format, and then resubmitted to the compiler as input.

Bibliography

[Ah86] Aho, Alfred V.; Ravi Sethi; and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing, Reading, Massachusetts, 1986.

[Bo81] Boswell, F. D.; T. R. Grove; and J. W. Welch. *Pascal Reference Manual and Waterloo Pascal User's Guide*. WATFAC Publications Limited, Waterloo, Ontario, 1981.

[Fe78] Feldman, S. I. "Make — A Program for Maintaining Computer Programs", distributed in *The UNIX Programmer's Manual*. AT&T Bell Laboratories, Murray Hill, New Jersey, 1978.

[Fe68] Feller, William. *An Introduction to Probability Theory and Its Applications*, Volume I (third edition). John Wiley & Sons, New York, New York, 1968.

[Gi76] Gilman, Leonard and Allen J. Rose. *APL: An Interactive Approach* (second edition, revised reprinting). John Wiley & Sons, New York, New York, 1976.

[Ho86] Holland, John H. "Escaping Brittleness: The Possibilities of General-Purpose Learning Algorithms Applied to Parallel Rule-Based Systems" in *Machine Learning: An Artificial Intelligence Approach*, Volume II, pages 593-623. Edited by Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell. Morgan Kaufmann Publishers, Los Altos, California, 1986.

[Ho86-2] Holland, John H.; Keith J. Holyoak; Richard E. Nisbett; and Paul R. Thagard. *Induction: Processes of Inference, Learning, and Discovery*. The MIT Press, Cambridge, Massachusetts, 1986.

[Jo78] Johnson, Stephen C. "Yacc: Yet Another Compiler-Compiler", distributed in *The UNIX Programmer's Manual*. AT&T Bell Laboratories, Murray Hill, New Jersey, 1978.

[Ke78] Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Ke84] Kernighan, Brian W. and Rob Pike. *The UNIX Programming Environment*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[Le78] Lesk, M. E. and E. Schmidt. "Lex — A Lexical Analyzer Generator", distributed in *The UNIX Programmer's Manual*. AT&T Bell Laboratories, Murray Hill, New Jersey, undated (1978?).

[Ri86] Riolo, Rick L. "CFS-C: A Package of Domain Independent Subroutines for Implementing Classifier Systems in Arbitrary, User-Defined Environments". Technical Report, Logic of Computers Group, Division of Computer Science and Engineering, University of Michigan, Ann Arbor, Michigan, January 1986.

[Wi84] Wilensky, Robert. *LISPcraft*. W. W. Norton & Company, New York, New York, 1984.

Appendix A: Language Description

`face` is a classifier programming environment in two parts: an interactive programming language and a user classifier system. The programming language supports simple data objects (numbers, strings, lists), global variables, basic control statements ("for", "if", "repeat", "while"), pre-defined functions, and user-defined functions with local variables. The user classifier system is any program that can communicate with the programming language through special functions written in an application-dependent module.

A.1. Programming Language

The `face` programming language consists of comments, data objects, variables, expressions, statements, and functions.

A.1.1. Comments

A comment starts with a "#" symbol and goes to the end of a line. Comments are treated as white space (blanks or tabs) and ignored. Comments may not occur inside a reserved word or other token. (Please note that "#" is just a normal character when it appears in a string.)

A.1.2. Data Objects

There are three types of simple data objects: real numbers, character strings, and the NULL value.

Numbers consist of one or more signed decimal digits, possibly containing a decimal point, optionally followed by an exponent. The following are examples of legal numbers:

```
0
42
-1987
+3.14159
6.022045e23
```

Spaces and punctuation marks (",") are not allowed in a number.

Strings consist of zero or more characters between quotation marks. Quotation marks are the regular quote character ("), the closing quote or acute character (´), and the opening quote or grave character (`). A string may include any character except for the newline and end-of-file characters. Inside a string, you may use the following escape sequences for special characters:

`\n`	newline character
`\t`	tab character
`\\`	\ character

and \ followed by the quoting character gives you one quotation mark inside the string. The following are examples of legal strings:

`""`	empty string
`´ h e l l o ´`	"hello" with spaces
`"\n"`	newline
`"(\")"`	(")

Strings may have a sign. By default, strings are "positive". You may negate a string by putting a minus sign in front of the string:

```
- "pattern"
```

Positive and negative strings can not be combined with any of the operators described later. (Negative strings are used to specify negative conditions in classifier rules.)

The last simple data object is the NULL value. NULL represents the absence of a value. NULL is not the same as an empty string. (An empty string is a *string*, while NULL is nothing.) NULL values occur when you use a variable that has not been assigned a value.

Simple data objects can be combined into lists. A list consists of zero or more elements. Each element may be a number, a string, the NULL value, or another list. The smallest list is the empty list:

```
{ }
```

The empty list has no elements, and hence has a size of zero. Lists of size one are legal:

`{ 3 }`	first element is the number 3
`{ "word" }`	first element is the string "word"

(The spaces shown above are not necessary.) Lists with more than one element have the elements separated by commas (","):

`{ 1 , 2 , 3 }`	list of three numbers
`{ 1, "two", NULL }`	elements can be different types
`{1, {2, 3, 4}, 5, 6}`	list within a list

This last list is of size four (not six), because the second element is a list of size three.

Lists in this language replace arrays and records found in other languages. Like strings, lists can be signed.

A.1.3. Variables

A variable is a named object. Names consist of a letter followed by zero or more letters or digits. The following are examples of legal names:

```
k
Prog67
index
```

Upper and lower case letters are different in names: "Prog67" is *not* the same variable as "prog67". Some of the possible names are reserved words, and may not be used as variables:

```
and break by div do elif else end eq eqp for from
function ge gt if le lt mod ne nep not null or
procedure repeat return to then until while
```

These reserved words are recognized in any combination of upper and lower case (unlike variable names). A few other names are assigned to pre-defined functions and variables; you should avoid using these names, but the language won't prevent you from assigning your own value. There are two pre-defined variables:

```
false := 0 ;
true := 1 ;
```

"if", "repeat", and "while" statements require conditional expressions that evaluate to either *false* (0) or *true* (1); relational operators return 0 or 1 as their result.

Variables can be assigned values by naming them on the left side of an assignment operator (":="). To give the variable *a* the value 27, use:

```
a := 27 ;
```

The semicolon is a required part of the syntax and marks the end of an executable statement. To look at the value of a variable, type the name followed by a semicolon (followed by a newline, of course):

```
a ;    prints   27
```

If you use a variable that hasn't been given a value, then the value will be NULL.

These are examples of global variables. Later, local variables will be introduced when functions are defined.

A.1.4. Expressions

An expression takes one or more data objects and returns a new object. Standard operators are used.

A.1.4.1. Number Operators

For numbers, the following operators are supported:

operator	type	explanation
+	binary	real addition
+	unary	(no effect)
-	binary	real subtraction
-	unary	negate sign of number
*	binary	real multiplication
/	binary	real division
** ^	binary	real exponentiation
div	binary	integer quotient
mod %	binary	integer remainder
and &	binary	logical AND
not ¬	unary	logical negation
or |	binary	logical OR
eq = ==	binary	equal relation
ge >= =>	binary	greater than or equal relation
gt >	binary	greater than relation
le <= =<	binary	less than or equal relation
lt <	binary	less than relation
ne != <>	binary	not equal relation

The standard arithmetic operators ("+", "-", "*", "/") are given the usual definitions:

`22 + -3;`	prints	`19`
`11 - 4;`	prints	`7`
`2 * 3;`	prints	`6`
`7 / 2;`	prints	`3.5`
`7 div 2;`	prints	`3`
`7 mod 2;`	prints	`1`

since seven divided by two has an integer quotient of three and a remainder of one.

The logical operators are functions of 0 and 1:

`not 0;`	prints	`1`
`0 and 1;`	prints	`0`
`0 or 1;`	prints	`1`
`1 and 1;`	prints	`1`

Using a value other than 0 or 1 will force an execution error. (The same applies to all operators and functions when given an illegal value.)

Relational operators compare two numbers, and return a 1 if the specified condition is true, and 0 otherwise:

`3 eq 4;`	prints	`0`
`3 < 4;`	prints	`1`
`3 != 4;`	prints	`1`

A.1.4.2. String Operators

For strings, the following operators are supported:

operator	type	explanation
+	binary	string concatenation
+	unary	(no effect)
-	unary	negate sign of string
and &	binary	bit string AND
not ¬	unary	bit string negation
or		binary	bit string OR
eq = ==	binary	equal relation
eqp	binary	equal pattern relation
ge >= =>	binary	greater than or equal relation
gt >	binary	greater than relation
le <= =<	binary	less than or equal relation
lt <	binary	less than relation
ne != <>	binary	not equal relation
nep	binary	not equal pattern relation
[]	binary	subscription (indexing)

Adding two strings with "+" gives you a string with the two parts concatenated:

`"hello" + " there";`   prints   `"hello there"`

Bit string operations assume that a string represents all possible binary values with a certain pattern. The pattern is specified with "0" for zero bits, "1" for one bits, "#" for don't cares ("0" or "1"), and "?" for positions where no bit is legal. Thus, the bit string pattern "0#1" represents all binary values with a zero followed by any digit followed by a one. Namely, 001 and 011. The bit string AND operation combines two strings (with the same length and sign) into a new string which specifies all binary values that match the first string *and* the second string:

AND	0	1	#	?
0	0	?	0	?
1	?	1	1	?
#	0	1	#	?
?	?	?	?	?

The expression

```
"0" and "1";    prints    "?"
```

because no binary value can have both "0" and "1" in the same bit position. Bit string AND is a well-defined operation: the binary values which match the resulting string are exactly those that match both input strings.

Bit string OR is not as well-defined:

OR	0	1	#	?
0	0	#	#	0
1	#	1	#	1
#	#	#	#	#
?	0	1	#	?

The resulting string can match more binary values than are strictly matched by either of the input strings. For example, "0#" matches 00 and 01, "11" matches only 11, but (by definition)

```
"0#" or "11";    prints    "##"
```

This matches 00, 01, 11, and the unwanted value 10. Bit string OR is normally only used when creating a pattern condition in a classifier rule.

You may find it easier to understand bit strings if you note that the "0" character means bit 0 is legal and bit 1 is not legal; the "1" character means that 0 is illegal and 1 is legal; "#" means that both 0 and 1 are legal; and "?" means that both are illegal:

character	is 0 legal?	is 1 legal?
"0"	yes	no
"1"	no	yes
"#"	yes	yes
"?"	no	no

The bit string operations are Boolean functions of these underlying "yes" and "no" conditions. The expression

```
"0" and "1"
```

is equivalent to logically AND'ing the first row (yes/no) with the second row (no/yes) to get the last row (no/no).

Bit string negation replaces each "0" with a "1", each "1" with a "0", each "#" with "?", and each "?" with "#":

```
not "0";        prints    "1"
not -"0#1";     prints   -"1?0"
```

Strings are compared according to the ASCII sequence. When two strings have different lengths, and one string is equal to the beginning of the other string, then the shorter string is less than the longer string:

```
"A" < "a";         prints   1
"k" < "ke";        prints   1
"" ne "hello";     prints   1
```

The "eqp" and "nep" operators do equality comparisons with pattern matching. This is not much different than normal comparisons, except that the don't care character ("#") is equal to all characters:

```
"00" eqp "0#";    prints  1
"11" eqp "0#";    prints  0
```

Pattern comparisons are useful when looking at messages generated by a classifier system.

Strings may be subscripted by choosing either an individual character, or a subrange of characters, as in the following examples:

```
"hello"[2];      prints   "e"
"hello"[2:3];    prints   "el"
```

The first character in a string has index 1. It is a mistake to subscript a string with a starting index less than 1 or greater than the size of the string. In a subrange, the final index may be any non-negative integer. If the subrange final index is less than the starting index, then an empty string is returned. If the final index is greater than the string length, then the rest of the string is returned (no padding). Note that since subscripting a string returns a string, the result can be subscripted again! The following all print the string "lo":

```
"hello"[2:5][3:4];
"hello"[1:9, 4:5];
"hello"[4] + "hello"[5];
```

When indexing a single character from a string ("[n]" form), the sign of the string is ignored. When indexing a subrange ("[m:n]" form), the sign of the string is copied to the result. (This keeps indexed strings consistent with indexed lists.) Subscripted strings are never assignable.

A.1.4.3. List Operators

For lists, the following operators are supported:

operator	type	explanation
+	binary	list concatenation
+	unary	(no effect)
-	unary	negate sign of list
eq = ==	binary	equal relation
eqp	binary	equal pattern relation
ge >= =>	binary	greater than or equal relation
gt >	binary	greater than relation
le <= =<	binary	less than or equal relation
lt <	binary	less than relation
ne != <>	binary	not equal relation
nep	binary	not equal pattern relation
[]	binary	subscription (indexing)

You may concatenate two lists with the binary "+" operator:

`{1, {2, 3}} + {4, 5};` prints `{1, {2, 3}, 4, 5}`

Comparisons between lists are recursive: both lists should have similar structures, and the comparison terminates prematurely when a difference is found.

Lists may be subscripted in the same way as strings. Subscripting a single element in a named list (global variable) is assignable. For example, if the global variable *s* has the value:

```
s := {7, 5, 6};
```

```
s[1];    prints    7
s[2:3];  prints    {5, 6}
s[2:2];  prints    {5}
s[2:1];  prints    {}
```

Individual elements can be reassigned. The expression

```
s[2] := "new";
```

changes the second element of *s* to be the string "new". The new value of *s* is:

```
{7, "new", 6}
```

If the result of list subscription is a list or string, then you can subscript the result.

A.1.4.4. Combining Expressions

When expressions are combined, operators are given the following priorities:

priority	association	operators
highest	right	subscription ([])
	right	exponentiation (**)
	right	not, unary +, unary -
	left	*, /, div, mod
	left	+, -
	left	eq, eqp, le, lt, ge, gt, ne, nep
	left	and
	left	or
lowest	right	assignment (:=)

That is, subscription is performed first, then exponentiation, then unary operators, then multiplication and division, then addition and subtraction, etc. If you want an expression to be evaluated in a different order, then you must use parentheses:

`6 + 4 / 2;` prints 8

`(6 + 4) / 2;` prints 5

A.1.5. Statements

The simplest statement is an expression: the expression is evaluated and the result is printed. Expressions include function calls (described later). Other statements are the "for", "if", "repeat", and "while" control statements. Every statement must end with a semicolon (;). Empty statements are legal, so extra semicolons are safely ignored. Complete programs are created by nesting control statements.

When a statement returns a value (expression or function call), then the value is printed — unless the statement is an assignment, or a function which returns the NULL value.

A.1.5.1. FOR Statement

A *for* statement consists of an index variable, an initial value, an increment, a final value, and some statements. All parts are optional, except for the variable name. The syntax is:

```
for name
from expression
to expression
by expression
do statements
end;
```

name must be the name of a variable (global or local). All three expressions ("from", "to", and "by") must evaluate to numbers. If the *from* clause is missing, an initial value of 1 is assumed. If the *to* clause is missing, a final value of 1 is assumed. If the *by* clause is missing, an increment of 1 is assumed. The *by* expression (if given) must be non-zero.

The *for* statement assigns the initial value to the index variable. If the index variable is less than or equal to the final value (when the increment is positive), or greater than or equal to the final value (when the increment is negative), then the *statements* are executed, the increment is added to the index variable, and this check is performed again.

The initial, final, and increment values are computed at the beginning of the *for* loop: changing them inside the loop will have no effect. Similarly, changing the index variable inside the loop does not affect the *for* loop. When the loop terminates, the value assigned to the index variable should not be used. (If you want to manipulate the index variable yourself, then use a *repeat* or *while* statement.)

The following example computes the sum of the first nine numbers:

```
sum := 0;
for i from 1 to 9 do
    sum := sum + i;
end;
```

A.1.5.2. IF Statement

An *if* statement has a conditional expression, an optional "then" clause, and an optional "else" clause. The conditional expression must evaluate to either *true* (1) or *false* (0). If *true*, the statements in the *then* clause are executed; otherwise, the statements in the *else* clause are executed. The syntax is:

```
if expression
then statements
else statements
end;
```

Examples:

```
if (a < 0)      # assume "a" is a number
then a := 0;    # enforce minimum value
end;
```

```
if (sign(a) < 0)    # "a" is any data object
then write("a is negative\n");
else
    if (sign(a) > 0)
    then write("a is positive\n");
    else write("a is zero or NULL\n");
    end;
end;
```

When an *if* statement has multiple disjoint conditions, it is better to use an "else-if" form. The previous "sign" example can be rewritten as:

```
if (sign(a) < 0)    # "a" is any data object
then write("a is negative\n");
elif (sign(a) > 0)
then write("a is positive\n");
else write("a is zero or NULL\n");
end;
```

In this language, "else-if" statements replace the "switch" statements found in "C" and the "case" statements found in Pascal.

A.1.5.3. REPEAT and WHILE Statements

repeat and *while* statements have a conditional expression and some statements inside a loop. The conditional expression must evaluate to either *true* (1) or *false* (0). For the *repeat* statement, the statements in the loop are executed and the expression is checked. If the expression is *false*, then the statements are executed again until the expression evaluates to *true*. (The statements in a *repeat* loop are always executed at least once.) For the *while* statement, the expression is checked. If the expression is *true*, then the statements in the loop are executed until the expression evaluates to *false*.

The syntax of a *repeat* loop is:

```
repeat statements
until expression ;
```

The syntax of a *while* loop is:

```
while expression
do statements
end;
```

Given a number *n*, the following example computes the next higher power of two:

```
power := 1;
while (power <= n)
do
    power := power * 2;
end;
```

A.1.6. Functions

A function is a small program which is given a name so that it can be used later. Functions have zero or more parameters, can execute other statements, and may return a result. The syntax for defining a function is:

```
function name ( parameters )
do statements
end;
```

name is the name of the function (such as the infamous "f"). *parameters* is either empty or a list of local variable names. If *parameters* is empty, then the parentheses "(" and ")" may be omitted. *statements* are the executable statements that make up the function. (An empty function is legal.)

All variables named in the parameter list are local to the function. When a function is called, the caller gives you values for some or all of the local variables in the parameter list. Any parameters omitted by the caller are automatically assigned NULL values. Hence, you can safely put more variables in the parameter list than you expect the caller to provide; the extra variables become temporary variables that disappear when the function returns to the caller.

The only way for a function to return information to the caller is by changing global variables, or by assigning a result to the function's name. (If the function is not assigned a result, then the NULL value is returned.)

For example, the following is a verbose version of the recursive factorial function:

```
function f(n, temp)
do
    if (n <= 1)
    then temp := 1;
    else
        temp := n;
        temp := temp * f(n-1);
    end;
    f := temp;
end;
```

The caller is expected to supply a small number as the first parameter; the second parameter is just a temporary local variable. Of course, this function can be written more compactly as:

```
function f(n)
do
    if (n <= 1)
    then f := 1;
    else f := n * f(n-1);
    end;
end;
```

To call this function, you would type:

`f(0);`	prints	1
`f(1);`	prints	1
`f(2);`	prints	2
`f(3);`	prints	6
`f(4);`	prints	24

Since the factorial is returned as the function's result, it can be assigned to a variable, or used in another expression (as is done in the function itself when it recurses).

A.2. Pre-Defined Functions

Pre-defined functions provide services that are too difficult or too awkward to do with simple operators. One service is communication with a user classifier system (described in the next section). Other services are control of the environment, formatted I/O, general information, mathematical functions, and string manipulation. Most functions are invoked as statements, which means that you must include the semicolon that terminates every statement.

A.2.1. Control Functions

Control functions start and stop `face`, or change the source of its input:

`exit ( )`

exits from `face` and returns you to UNIX. This is equivalent to typing an end-of-file character (control-D) on standard input. A synonym for *exit* is *quit*.

`load (` *file* `)`

starts reading input from a file named by the string *file*. The commands in this file are read and executed until either the end of the file is reached, or an error occurs. Then input goes back to the previous file (or standard input). If the file contains a formatted input *scanf* statement, which is also executed, then the file must contain the appropriate data. (This does not apply if *scanf* appears in a function definition which is executed later.)

`save (` *file* `)`

saves the value of all global variables in a file named by the string *file*. The special classifier variables *messlist* and *rulelist* are saved as calls to the functions *message* and *rule*. If *file* is omitted, or NULL, then the information is printed on standard output (your terminal). After an error, this is a good way to dump everything to see what happened. The file created by this function can be loaded with *load*.

`stop ( )`

stops the execution of the current statement or function. If input is coming from a file, then the file is closed (just like a fatal error). If input is coming from standard input (the terminal), then the next statement is read.

`system (` *command* `)`

calls the UNIX "sh" shell with a *command* string. No status is returned: if the command fails, the shell will print an error message on your terminal.

A.2.2. Formatted I/O Functions

Formatted I/O comes in two flavors: easy and hard. The easy I/O function is *write*:

`write (` *p1*, *p2*, `... )`

writes all of its parameters on standard output without adding spaces, newlines, or quotes to strings. You may give any number of parameters. *write* always works, because nothing can go wrong. The last parameter to *write* is usually a newline string. Example:

```
write("the value of a is ", a, "\n");
```

(There is no *read* function — use *scanf*.)

The other four formatted I/O functions are not so easy: if you make a mistake, then you can crash the `face` program. The reason for this is very simple: `face` does not perform the I/O itself. The parameters you specify are given to a system subroutine by the same name. If the parameters are bad, or even slightly wrong, there is a good chance that `face` will lose control. Should that happen, you will get a nasty "core dump" message. The author of `face` will refuse to look at any "bug" that occurs during formatted I/O.

These formatted I/O routines need information about the parameters that are being passed to the system subroutines. Because data in this language is not strongly typed (as in Pascal), the type of an I/O parameter must be guessed by looking at the current value. That is, if you want to read a number, then the variable you give to receive the number must be currently assigned to a number. Similarly, to read a string, the receiving variable must be assigned to a string. (The current values are thrown away, but the type information is still necessary.) During output, a similar restriction applies. If your format string specifies a %g format code, then the corresponding parameter must be a number. If the format code is %s, then the parameter must be a string. (Other codes may be used at your own risk.)

`printf` (*format*, *p1*, *p2*, ..., *p9*)

prints a formatted string with up to nine parameters on standard output. The value of the parameters must correspond to the format codes in the string *format*. Numbers should be written with the %e, %f, or %g format codes; strings should be written with %s. Examples:

```
number := 42;
printf("number is %g\n", number);
string := "hello";
printf("string is '%s'\n", string);
```

`scanf` (*format*, *p1*, *p2*, ..., *p9*)

reads up to nine parameters according to a format string. Input is read from the current *load* file, or from standard input if there is no *load* file. The current value of the parameters must correspond to the format codes in the string *format*. Numbers should be read with the %le or %lf format codes; strings should be read with %c or %s. Example:

```
number := 0;      # any number is okay
string := "";     # any string will do
scanf("%lf %s", number, string);
```

Like the real "scanf" subroutine, *scanf* returns the number of items correctly read. You should probably assign this count to a dummy variable and ignore it.

`sprintf` (*string*, *format*, *p1*, *p2*)

is like *printf*, but the output goes into *string* (which must be a variable that already has a string value). For obscure reasons, *sprintf* is limited to two data parameters.

`sscanf` (*string*, *format*, *p1*, ..., *p9*)

is like *scanf*, but the input comes from *string*. Example:

```
thing := 0;    # use any number here
count := sscanf("1234xyz", "%lf", thing);
```

would assign the value 1234 to the variable *thing*. *count* will be 1.

A.2.3. General Information Functions

General functions provide information that could be done by operators, but which are traditionally done by simple functions:

`abs ( `*expr*` )`

returns the absolute value of an expression. The absolute value of NULL is NULL. The absolute value of a number is the positive part. For lists and strings, the absolute value has a positive sign.

`random ( `*limit*` )`

returns a random integer greater than or equal to zero and less than the positive integer *limit*.

`round ( `*expr*`, `*scale*` )`

rounds the number *expr* to the closest integer. If the second parameter is given, then it must be a non-zero *scale* factor: 0.01 rounds to two digits after the decimal point, 100 rounds to the nearest multiple of one hundred, etc. The default *scale* factor is 1.

`sign ( `*expr*` )`

returns the sign of an expression. The NULL value has sign NULL. Numbers have a sign of -1, 0, or +1. Lists and strings have a sign of -1 or +1.

`size ( `*expr*` )`

returns the size of an expression. The NULL value has size 0. Numbers have size 1. The size of a string is the number of characters in the string. The size of a list is the number of elements. *size* is usually called in a *for* loop when you want to loop through all elements in a list.

`trunc` (*expr*, *scale*)

truncates the number *expr* to its integer part by throwing away anything after the decimal point. If the second parameter is given, then it must be a non-zero *scale* factor: 0.01 truncates to two digits after the decimal point, 100 truncates any part less than one hundred, etc. The default *scale* factor is 1.

`type` (*expr*, *string*)

returns the type of an expression as a string: "null", "number", "set" (for lists), or "string". (If something goes wrong, you may also see "dummy" or "element".) You may specify a second parameter, in which case the type is compared against your *string*; a 1 is returned if they are equal, and a 0 is returned otherwise.

A.2.4. Mathematical Functions

The math functions take one or two numbers as parameters and return one number as a result. The names of these functions and their parameters are identical to the standard UNIX math library routines:

`acos` (*x*)	arc cosine (inverse)
`asin` (*x*)	arc sine (inverse)
`atan` (*x*)	arc tangent (inverse)
`atan2` (*y*, *x*)	arc tangent of *y* over *x*
`cbrt` (*x*)	cube root
`cos` (*x*)	cosine (radians)
`exp` (*x*)	natural exponential e^x
`log` (*x*)	natural logarithm (base *e*)
`log10` (*x*)	logarithm base ten
`pow` (*x*, *y*)	exponential x^y
`sin` (*x*)	sine (radians)
`sqrt` (*x*)	square root
`tan` (*x*)	tangent (radians)

A.2.5. String Manipulation Functions

Classifier systems work with bit strings. Bit strings are difficult for people to understand. It is much easier to create lists using symbolic names for values. For example, if your classifier machine is playing a game of tic-tac-toe, then each square can be empty, "X", or "O". Three possible values require two encoding bits. One scheme is:

```
empty := "00";
X := "10";
O := "11";
```

Then the following list is sufficient to specify nine squares:

```
board := {
    empty, empty, X,
    empty, O, X,
    empty, empty, O
    };
```

(which is a win for "O" if "O" moves next). Once the names are evaluated, *board* will be assigned the list:

```
board := {"00","00","10","00","11","10","00","00","11"};
```

This is no longer easy for a person to read, but still isn't readable enough for the classifier system. The classifier expects bit strings (not lists). To convert *board* into a bit string, the elements need to be packed together:

```
"000010001110000011"
```

Conveniently, there are functions called *pack* and *unpack* to do this:

`pack` (*value*, *pattern*)

The first parameter *value* is converted into a packed character string. If the second parameter *pattern* is missing or NULL, then the packed string looks like *value*, except that there are no commas, spaces, list delimiters, or NULL values:

`pack(3);`	prints	`"3"`
`pack({1, 2});`	prints	`"12"`
`pack({"hello",{-37,NULL}});`	prints	`"hello-37"`

If *pattern* is given, then it should have a list structure similar to *value* (but possibly simpler). For each NULL element in *value*, the corresponding element in *pattern* is converted into a string:

`pack({"abc",NULL,"ghi"},"def");`	prints	`"abcdefghi"`

(This example uses the same pattern string for all *value* elements.) Non-NULL elements are converted into strings that look like the *pattern* element. The sign of the *value* element is preserved only if the *pattern* element has a negative sign. Examples:

`pack(5, 1);`	prints	`"5"`
`pack(-5, 1);`	prints	`"5"`
`pack(-5, -1);`	prints	`"-5"`

`pack("hello", -"thing");`	prints	`"hello"`
`pack("hello", -"thi");`	prints	`"hel"`
`pack("hel", -"thing");`	prints	`"helng"`
`pack(-"he", -"thing");`	prints	`"-heing"`

Value strings packed according to a pattern string have the same length as the pattern string: if the value string is shorter than the pattern string, then the remainder is taken from the pattern; if the value is longer, then it is truncated.

pack is recursively defined for lists:

```
pack(-{-1,-2,-3},-{-5,5,-5});     prints    "--12-3"
```

`unpack ( value, pattern )`

unpacks *value* according to *pattern*. Each string in *value* is replaced by something that looks like *pattern*. The elements in *pattern* must have a negative sign if signs are expected in the *value* string. Examples:

```
unpack("abcd", {"xx", "xx"});    prints   {"ab", "cd"}
 unpack("-123x", {-5, "z"});     prints   {-123, "x"}
```

unpack is the reverse of *pack* for individual strings. For lists, *unpack* is recursively defined to apply the same *pattern* to each string element; other elements are not changed:

```
unpack({1,"abc",NULL},{"xx","y"});   prints   {1,{"ab","c"},NULL
```

pack and *unpack* can quickly compose and decompose classifier bit strings. Their recursive definition makes them difficult to understand, but they can usually be convinced to do what you want them to do.

Another string manipulation function called *pretty* is in the file "pretty.f" in the same directory as `face`, and should be copied when you copy `face`:

`pretty ( value, picture )`

prints a *value* in a pretty format. The first parameter may be a number, a string, a list, or the NULL value. The second parameter should have the same list structure as the first parameter, but with one extra level of lists to provide a mapping from *value* elements to name strings. For example, suppose there is a field in a classifier message that has the string "00" for NO, "11" for YES, and "01" for MAYBE. Then a mapping picture would be:

```
picture := {"00","NO","11","YES","01","MAYBE"};
```

pretty first tries to find an identical picture element (no pattern matching). The function call:

```
pretty("00", picture);
```

would print "NO" (without quotes or newlines). If an identical element can not be found, then *pretty* tries again with pattern matching. The function call:

```
pretty("#1", picture);
```

would print "(YES or MAYBE)". Finally, if pattern matching fails, then the *value* is printed in square brackets. The function call:

```
pretty("10", picture);
```

would print "[10]" since there is no legal mapping.

(This function is recursively defined for lists. The caller is responsible for writing any newlines before or after the "pretty" output. Bad parameters will result in obscure error messages. As this is a user-defined function, you may inspect the source code and change it to suit your application.)

A.3. User Classifier System

The classifier system is intended to be an artificial intelligence application using genetic bit strings for learning. None of this matters to the programming language. As long as the classifier system satisfies the following requirements, it can be interfaced with `face`:

(1) The user classifier system must be a program which can be executed via the *execl* system call. One argument string will be given on the "command" line. The classifier should read from standard input ("stdin") and write on standard output ("stdout"). Other logical I/O units may be used, but only standard input and output will be connected to `face`.

(2) The classifier program must be command driven. `face` will send a command to the classifier's standard input. The classifier must perform any necessary action, possibly replying on standard output, and then wait for the next command. There should be no unsolicited input or output, since `face` is unable to handle this, and will report anything it doesn't understand as an error message.

(3) Support for the classifier program must be written in "C" in the "fause.c" module. This module may define functions which are visible to the user, may have variables which are treated in special ways, and may perform all actions that are normally allowed in "C" programs. The function parameters and results must be standard `face` data types.

(4) For most classifiers, only six "C" functions are required: *close*, *flagmess*, *flagrule*, *open*, *receive*, and *send*. No changes to these routines should be necessary if the following guidelines are observed:

  (a) Before reading a command, print "ready" (without the quotes) followed by a newline character on standard output, call *fflush* for *stdout*, and then read a complete line from *stdin* into a buffer with *gets*. Process commands from this buffer, so that extraneous input can be ignored without leaving unread characters on *stdin*.

(b) The *open* and *close* protocols are simple enough, and should not be changed.

(c) Special variables such as *messlist* and *rulelist* require numerous hooks into the language. These hooks are necessary because the lists are big, change frequently, and must only be refetched on demand (that is, when referenced as a variable — which occurs below the level of user-defined functions). If you have data in your classifier which you want the user to see, think carefully before deciding to create a new special variable. It is much easier to write a user-defined function which manipulates global variables and talks to the classifier system through customized *send* and *receive* strings.

(d) The protocol for fetching the message and rule lists is as follows: The string "messlist" or "rulelist" is sent to the classifier (followed by the usual newline). The classifier is expected to return one message (or rule) per line. Each line should be a properly formatted number, string, list, or NULL value. Each line will become one element in the list. This allows the classifier to completely determine the structure of the message and rule lists. The user-defined *message* and *rule* functions should have similar definitions.

Ignoring these guidelines will create unnecessary work.

A.3.1. Robert Chai's Classifier System

Robert Chai's classifier is a bit-string learning system currently called `robert`.

A.3.1.1. Global Variables

There are two special global variables called *messlist* and *rulelist*. Both variables are lists. *messlist* is a list of message strings; each element is a bit string consisting of the characters "0" and "1". *rulelist* is a list of rules, where each rule has the list structure:

{ *conditions* , *action* , *strength* }

conditions is a list of message pattern strings consisting of the characters "0", "1", and "#" (don't care). Pattern strings may have positive signs (where a message matches if it has the same pattern), or negative signs (where a message matches if it does *not* have the pattern). All condition strings must match before the *action* pattern string is invoked. *strength* is the rule's strength as a number starting from zero.

You may use *messlist* and *rulelist* as normal variables: you can subscript them with "[]", find out how big they are with the *size* function, etc. However, you can not change them — both variables are read-only. The only way to create new messages is with the *message* function (described later), and the only way to add new rules is with the *rule* function. This restriction is imposed because these variables are owned by the classifier system, not the programming language. `face` tries to make this transparent to the user by knowing which functions change *messlist* and *rulelist*, and fetching new copies from the classifier system when necessary.

A.3.1.2. Pre-Defined Functions

The following functions have been implemented. Many of these functions are written as user-defined functions and should be loaded from the "user.f" file when you run `face`.

`clear ( )`

clears all data in the user classifier system, effectively performing an initialization.

`close ( )`

closes the connection ("pipe") to the classifier system. The string "close" is sent to the classifier followed by a newline character. No response is expected. (*close* is implicitly done before `face` exits.)

`crossover ( )`

picks two rules at random and creates a new rule by swapping some of the bits.

`flagmess ( )`

flags the message list as invalid, so that a new copy will be fetched upon the next reference to *messlist*. This is automatically done when a pre-defined function sends a command to the classifier system which may affect the message list. You may need to call *flagmess* if you implement a new feature.

`flagrule ( )`

flags the rule list as invalid. See the previous explanation o *,:agmess*.

`generate ( )`

does one generation of the classifier system. Each generation applies the rule list to the message list, produces a new message list, and updates the strength of the rules.

`invert ( )`

picks a rule at random and creates a new rule by inverting some of the bits.

`message ( string )`

sends a new message to the classifier system. *string* may be any expression which evaluates to a string. The characters in *string* should be "0" or "1".

`mutate ( )`

picks a rule at random and creates a new rule by replacing some of the bits.

`open ( file, arg )`

opens a connection ("pipe") to a user classifier system. If the first parameter *file* is given, then it must be a string containing the name of the classifier's executable file. If the second parameter *arg* is given, then it must be a string to be given to the classifier as the first argument on its "command" line. If either parameter is missing, or NULL, then the defaults will be used. The classifier system is expected to return the string "ready" followed by a newline. (*open* is implicitly done on the first call to *receive* or *send*.)

`payoff ( number )`

sends a pay-off *number* to the classifier system. Positive numbers usually mean a successful result ("win"); negative numbers usually mean an error ("loss").

`receive ( string )`

receives a line from the classifier system. A line consists of all characters except for the newline. If *string* is missing or NULL, then the line is returned as this function's value in the form of a string. If *string* is given, then it is an expected reply from the classifier system; anything else will be considered an error. (No function value is returned in this last case.)

`rule ( list )`

sends a new rule to the classifier system. *list* must be a list of three elements: the first element is a list of condition strings; the second element is an action string; the third element is a strength number.

`send ( string )`

sends a string to the classifier system, followed by a newline character. The newline is added by this function, and should not appear in your *string*. No response is expected by this function: you may need to call *receive* to read a reply from the classifier.

Note: if you send a command that should be done by a pre-defined function, then the classifier data in `face` may not be consistent with the correct data in the classifier system. See *flagmess* and *flagrule*.

`switch ( number )`

switches to a new copy of the classifier system. *number* should be a number from 1 to 9, or it may be NULL for the default value of 1. Each copy of the classifier system has its own message and rule lists. The current switch value is available in the global variable *switchnumber*.

A.4. Running face

A copy of face is in the directory:

/u1/grad/fenske/Face

on the "pembina" machine. You should copy face to one of your directories. You may also want to copy the sample programs:

pretty.f   prime.f   user.f

and will need a copy of the user classifier system. To run face, type:

face

face will start running, and will print this introduction:

```
face da class: an interactive
classifier programming language

ready for input
```

You may now type any legal expression, statement, or function definition. Please remember that every line must end with a semicolon (";").

To exit from face, type:

```
exit();
```

or the end-of-file character for your terminal (usually control-D).

A.4.1. Command Options

On the command line that invokes `face`, you may specify the following options:

`-a`*string*

sets the string to be given to the user classifier system as the first argument on its command line. The default string is "-p", and may be explicitly given by "-a-p". The "p" signifies that input and output is to a process via a pipe.

`-f`*name*

sets the file name of the user classifier system. The default name is "robert", and may be explicitly given by "-frobert".

`-t`

traces all input from and output to the classifier system. This is normally only useful when debugging the classifier. The default is no tracing.

Any other options on the command line are assumed to be file names. The named files will be read and executed before commands are read from the terminal. This is a good way to load variables that were previously saved with the *save* function.

A.5. Restrictions

(1) Strings should be at most 999 characters long. When a string is allocated before its final size can be known, a maximum length of 999 bytes is assumed. The following routines are affected: formatted I/O with *scanf*, *sprintf*, and *sscanf* ("FioCheck" internal routine); explicit input strings in an executable statement or function definition ("LexString"); the pre-defined *pack* function ("PrePack"); the pre-defined *receive* function which reads from the user classifier system ("UserReceive"). This maximum size is determined by a *MAXSTRING* definition in the "fainc.h" file, and may be changed when rebuilding the compiler.

(2) Parameters to the formatted input and output functions (*printf*, *scanf*, *sprintf*, and *sscanf*) must agree with the fields specified in the format string. Input parameters must be assigned dummy values so that the pre-defined functions know what to use with the real system calls. Failure to do this will result in obscure core dumps ("crashes").

(3) File line numbers in error messages tell you where the compiler was when it detected the error (which may not be where the error is). If you mix statements and data in a file, then the data lines are not counted in the line number, because they are not read by the compiler.

Appendix B: Program Listings

The `face` programming language is constructed from seventeen different source modules which are built under the control of a *make* command file [Fe78]. Most of the code is written in the "C" programming language [Ke78]; the remainder is written in either LEX [Le78] or YACC [Jo78]. LEX and YACC are compiler-writing tools, and convert language descriptions into "C" code.

File	Lines	Description
`makefile`	110	dependencies and commands for rebuilding
`face.c`	102	main program
`faexe.c`	2,007	parse tree execution
`faglo.c`	36	global variables
`fainc.h`	256	includes: definitions and data types
`falex.l`	214	input lexical tokens (LEX)
`famem.c`	393	memory allocation
`fapre.c`	1,283	pre-defined language functions
`fasub.c`	1,046	support subroutines
`fause.c`	732	pre-defined user classifier support
`fayac.y`	703	language grammar syntax (YACC)
`kpr.c`	298	Keith's "pr" print utility
`telex.c`	129	test for lexical routines
`tepar.c`	172	test for parse tree routines
`terob.c`	111	test for Robert Chai's classifier
`pretty.f`	111	example user-defined function
`user.f`	179	user-defined classifier support functions
total lines	7,882	

Appendix C: Execution Profile

Programs written in the `face` language run approximately 100 times slower than the same program written in "C". Half of this delay can be attributed to interpreting the parse tree instead of generating compiled code. The remaining time is spent manipulating dynamic data objects.

During development, an execution profile was created with the UNIX *gprof* utility. This profile counts how often subroutines are called and estimates how much of the total CPU time is spent in each subroutine. Statistics are printed in descending order of CPU time. The information can be used to improve the performance of a program by changing sections where the most CPU time is spent.

The first working version of `face` had a simple approach to pushing and popping values from the stack. The *PushStack* routine cleared the *free* and *owner* stack fields, and allocated a new dummy value structure. *PopStack* released this dummy structure. For a test program which found all prime numbers from 1 to 100, there were 13,225 calls to the *malloc* dynamic memory allocation routine and 13,111 calls to the *free* de-allocation routine.

A newer version initialized all stack fields to zero or NULL at the beginning of the program. *PushStack* was changed to create a dummy value structure only if the current *dummy* pointer was NULL; otherwise, the old dummy structure was used. *PopStack* was changed to release the value pointed to by the dummy structure (but not the dummy structure itself), and to set this pointer to NULL. Calls to *malloc* for the same test program were reduced to 4,812; calls to *free* were reduced to 4,681. The new version used 5.06 seconds of CPU time compared to 7.88 seconds for the old version (1.56 times faster).

A comparison of some CPU times for the old and new versions follows:

subroutine	old version			new version		
	seconds	% total	calls	seconds	% total	calls
ExecParse	1.31	16.6		1.26	24.9	
FreeValue	0.96	12.2		0.30	5.9	
CheckStack	0.65	8.2		0.58	11.5	
malloc	0.62	7.9	13,225	0.25	4.9	4,812
MakeValue	0.45	5.7		0.14	2.8	
free	0.35	4.4	13,111	0.13	2.6	4,681
PushStack	0.35	4.4		0.36	7.1	
PopStack	0.33	4.2		0.40	7.9	
GetMemory	0.25	3.2		0.14	2.8	
FreeString	0.18	2.3		0.05	1.0	

Both versions spent roughly half of their time manipulating stack or dynamic data (*FreeValue* to *FreeString*). Fifty percent is a heavy price to pay for dynamic data. Even though no single subroutine consumes all of the time, there was room for improvement in the old version. The problem was to find a common area that affected all of the subroutines named above. Clearly, this was memory allocation. By reducing the number of calls to create and destroy stack dummy structures (a change to only a few lines of code), the total CPU time was reduced by one third.

```
# Face/makefile

# Keith Fenske
# Department of Computing Science
# The University of Alberta
# Edmonton, Alberta, Canada
# T6G 2H1

# December 1987

# Copyright (c) 1987 by Keith Fenske.  All rights reserved.


# defaults

CFLAGS=-O
copies=1
paper=default
printon=bothsides
priority=n
return=bins


# file lists

OBJECT = face.o faexe.o faglo.o famem.o fapre.o fasub.o fause.o lex.yy.o y.tab.o
SOURCE = makefile face.c faexe.c faglo.c fainc.h falex.l famem.c fapre.c \
        fasub.c fause.c fayac.y kpr.c telex.c tepar.c terob.c pretty.f user.f


# the whole thing

all : face guide.xp kpr
        @echo "§Face is ready"


# individual pieces

$(OBJECT) : fainc.h

face : $(OBJECT)
        cc $(OBJECT) -ll -lm -o face

guide.xp : guide.ms
        tbl guide.ms | eqn -Tx-r | troff -ms -Tx-r >guide.xp

kpr : kpr.c
        cc kpr.c -o kpr

lex.yy.c : falex.l
        lex falex.l

lex.yy.o : fainc.h lex.yy.c y.old.h

y.old.h : y.tab.h
        -cmp -s y.old.h y.tab.h || cp y.tab.h y.old.h

y.tab.c y.tab.h : fayac.y
        yacc -d fayac.y


# test programs (must be explicitly named to be built)

telex : faglo.o famem.o lex.yy.o telex.o
        cc faglo.o famem.o lex.yy.o telex.o -ll -o telex

telex.o : fainc.h telex.c y.old.h

tepar : faglo.o famem.o lex.yy.o tepar.o y.tab.o
        cc faglo.o famem.o lex.yy.o tepar.o y.tab.o -ll -o tepar
```

```

tepar.o : fainc.h tepar.c

terob : fainc.h terob.c
	cc terob.c -o terob


# utility functions

clean :
	-rm a.out core lex.yy.c telex tepar terob y.old.h y.output y.tab.[ch]
	-rm *.o *.xp

count :
	wc $(SOURCE)

pembina :
	rcp face pretty.f prime.f user.f pembina:Face

permit :
	-chmod 644 *
	-chmod 755 face kpr telex tepar terob

print : print.guide print.prime print.source

print.guide : guide.xp
	mpr -i -m -p"copies=$(copies) paper=$(paper) printon=$(printon) \
		priority=$(priority) return=$(return)" <guide.xp

print.prime : kpr
	kpr -p269 -s prime.f prime.s prime.p >prime.xp
	mpr -i -m -p"copies=$(copies) paper=$(paper) printon=$(printon) \
		priority=$(priority) return=$(return)" <prime.xp

print.source : kpr
	kpr -n -p148 -s $(SOURCE) >source.xp
	mpr -i -m -p"copies=$(copies) pages=200 paper=$(paper) \
		printon=$(printon) priority=$(priority) return=$(return)" \
		<source.xp
```

```c
/*

face.c -- Classifier Interface


Keith Fenske
Department of Computing Science
The University of Alberta
Edmonton, Alberta, Canada
T6G 2H1

December 1987

Copyright (c) 1987 by Keith Fenske.  All rights reserved.


This is the main program for the interactive classifier interface ("face").
The following arguments ("flags", "options", "parameters", "switches") may be
given on the command line:

	-a<string>
		sets the string which will be given to the user classifier
		system as its first argument.  The default "<string>" is "-p"
		to indicate that commands are coming from a pipe.

	-f<name>
		sets the executable file name of the user classifier system.
		The default "<name>" is "robert".

	-t
		traces all I/O with the classifier system.  Generally only used
		for debugging.  The default is no tracing.

Any other arguments must be file names.  These files will be parsed and
executed before input is read from the terminal (standard input).  This is a
good way to load global variables previously saved with the "save" function.

*/

#include "fainc.h"				/* our standard includes */


main(argc, argv)
	int argc;				/* number of arguments */
	char * argv[];				/* argument strings */
{
	int i;					/* index variable */

	/* introduction */

	printf("\nface da class: an interactive classifier programming language\n");

	/* pre-defined symbols */

	ClearStack();				/* get execution stack ready */
	PreDefine();				/* do pre-defined symbols */
	UserDefine();				/* do user-defined symbols */

	/* process command line arguments */

	for (i = 1 ; i < argc ; i ++)
	{
		if (argv[i][0] == '-')
		{
			switch (argv[i][1])
			{
			case ('a'):
			case ('A'):
				UserArg = & argv[i][2];
				break;
```

```
                        case ('f'):
                        case ('F'):
                                UserFile = & argv[i][2];
                                break;
                        case ('t'):
                        case ('T'):
                                UserTrace = YES;
                                break;
                        default:
                                fprintf(stderr, "%s: unknown flag: '%s'\n",
                                        argv[0], argv[i]);
                                exit(-1);
                                break;
                        }
                }
                else
                {
                        /* must be a file name to load and execute */

                        ExecFile(argv[i]);
                }
        }

        /* now read and execute from standard input */

        printf("\nready for input\n");
        ExecFile(NULL);

        /* exit back to UNIX (or our parent process) */

        PreExit(NULL);
}
```

```
/*

faexe.c -- Execute Parse Tree


Keith Fenske
Department of Computing Science
The University of Alberta
Edmonton, Alberta, Canada
T6G 2H1

December 1987

Copyright (c) 1987 by Keith Fenske.  All rights reserved.


These are the routines to execute a parse tree built up by "fayac.y".  The YACC
grammar does essentially no checking while creating the parse tree.  It is up
to the main routine "ExecParse" and its subroutines to verify that the actions
in the parse tree are meaningful.

*/

#include "fainc.h"                    /* our standard includes */
#include <math.h>                     /* math routines */
#include <setjmp.h>                   /* system long jump */


/*
Define a type for saving "jump buffers", which are used by "setjmp" and
"longjmp" for stack information.  We make the whole buffer into a structure,
so that we can assign them easily.
*/
        typedef struct { jmp_buf env; } JumpThing;

        JumpThing ErrorJump;          /* global jump for errors */
        JumpThing ReturnJump;         /* global jump for function returns */


/*
Define some logic tables for the bit string operations AND and OR.  Each table
is for 0, 1, don't care ("#"), and illegal ("?").   The tables are:

                AND| 0 | 1 | # | ? |
                ---+---+---+---+---+
                 0 | 0 | ? | 0 | ? |
                ---+---+---+---+---+
                 1 | ? | 1 | 1 | ? |
                ---+---+---+---+---+
                 # | 0 | 1 | # | ? |
                ---+---+---+---+---+
                 ? | ? | ? | ? | ? |
                ---+---+---+---+---+

                OR | 0 | 1 | # | ? |
                ---+---+---+---+---+
                 0 | 0 | # | # | 0 |
                ---+---+---+---+---+
                 1 | # | 1 | # | 1 |
                ---+---+---+---+---+
                 # | # | # | # | # |
                ---+---+---+---+---+
                 ? | 0 | 1 | # | ? |
                ---+---+---+---+---+

This may look funny (why is "0" AND'ed with "1" undefined?), until you accept
that bit strings represent all possible binary values that match a given
pattern.  No bit string can have both "0" and "1" in the same position.
*/
```

```
        char AndTable[4][4] = { '0', BADBIT, '0', BADBIT,
                                BADBIT, '1', '1', BADBIT,
                                '0', '1', ANYBIT, BADBIT,
                                BADBIT, BADBIT, BADBIT, BADBIT };

        char OrTable[4][4] = { '0', ANYBIT, ANYBIT, '0',
                                ANYBIT, '1', ANYBIT, '1',
                                ANYBIT, ANYBIT, ANYBIT, ANYBIT,
                                '0', '1', ANYBIT, BADBIT };


/*
CheckStack()

Check that the current stack pointer is valid.  Generate an internal error
message if not.
*/

CheckStack()
{
        if (SP < 0)
        {
                PrintLine();
                fprintf(stderr,
                        "internal error: CheckStack SP = %d is negative\n",
                        SP);
                SP = 0;                         /* fix */
                ExecAbort();
        }
        else if (SP >= STACKSIZE)
        {
                PrintLine();
                fprintf(stderr,
                        "internal error: CheckStack SP = %d not less than %d\n",
                        SP, STACKSIZE);
                SP = STACKSIZE - 1;             /* fix */
                ExecAbort();
        }
}


/*
ClearStack()

Clear all entries in the stack so that it is safe for PushStack() to assume
that all non-NULL "dummy" fields point to a legitimate dummy value structure.
*/

ClearStack()
{
        int i;                          /* index variable */

        for (i = 0 ; i < STACKSIZE ; i ++)
        {
                Stack[i].dummy = NULL;
                Stack[i].free = YES;
                Stack[i].owner = NULL;
        }
        FP = SP = 0;                    /* stack pointers */
}



/*
DumpStack()

Debugging routine to dump the entire stack.
*/

DumpStack()
```

```
{
        int i;                          /* index variable */

        printf("\nStack dump with FP = %d and SP = %d\n", FP, SP);
        for (i = 0 ; i <= SP ; i ++)
        {
                printf("Stack %d : dummy %x  free %d  owner %x  value ", i,
                        Stack[i].dummy, Stack[i].free, Stack[i].owner);
                PrintValue(stdout, Stack[i].dummy, YES);
                printf("\n");
        }
}


/*
ExecAbort()

Abort this call to ExecFile() by doing a long jump back into the most recent
call to "setjmp".  It may be ugly, but it works.
*/

ExecAbort()
{
        longjmp(ErrorJump.env, YES);
}


/*
ExecAssign()

The caller has pushed a left side onto the stack at [SP-1] and a right side at
[SP], and now wants us to assign the right side to the left side.  This is used
by OpASSIGN and OpFOR.  The right side is popped off the stack, but we leave
the assigned left side.
*/

ExecAssign()
{
        if (Stack[SP-1].owner != NULL)
        {
                /* free any old value owned by the owner */

                FreeValue(Stack[SP-1].owner->this);

                /* copy the new value */

                MakeDynamic(SP);        /* copy right side */
                Stack[SP].free = NO;    /* but assignment kills free */
                Stack[SP-1].free = NO;  /* this is not the real variable! */
                Stack[SP-1].dummy->this = Stack[SP].dummy->this;
                Stack[SP-1].owner->this = Stack[SP].dummy->this;
                PopStack();             /* kill right side */
                                        /* keep new left side */
        }
        else
        {
                PrintLine();
                fprintf(stderr, "left side of ':=' is not assignable\n");
                ExecAbort();
        }
}


/*
ExecCompare()

Compare two values, given four sign flags (equal, greater than, less than, or
not comparable) and a flag to indicate if don't cares ("#") are acceptable in
strings.
*/
```

```

ExecCompare(par, equal, greater, less, noteq, pattern)
        ParseThing * par;               /* a parse tree pointer */
        int equal;                      /* YES if left = right is okay */
        int greater;                    /* YES if left > right is okay */
        int less;                       /* YES if left < right is okay */
        int noteq;                      /* YES if not comparable is okay */
        int pattern;                    /* YES if "#" special in strings */
{
        int compare;                    /* result from CompareValue() */
        NUMBER result;                  /* FALSE or TRUE */

        ExecParse(par->one);            /* left side */
        ExecParse(par->two);            /* right side */

        compare = CompareValue(Stack[SP-1].dummy->this, Stack[SP].dummy->this,
                pattern);

        PopStack();                     /* kill right side */
        PopStack();                     /* kill left side */

        if (((compare == CMPEQ) && equal)
        || ((compare == CMPGT) && greater)
        || ((compare == CMPLT) && less)
        || ((compare == CMPNE) && noteq))
                result = TRUE;
        else
                result = FALSE;

        PushStack();                    /* our result */
        Stack[SP].dummy->this = MakeValue(ValNUMBER);
        Stack[SP].dummy->this->number = result;
}


/*
ExecFile()

Read, parse, and execute the statements in a given file (or standard input if
the file name string pointer is NULL).  Return on an end-of-file or error.
Ignore errors when reading from standard input.
*/

ExecFile(cp)
        char * cp;                      /* file name string pointer */
{
        FILE * fp;                      /* new file pointer */
        JumpThing olderror;             /* old (previous) error jump */
        FILE * oldfp;                   /* old (previous) file pointer */
        int oldframe;                   /* old (previous) frame pointer */
        int oldline;                    /* old (previous) file line number */
        JumpThing oldreturn;            /* old (previous) function return */
        int oldstack;                   /* old (previous) stack pointer */

        oldframe = FP;                  /* must return to this frame */
        oldstack = SP;                  /* must return to this stack */

        /* open the file for reading */

        oldfp = yyin;                   /* save LEX file pointer */
        oldline = LineNumber;           /* save file line number */

        if (cp == NULL)
        {
                fp = stdin;             /* use standard input */
                LineNumber = -9999;     /* fake file line number */
        }
        else
        {
                printf("loading file '%s'\n", cp);
```

```
		fp = fopen(cp, "r");
		if (fp == NULL)
		{
			fprintf(stderr,
				"load failed: can't open file '%s' for reading\n",
				cp);
			return;
		}
		LineNumber = 1;
	}

	yyin = fp;			/* hopefully, re-direct LEX input */

	/* parse statements, with a long jump set for errors */

	olderror = ErrorJump;		/* save previous error jump */
	oldreturn = ReturnJump;		/* save previous function return */

	while (!feof(fp))
	{
		int status;		/* status of called function */
		int thisline;		/* file line number */

		/* save file line number, which ExecParse changes */

		thisline = LineNumber;

		if ((setjmp(ErrorJump.env) == 0)
		&& (setjmp(ReturnJump.env) == 0))
		{
			ParseTree = NULL;
			status = yyparse();
			thisline = LineNumber;
			if (status == 0)
			{
				ExecParse(ParseTree);
				FreeParse(ParseTree);
			}
			else if (cp != NULL)
				break;
		}
		else
		{
			/* fix up the stack after abort */

			if (FP != oldframe)
			{
				/*
				printf("restoring frame pointer from %d to %d\n",
					FP, oldframe);
				*/
				FP = oldframe;
			}

			if (SP > oldstack)
			{
				/*
				printf("restoring stack pointer from %d to %d\n",
					SP, oldstack);
				*/
				while (SP > oldstack)
					PopStack();
			}

			/* stop looking at this file, if not stdin */

			if (cp != NULL)
				break;
		}
		LineNumber = thisline;		/* restore file line number */
```

```
	}

	ErrorJump = olderror;			/* restore previous error jump */
	ReturnJump = oldreturn;			/* restore previous function return */

	/* print pretty messages for end-of-file */

	if (cp == NULL)
		printf("\nend-of-file on standard input\n");
	else
	{
		if (feof(fp))
			printf("\nend-of-file on file '%s'\n", cp);
		else
			printf("\nclosing file '%s'\n", cp);
		fclose(fp);
	}

	yyin = oldfp;				/* hopefully, restore LEX input */
	LineNumber = oldline;			/* restore file line number */
}


/*
ExecFunction()

Execute a parse tree node as a function call.  This is used by OpNAME and
OpFUNCTION.  After all of the parameters have been generated, the new function
is given a frame pointer (FP) for its parameters so that its stack looks like
this:

			Stack[FP] = function result (initially NULL)
			Stack[FP+1] = first parameter
			Stack[FP+2] = second parameter
				    .
				    .
				    .

For user-defined functions, we copy by value any parameters which have not been
declared as "passed by address" (free = NO).  Pre-defined functions must take
care of themselves.
*/

ExecFunction(par)
	ParseThing * par;			/* a parse tree pointer */
{
	int i;					/* index variable */
	int newframe;				/* new frame pointer (FP) */
	int oldframe;				/* old (previous) frame pointer (FP) */
	JumpThing oldreturn;			/* old (previous) function return */
	SymbolThing * sym;			/* a symbol table pointer */

	oldframe = FP;				/* save previous frame pointer */
	oldreturn = ReturnJump;			/* save previous function return */

	/* check that this symbol really is a function */

	if (par->symbol->type != SymFUNCTION)
	{
		PrintLine();
		fprintf(stderr, "symbol '%s' is not a function\n",
			par->symbol->name);
		ExecAbort();
	}

	/* make room for a result */

	PushStack();
	newframe = SP;				/* where new frame pointer will be */

```

```
        /* copy the caller's parameter list to the stack */
        /* can't use new FP yet, because some parameters may be local */

        ExecParse(par->one);

        /* add NULL parameters for anything missing */

        FP = newframe;
        while ((SP - FP) < par->symbol->count)
                PushStack();

        /* call the function (user-defined or pre-defined) */

        switch (par->symbol->special)
        {
        case (0):
                /* user-defined function */
                /* check if parameters should be made dynamic (copied) */

                i = FP + 1;             /* stack entry for first parameter */
                sym = par->symbol->local->next;         /* first parameter */
                while (sym != NULL)
                {
                        if (sym->free == NO)
                        {
                                /* may be passed by address */
                        }
                        else
                        {
                                /* must be passed by value (copied) */

                                MakeDynamic(i);
                        }
                        i ++;
                        sym = sym->next;
                }

                /* call the user-defined function, with a return trap */

                if (setjmp(ReturnJump.env) == 0)
                {
                        ExecParse(par->symbol->parse);
                }
                else
                {
                        /* must be an early "return" statement */
                }
                break;

        case (SpeABS):
                PreAbs(par);
                break;
        case (SpeACOS):                         /* math routine */
                PreMath(par, acos, "acos function");
                break;
        case (SpeASIN):                         /* math routine */
                PreMath(par, asin, "asin function");
                break;
        case (SpeATAN):                         /* math routine */
                PreMath(par, atan, "atan function");
                break;
        case (SpeATAN2):                        /* math routine */
                PreMath(par, atan2, "atan2 function");
                break;
        case (SpeCBRT):                         /* math routine */
                PreMath(par, cbrt, "cbrt function");
                break;
        case (SpeCLOSE):                        /* user classifier */
                PreClose(par);
                break;
```

```
        case (SpeCOS):                                  /* math routine */
                PreMath(par, cos, "cos function");
                break;
        case (SpeEXIT):
                PreExit(par);
                break;
        case (SpeEXP):                                  /* math routine */
                PreMath(par, exp, "exp function");
                break;
        case (SpeFLAGMESS):                             /* user classifier */
                PreFlagMess(par);
                break;
        case (SpeFLAGRULE):                             /* user classifier */
                PreFlagRule(par);
                break;
        case (SpeLOAD):
                PreLoad(par);
                break;
        case (SpeLOG):                                  /* math routine */
                PreMath(par, log, "log function");
                break;
        case (SpeLOG10):                                /* math routine */
                PreMath(par, log10, "log10 function");
                break;
        case (SpeOPEN):                                 /* user classifier */
                PreOpen(par);
                break;
        case (SpePACK):
                PrePack(par);
                break;
        case (SpePOW):                                  /* math routine */
                PreMath(par, pow, "pow function");
                break;
        case (SpePRINTF):
                PrePrintf(par);
                break;
        case (SpeRANDOM):
                PreRandom(par);
                break;
        case (SpeRECEIVE):                              /* user classifier */
                PreReceive(par);
                break;
        case (SpeROUND):
                PreRound(par);
                break;
        case (SpeSAVE):
                PreSave(par);
                break;
        case (SpeSCANF):
                PreScanf(par);
                break;
        case (SpeSEND):                                 /* user classifier */
                PreSend(par);
                break;
        case (SpeSIGN):
                PreSign(par);
                break;
        case (SpeSIN):                                  /* math routine */
                PreMath(par, sin, "sin function");
                break;
        case (SpeSIZE):
                PreSize(par);
                break;
        case (SpeSPRINTF):
                PreSprintf(par);
                break;
        case (SpeSQRT):                                 /* math routine */
                PreMath(par, sqrt, "sqrt function");
                break;
        case (SpeSSCANF):
```

```
                PreSscanf(par);
                break;
        case (SpeSTOP):
                PreStop(par);
                break;
        case (SpeSYSTEM):
                PreSystem(par);
                break;
        case (SpeTAN):                          /* math routine */
                PreMath(par, tan, "tan function");
                break;
        case (SpeTRUNC):
                PreTrunc(par);
                break;
        case (SpeTYPE):
                PreType(par);
                break;
        case (SpeUNPACK):
                PreUnpack(par);
                break;
        case (SpeVALUE):
                PreValue(par);
                break;
        case (SpeWRITE):
                PreWrite(par);
                break;
        default:
                PrintLine();
                fprintf(stderr,
                        "internal error: ExecFunction symbol special = %d\n",
                        par->symbol->special);
                ExecAbort();
        }

        /* pop everything off the stack, except for the result */

        while (FP < SP)
                PopStack();

        FP = oldframe;                  /* restore previous frame pointer */
        ReturnJump = oldreturn;         /* restore previous function return */
}


/*
ExecName()

Given the address of a symbol table entry, push the value of the symbol onto
the stack.  This is used by OpFOR and OpNAME.
*/

ExecName(sym)
        SymbolThing * sym;              /* a symbol table pointer */
{
        int n;                          /* stack entry number */

        switch (sym->type)
        {
        case (SymFUNCTION):
                PrintLine();
                fprintf(stderr, "function '%s' can not be used here\n",
                        sym->name);
                ExecAbort();
                break;
        case (SymGLOBAL):
                PushStack();
                if (sym->dummy == NULL)
                        sym->dummy = MakeValue(ValDUMMY);
                if (sym->special == 0)
                {
```

```
					Stack[SP].dummy->this = sym->dummy->this;
					Stack[SP].free = NO;
					Stack[SP].owner = sym->dummy;
				}
				else
				{
					/* must be a user classifier variable */

					UserOpName(sym);
				}
				break;
			case (SymLOCAL):
				PushStack();
				n = FP + sym->offset;
				if (Stack[n].owner == NULL)
				{
					/* regular local symbol not attached anywhere else */

					Stack[SP].dummy->this = Stack[n].dummy->this;
					Stack[SP].free = NO;
					Stack[SP].owner = Stack[n].dummy;
				}
				else
				{
					/* local copy of a more global symbol */
					/* value may have changed; get new value from owner */

					Stack[SP].dummy->this = Stack[n].owner->this;
					Stack[SP].free = NO;
					Stack[SP].owner = Stack[n].owner;
				}
				break;
			default:
				PrintLine();
				fprintf(stderr, "internal error: ExecName symbol type = %d\n",
					sym->type);
				ExecAbort();
				break;
		}
}


/*
ExecParse()

Recursively execute a parse tree.  This is one big switch statement, where each
case pushes and pops the stack in its own way.  See OpSTMT for some error
checking. . . .
*/

ExecParse(par)
	ParseThing * par;		/* a parse tree pointer */
{
	int error;			/* YES means bad values */
	int j, k, m, n;			/* working integers */
	int oldframe;			/* old (previous) frame pointer */
	int oldline;			/* old (previous) file line number */
	int oldstack;			/* old (previous) stack pointer */
	ValueThing * val;		/* a value structure pointer */
	NUMBER w, x, y, z;		/* working numbers */

	if (par == NULL)
		return;

	/* change file line number to value saved in parse tree */

	oldline = LineNumber;
	LineNumber = par->line;

	/* check stack, before doing anything important */
```

```

        /* CheckStack(); */     /* leave this to PopStack, PushStack */

        /* one big, big switch statement */

        switch (par->type)
        {

/* and */
/* or */

case (OpAND):
case (OpOR):
        PushStack();                        /* our result */
        ExecParse(par->one);                /* left side */
        ExecParse(par->two);                /* right side */

        error = NO;                         /* assume no errors */

        if ((Stack[SP-1].dummy->this == NULL)
        || (Stack[SP].dummy->this == NULL))
        {
                error = YES;
        }
        else if ((Stack[SP-1].dummy->this->type == ValNUMBER)
        && (Stack[SP].dummy->this->type == ValNUMBER))
        {
                x = Stack[SP-1].dummy->this->number;
                y = Stack[SP].dummy->this->number;

                switch (par->type)
                {
                case (OpAND):
                        if (x == FALSE)
                                if ((y == FALSE) || (y == TRUE))
                                        w = FALSE;
                                else
                                        error = YES;
                        else if (x == TRUE)
                                if (y == FALSE)
                                        w = FALSE;
                                else if (y == TRUE)
                                        w = TRUE;
                                else
                                        error = YES;
                        else
                                error = YES;
                        break;
                case (OpOR):
                        if (x == FALSE)
                                if (y == FALSE)
                                        w = FALSE;
                                else if (y == TRUE)
                                        w = TRUE;
                                else
                                        error = YES;
                        else if (x == TRUE)
                                if ((y == FALSE) || (y == TRUE))
                                        w = TRUE;
                                else
                                        error = YES;
                        else
                                error = YES;
                        break;
                default:
                        PrintLine();
                        fprintf(stderr,
                                "internal error: OpAND parse type = %d\n",
                                par->type);
                        error = YES;
```

```
				break;
			}
			if (!error)
			{
				Stack[SP-2].dummy->this = val = MakeValue(ValNUMBER);
				val->number = w;
			}
	}
	else if ((Stack[SP-1].dummy->this->type == ValSTRING)
	&& (Stack[SP].dummy->this->type == ValSTRING))
	{
		char * left, * right, * result;

		left = Stack[SP-1].dummy->this->string;
		right = Stack[SP].dummy->this->string;
		result = NULL;

		j = strlen(left);
		k = strlen(right);

		CheckSign(Stack[SP-1].dummy->this);
		CheckSign(Stack[SP].dummy->this);

		if ((j == k)
		&& (Stack[SP-1].dummy->this->sign ==
			Stack[SP].dummy->this->sign))
		{
			Stack[SP-2].dummy->this = val = MakeValue(ValSTRING);
			val->sign = Stack[SP].dummy->this->sign;
			val->string = result = GetMemory(k + 1);

			while (*left)
			{
				if ((*left) == '0')
					m = 0;
				else if ((*left) == '1')
					m = 1;
				else if ((*left) == ANYBIT)
					m = 2;
				else if ((*left) == BADBIT)
					m = 3;
				else
				{
					error = YES;
					break;
				}

				if ((*right) == '0')
					n = 0;
				else if ((*right) == '1')
					n = 1;
				else if ((*right) == ANYBIT)
					n = 2;
				else if ((*right) == BADBIT)
					n = 3;
				else
				{
					error = YES;
					break;
				}

				if (par->type == OpAND)
					(*result) = AndTable[m][n];
				else
					(*result) = OrTable[m][n];

				left ++;
				right ++;
				result ++;
			}
```

```
                                        (*result) = '\0';
                                }
                                else
                                        error = YES;
                        }
                        else
                                error = YES;

                        if (error)
                        {
                                PrintLine();
                                fprintf(stderr, "bad values in 'and' or 'or': ");
                                PrintValue(stderr, Stack[SP-1].dummy, YES);
                                fprintf(stderr, " and ");
                                PrintValue(stderr, Stack[SP].dummy, YES);
                                fprintf(stderr, "\n");
                                ExecAbort();
                        }
                        PopStack();                     /* kill right side */
                        PopStack();                     /* kill left side */
                        break;

                /* assignment */

                case (OpASSIGN):
                        ExecParse(par->one);            /* left side */
                        ExecParse(par->two);            /* right side */
                        ExecAssign();                   /* do assignment */
                        break;

                /* concatenate set elements */

                case (OpCONCAT):
                        PushStack();
                        Stack[SP].dummy->this = val = MakeValue(ValELEMENT);

                        /* generate new (left) element */

                        ExecParse(par->one);
                        MakeDynamic(SP);
                        val->this = Stack[SP].dummy->this;      /* link */
                        Stack[SP].free = NO;
                        PopStack();

                        /* generate right recursive tail */

                        if (par->two != NULL)
                        {
                                ExecParse(par->two);
                                MakeDynamic(SP);
                                val->next = Stack[SP].dummy->this;      /* link */
                                Stack[SP].free = NO;
                                PopStack();
                        }
                        break;

                /* division (integer quotient) */
                /* modulo (integer remainder) */

                case (OpDIV):
                case (OpMOD):
                        PushStack();                    /* our result */
                        ExecParse(par->one);            /* left side */
                        ExecParse(par->two);            /* right side */

                        error = NO;                     /* assume no errors */

                        if ((Stack[SP-1].dummy->this == NULL)
                        || (Stack[SP].dummy->this == NULL)
                        || (Stack[SP-1].dummy->this->type != ValNUMBER)
```

```
        || (Stack[SP].dummy->this->type != ValNUMBER))
        {
                error = YES;
        }
        else
        {
                long i, j, k;           /* use long integers here */

                x = Stack[SP-1].dummy->this->number;    /* left side */
                y = Stack[SP].dummy->this->number;      /* right side */

                j = (long) x;
                k = (long) y;

                if ((x != (NUMBER) j) || (y != (NUMBER) k))
                {
                        /* truncation changes value */

                        error = YES;
                }
                else if (k == 0)
                {
                        /* can't divide by zero */

                        error = YES;
                }
                else
                {
                        switch (par->type)
                        {
                        case (OpDIV):
                                i = j / k;
                                break;
                        case (OpMOD):
                                i = j % k;
                                break;
                        default:
                                PrintLine();
                                fprintf(stderr,
                                        "internal error: OpDIV parse type = %d\n",
                                        par->type);
                                error = YES;
                                break;
                        }
                        w = (NUMBER) i;
                        Stack[SP-2].dummy->this = val = MakeValue(ValNUMBER);
                        val->number = w;
                }
        }

        if (error)
        {
                PrintLine();
                fprintf(stderr, "bad values in 'div' or 'mod': ");
                PrintValue(stderr, Stack[SP-1].dummy, YES);
                fprintf(stderr, " and ");
                PrintValue(stderr, Stack[SP].dummy, YES);
                fprintf(stderr, "\n");
                ExecAbort();
        }

        PopStack();                     /* kill right side */
        PopStack();                     /* kill left side */
        break;

/* equal relation */
/* equal pattern relation */
/* greater than or equal relation */
/* greater than relation */
/* less than or equal relation */
```

```
/* less than relation */
/* not equal relation */
/* not equal pattern relation */

case(OpEQ):
        ExecCompare(par, YES, NO, NO, NO, NO);
        break;
case(OpEQP):
        ExecCompare(par, YES, NO, NO, NO, YES);
        break;
case(OpGE):
        ExecCompare(par, YES, YES, NO, NO, NO);
        break;
case(OpGT):
        ExecCompare(par, NO, YES, NO, NO, NO);
        break;
case(OpLE):
        ExecCompare(par, YES, NO, YES, NO, NO);
        break;
case(OpLT):
        ExecCompare(par, NO, NO, YES, NO, NO);
        break;
case(OpNE):
        ExecCompare(par, NO, YES, YES, YES, NO);
        break;
case(OpNEP):
        ExecCompare(par, NO, YES, YES, YES, YES);
        break;

/* for */

case (OpFOR):

        /* get initial ("from") value */

        if (par->one == NULL)
                x = ONE;
        else
        {
                ExecParse(par->one);
                if ((Stack[SP].dummy->this == NULL)
                || (Stack[SP].dummy->this->type != ValNUMBER))
                {
                        PrintLine();
                        fprintf(stderr,
                                "bad 'from' value in 'for' statement: ");
                        PrintValue(stderr, Stack[SP].dummy, YES);
                        fprintf(stderr, "\n");
                        ExecAbort();
                }
                x = Stack[SP].dummy->this->number;
                PopStack();
        }

        /* get final ("to") value */

        if (par->two == NULL)
                y = ONE;
        else
        {
                ExecParse(par->two);
                if ((Stack[SP].dummy->this == NULL)
                || (Stack[SP].dummy->this->type != ValNUMBER))
                {
                        PrintLine();
                        fprintf(stderr, "bad 'to' value in 'for' statement: ");
                        PrintValue(stderr, Stack[SP].dummy, YES);
                        fprintf(stderr, "\n");
                        ExecAbort();
                }
```

```
                        y = Stack[SP].dummy->this->number;
                        PopStack();
                }

                /* get increment ("by") value */

                if (par->three == NULL)
                        z = ONE;
                else
                {
                        ExecParse(par->three);
                        if ((Stack[SP].dummy->this == NULL)
                        || (Stack[SP].dummy->this->type != ValNUMBER)
                        || (Stack[SP].dummy->this->number == ZERO))
                        {
                                PrintLine();
                                fprintf(stderr, "bad 'by' value in 'for' statement: ");
                                PrintValue(stderr, Stack[SP].dummy, YES);
                                fprintf(stderr, "\n");
                                ExecAbort();
                        }
                        z = Stack[SP].dummy->this->number;
                        PopStack();
                }

                /* execute for loop */

                for (w = x ; ; w += z)
                {
                        /* assign value to index variable */

                        ExecName(par->symbol);          /* push variable onto stack */
                        PushStack();                    /* create a value to assign */
                        Stack[SP].dummy->this = val = MakeValue(ValNUMBER);
                        val->number = w;
                        ExecAssign();                   /* do the assignment */
                        PopStack();                     /* kill assigned value */

                        /* check loop condition */

                        if (((z < ZERO) && (w < y))
                        || ((z >= ZERO) && (w > y)))
                                break;

                        /* execute statement body */

                        ExecParse(par->four);
                }
                break;

        /* function call */

        case (OpFUNCTION):
                ExecFunction(par);
                break;

        /* if */

        case (OpIF):
                ExecParse(par->one);            /* test condition */

                error = NO;                     /* assume no errors */

                if ((Stack[SP].dummy->this == NULL)
                || (Stack[SP].dummy->this->type != ValNUMBER))
                {
                        error = YES;
                }
                else
                {
```

```
                x = Stack[SP].dummy->this->number;

                /* check test condition */

                if (x == TRUE)
                {
                        ExecParse(par->two);
                }
                else if (x == FALSE)
                {
                        ExecParse(par->three);
                }
                else
                        error = YES;
        }

        if (error)
        {
                PrintLine();
                fprintf(stderr, "bad condition in 'if' statement: ");
                PrintValue(stderr, Stack[SP].dummy, YES);
                fprintf(stderr, "\n");
                ExecAbort();
        }

        PopStack();                     /* kill test condition */
        break;

/* index (subscript or subrange) */

case (OpINDEX):
        PushStack();                    /* our result */

        /* generate expression to be indexed */

        ExecParse(par->one);

        if ((Stack[SP].dummy->this == NULL)
        || ((Stack[SP].dummy->this->type != ValSET)
        && (Stack[SP].dummy->this->type != ValSTRING)))
        {
                PrintLine();
                fprintf(stderr,
                        "subscripted expression must be a set or string: ");
                PrintValue(stderr, Stack[SP].dummy, YES);
                fprintf(stderr, "\n");
                ExecAbort();
        }

        /* generate index lower bound */

        ExecParse(par->two);

        error = NO;                     /* assume no errors */

        if ((Stack[SP].dummy->this == NULL)
        || (Stack[SP].dummy->this->type != ValNUMBER))
        {
                error = YES;
        }
        else
        {
                x = Stack[SP].dummy->this->number;
                m = n = (long) x;

                if (x != (NUMBER) m)
                {
                        /* truncation changes value */

                        error = YES;
```

```
                }
                else if (m <= 0)
                {
                        /* lower bound must be positive */

                        error = YES;
                }
        }

        if (error)
        {
                PrintLine();
                fprintf(stderr,
                        "subscript lower bound must be a positive integer: ");
                PrintValue(stderr, Stack[SP].dummy, YES);
                fprintf(stderr, "\n");
                ExecAbort();
        }

        PopStack();

        /* generate index upper bound */

        if (par->three != NULL)
        {
                ExecParse(par->three);

                error = NO;                             /* assume no errors */

                if ((Stack[SP].dummy->this == NULL)
                || (Stack[SP].dummy->this->type != ValNUMBER))
                {
                        error = YES;
                }
                else
                {
                        x = Stack[SP].dummy->this->number;
                        n = (long) x;

                        if (x != (NUMBER) n)
                        {
                                /* truncation changes value */

                                error = YES;
                        }
                        else if (n < 0)
                        {
                                /* upper bound must be non-negative */

                                error = YES;
                        }
                }

                if (error)
                {
                        PrintLine();
                        fprintf(stderr,
                                "subscript upper bound must be a non-negative integ
                        PrintValue(stderr, Stack[SP].dummy, YES);
                        fprintf(stderr, "\n");
                        ExecAbort();
                }

                PopStack();
        }

        /* do set subscription */

        if (Stack[SP].dummy->this->type == ValSET)
        {
```

```
	CheckSign(Stack[SP].dummy->this);

	if (m > n)
	{
		/* return a null set */

		Stack[SP-1].dummy->this = MakeValue(ValSET);
		Stack[SP-1].dummy->this->sign =
			Stack[SP].dummy->this->sign;
	}
	else
	{
		ValueThing * val;	/* a value structure */

		/* find first element */

		j = 1;
		val = Stack[SP].dummy->this->next;
		while ((j < m) && (val != NULL))
		{
			j ++;
			val = val->next;
		}
		if (val == NULL)
		{
			PrintLine();
			fprintf(stderr, "bad subscript: index %d", m);
			fprintf(stderr, " greater than set size");
			fprintf(stderr, " %d for ", (j-1));
			PrintValue(stderr, Stack[SP].dummy, YES);
			fprintf(stderr, "\n");
			ExecAbort();
		}

		/* single elements may be assignable */

		if (par->three == NULL)
		{
			if ((Stack[SP].free == YES)
			|| (Stack[SP].owner == NULL))
			{
				/* free is never assignable */

				Stack[SP-1].dummy->this =
					CopyValue(val->this);
				Stack[SP-1].free = YES;
				Stack[SP-1].owner = NULL;
			}
			else
			{
				Stack[SP-1].dummy->this = val->this;
				Stack[SP-1].free = NO;
				Stack[SP-1].owner = val;
			}
		}

		/* subranges are copied (not assignable) */

		else
		{
			ValueThing * new;

			new = MakeValue(ValSET);
			new->sign = Stack[SP].dummy->this->sign;
			Stack[SP-1].dummy->this = new;
			Stack[SP-1].free = YES;
			Stack[SP-1].owner = NULL;

			while ((j <= n) && (val != NULL))
			{
```

```
						j ++;
						new->next = MakeValue(ValELEMENT);
						new->next->this = CopyValue(val->this);
						new = new->next;
						val = val->next;
					}
				}
			}
		}

		/* do string subscription */
		/* easier because the result is never assignable */

		else
		{
			char * lp, * rp;		/* left, right string */

			CheckSign(Stack[SP].dummy->this);

			Stack[SP-1].dummy->this = val = MakeValue(ValSTRING);

			if (par->three == NULL)
				val->sign = 1;		/* if [n] indexing */
			else
			{				/* if [m:n] indexing */
				val->sign = Stack[SP].dummy->this->sign;
			}

			rp = Stack[SP].dummy->this->string;
			k = strlen(rp);

			if (m > n)
			{
				/* return a null string */

				val->string = lp = GetMemory(1);
				(*lp) = '\0';
			}
			else if (m > k)
			{
				PrintLine();
				fprintf(stderr, "bad subscript: index %d", m);
				fprintf(stderr, " greater than string size");
				fprintf(stderr, " %d for ", k);
				PrintValue(stderr, Stack[SP].dummy, YES);
				fprintf(stderr, "\n");
				ExecAbort();
			}
			else
			{
				if (k < n)
					n = k;		/* adjust upper bound */

				val->string = lp = GetMemory(n - m + 2);
				rp = rp + m - 1;
				for ( ; m <= n ; m ++)
					(*lp ++) = (*rp ++);
				(*lp) = '\0';
			}
		}
		PopStack();			/* kill subscripted expression */
		break;

	/* subtraction ("-") */
	/* division ("/") */
	/* multiplication ("*") */

	case (OpMINUS):
	case (OpSLASH):
	case (OpSTAR):
```

```
        PushStack();                        /* our result */
        ExecParse(par->one);                /* left side */
        ExecParse(par->two);                /* right side */

        error = NO;                         /* assume no errors */

        if ((Stack[SP-1].dummy->this == NULL)
         || (Stack[SP].dummy->this == NULL)
         || (Stack[SP-1].dummy->this->type != ValNUMBER)
         || (Stack[SP].dummy->this->type != ValNUMBER))
        {
                error = YES;
        }
        else
        {
                x = Stack[SP-1].dummy->this->number;    /* left side */
                y = Stack[SP].dummy->this->number;      /* right side */

                switch (par->type)
                {
                case (OpMINUS):
                        w = x - y;
                        break;
                case (OpSLASH):
                        if (y == ZERO)
                                error = YES;
                        else
                                w = x / y;
                        break;
                case (OpSTAR):
                        w = x * y;
                        break;
                default:
                        PrintLine();
                        fprintf(stderr,
                                "internal error: OpMINUS parse type = %d\n",
                                par->type);
                        error = YES;
                        break;
                }
                if (!error)
                {
                        Stack[SP-2].dummy->this = val = MakeValue(ValNUMBER);
                        val->number = w;
                }
        }

        if (error)
        {
                PrintLine();
                fprintf(stderr, "bad values in '*', '-', or '/': ");
                PrintValue(stderr, Stack[SP-1].dummy, YES);
                fprintf(stderr, " and ");
                PrintValue(stderr, Stack[SP].dummy, YES);
                fprintf(stderr, "\n");
                ExecAbort();
        }

        PopStack();                         /* kill right side */
        PopStack();                         /* kill left side */
        break;

/* name (may be a function call) */

case (OpNAME):
        if (par->symbol->type == SymFUNCTION)
        {
                ExecFunction(par);

        else
```

```
        {
                ExecName(par->symbol);
        }
        break;

/* negate (unary minus) */

case (OpNEGATE):
        ExecParse(par->one);            /* right side */
        MakeDynamic(SP);
        val = Stack[SP].dummy->this;
        if (val == NULL)
        {
                /* negated NULL is still just NULL */
        }
        else if (val->type == ValNUMBER)
        {
                val->number = - val->number;
        }
        else if ((val->type == ValSET) || (val->type == ValSTRING))
        {
                CheckSign(val);
                val->sign = - val->sign;
        }
        else
        {
                PrintLine();
                fprintf(stderr, "internal error: OpNEGATE value type = %d\n",
                        val->type);
                ExecAbort();
        }
        break;

/* not (logical negation) */

case (OpNOT):
        PushStack();                    /* our result */
        ExecParse(par->one);            /* right side */

        error = NO;                     /* assume no errors */

        if (Stack[SP].dummy->this == NULL)
        {
                error = YES;
        }
        else if (Stack[SP].dummy->this->type == ValNUMBER)
        {
                x = Stack[SP].dummy->this->number;

                if (x == FALSE)
                        w = TRUE;
                else if (x == TRUE)
                        w = FALSE;
                else
                        error = YES;

                if (!error)
                {
                        Stack[SP-1].dummy->this = val = MakeValue(ValNUMBER);
                        val->number = w;
                }
        }
        else if (Stack[SP].dummy->this->type == ValSTRING)
        {
                /* logically negate a bit string */

                char * new, * old;      /* string pointers */

                CheckSign(Stack[SP].dummy->this);
                Stack[SP-1].dummy->this = val = MakeValue(ValSTRING);
```

```
			val->sign = Stack[SP].dummy->this->sign;

			old = Stack[SP].dummy->this->string;
			k = strlen(old);
			val->string = new = GetMemory(k + 1);

			while ((*old) != '\0')
			{
				if ((*old) == '0')
					(*new) = '1';
				else if ((*old) == '1')
					(*new) = '0';
				else if ((*old) == ANYBIT)
					(*new) = BADBIT;
				else if ((*old) == BADBIT)
					(*new) = ANYBIT;
				else
				{
					error = YES;
					break;
				}
				new ++;
				old ++;
			}
			(*new) = '\0';		/* terminate string */
		}
		else
			error = YES;

		if (error)
		{
			PrintLine();
			fprintf(stderr, "bad value in 'not': ");
			PrintValue(stderr, Stack[SP].dummy, YES);
			fprintf(stderr, "\n");
			ExecAbort();
		}

		PopStack();				/* kill right side */
		break;

	/* null */

	case (OpNULL):
		PushStack();
		break;

	/* number */

	case (OpNUMBER):
		PushStack();
		Stack[SP].dummy->this = val = MakeValue(ValNUMBER);
		val->number = par->number;
		break;

	/* parameter list */

	case (OpPAR):
		ExecParse(par->one);			/* left recursive side */
		ExecParse(par->two);			/* this parameter (right) */
		break;

	/* addition ("+") */
	/* addition (set concatentation) */
	/* addition (string concatentation) */

	case (OpPLUS):
		PushStack();				/* our result */
		ExecParse(par->one);			/* left side */
		ExecParse(par->two);			/* right side */
```

```

    error = NO;                      /* assume no errors */

    if ((Stack[SP-1].dummy->this == NULL)
    || (Stack[SP].dummy->this == NULL))
    {
            error = YES;
    }
    else if ((Stack[SP-1].dummy->this->type == ValNUMBER)
    && (Stack[SP].dummy->this->type == ValNUMBER))
    {
            x = Stack[SP-1].dummy->this->number;    /* left side */
            y = Stack[SP].dummy->this->number;      /* right side */
            w = x + y;

            Stack[SP-2].dummy->this = val = MakeValue(ValNUMBER);
            val->number = w;
    }
    else if ((Stack[SP-1].dummy->this->type == ValSET)
    && (Stack[SP].dummy->this->type == ValSET)
    && (Stack[SP-1].dummy->this->sign == Stack[SP].dummy->this->sign))
    {
            ValueThing * head;      /* head (left) set */
            ValueThing * tail;      /* tail (right) set */

            /* a little late, but check sign anyway */

            CheckSign(Stack[SP-1].dummy->this);
            CheckSign(Stack[SP].dummy->this);

            /* get link to right side */

            MakeDynamic(SP);
            tail = Stack[SP].dummy->this->next;
            Stack[SP].dummy->this->next = NULL;     /* kill link */
            Stack[SP].free = YES;

            /* attach link to existing left side */

            MakeDynamic(SP-1);
            head = Stack[SP-1].dummy->this;
            while (head->next != NULL)
                    head = head->next;
            head->next = tail;
            head = Stack[SP-1].dummy->this;
            Stack[SP-1].free = NO;

            /* now put result back onto stack */

            Stack[SP-2].dummy->this = head;
    }
    else if ((Stack[SP-1].dummy->this->type == ValSTRING)
    && (Stack[SP].dummy->this->type == ValSTRING)
    && (Stack[SP-1].dummy->this->sign == Stack[SP].dummy->this->sign))
    {
            char * cp;              /* a string pointer */

            /* a little late, but check sign anyway */

            CheckSign(Stack[SP-1].dummy->this);
            CheckSign(Stack[SP].dummy->this);

            /* get enough memory, and copy strings */

            j = strlen(Stack[SP-1].dummy->this->string);
            k = strlen(Stack[SP].dummy->this->string);

            Stack[SP-2].dummy->this = val = MakeValue(ValSTRING);
            val->sign = Stack[SP].dummy->this->sign;
            val->string = cp = GetMemory(j + k + 1);
```

```
			strcpy(cp, Stack[SP-1].dummy->this->string);
			strcat((cp + j), Stack[SP].dummy->this->string);
		}
		else
			error = YES;

		if (error)
		{
			PrintLine();
			fprintf(stderr, "bad values in '+': ");
			PrintValue(stderr, Stack[SP-1].dummy, YES);
			fprintf(stderr, " and ");
			PrintValue(stderr, Stack[SP].dummy, YES);
			fprintf(stderr, "\n");
			ExecAbort();
		}

		PopStack();				/* kill right side */
		PopStack();				/* kill left side */
		break;

	/* power */

	case (OpPOWER):
		/* convert power operator into a function call */
		/* can't do this in YACC: user may re-define "pow" symbol */

		PushStack();				/* our result */
		ExecParse(par->one);			/* left side */
		ExecParse(par->two);			/* right side */
		oldframe = FP;				/* save frame pointer */
		FP = SP - 2;				/* create fake frame */
		PreMath(NULL, pow, "power operator");	/* math routine */
		FP = oldframe;				/* restore frame pointer */
		PopStack();				/* kill right side */
		PopStack();				/* kill left side */
		break;

	/* repeat */

	case (OpREPEAT):
		do
		{
			/* execute statements */

			ExecParse(par->one);

			/* generate test condition */

			ExecParse(par->two);

			error = NO;				/* assume no errors */

			if ((Stack[SP].dummy->this == NULL)
			|| (Stack[SP].dummy->this->type != ValNUMBER))
			{
				error = YES;
			}
			else
			{
				x = Stack[SP].dummy->this->number;

				/* check test condition */

				if ((x == FALSE) || (x == TRUE))
				{
					/* legal value, but do nothing */
				}
				else
```

```
                                error = YES;
                }

                if (error)
                {
                        PrintLine();
                        fprintf(stderr, "bad condition in 'repeat' statement: ");
                        PrintValue(stderr, Stack[SP].dummy, YES);
                        fprintf(stderr, "\n");
                        ExecAbort();
                }

                PopStack();             /* kill test condition */
        }
        while (x == FALSE);
        break;

/* return from function */

case (OpRETURN):
        longjmp(ReturnJump.env, YES);
        break;

/* finished set */

case (OpSET):
        PushStack();
        Stack[SP].dummy->this = val = MakeValue(V[illegible]);
        val->sign = 1;

        /* generate element concatenations */

        if (par->one != NULL)
        {
                ExecParse(par->one);
                MakeDynamic(SP);
                val->next = Stack[SP].dummy->this;      /* link */
                Stack[SP].free = NO;
                PopStack();
        }
        break;

/* statement */

case (OpSTMT):
        oldstack = SP;                  /* must return to this level */

        /* do any preceding (left recursive) statements */

        ExecParse(par->one);

        /* do this statement */

        if (par->two != NULL)
        {
                ExecParse(par->two);

                /* anything to print? */

                if (SP == (oldstack + 1))
                {
                        if (par->two->type == OpASSIGN)
                        {
                                /* throw away assigned value */
                        }
                        else if ((Stack[SP].dummy->this == NULL)
                        && ((par->two->type == OpFUNCTION)
                        || ((par->two->type == OpNAME)
                        && (par->two->symbol->type == SymFUNCTION))))
                        {
```

```
                                        /* throw away NULL function result */
                                }
                                else
                                {
                                        PrintValue(stdout, Stack[SP].dummy, YES);
                                        fprintf(stdout, "\n");
                                }
                                PopStack();
                        }
                }

                /* is stack back to original level? */

                if (SP != oldstack)
                {
                        PrintLine();
                        fprintf(stderr,
                                "internal error: OpSTMT stack pointer is %d, not %d\n",
                                SP, oldstack);
                        ExecAbort();
                }
                break;

        /* string */

        case (OpSTRING):
                PushStack();
                Stack[SP].dummy->this = val = MakeValue(ValSTRING);
                val->sign = 1;
                val->string = CopyString(par->string);  /* must be "free" */
                break;

        /* while */

        case (OpWHILE):
                do
                {
                        /* generate test condition */

                        ExecParse(par->one);

                        error = NO;                     /* assume no errors */

                        if ((Stack[SP].dummy->this == NULL)
                        || (Stack[SP].dummy->this->type != ValNUMBER))
                        {
                                error = YES;
                        }
                        else
                        {
                                x = Stack[SP].dummy->this->number;

                                /* check test condition */

                                if (x == TRUE)
                                {
                                        ExecParse(par->two);
                                }
                                else if (x == FALSE)
                                {
                                        /* legal value, but do nothing */
                                }
                                else
                                        error = YES;
                        }

                        if (error)
                        {
                                PrintLine();
                                fprintf(stderr, "bad condition in 'while' statement: ");
```

```
                                PrintValue(stderr, Stack[SP].dummy, YES);
                                fprintf(stderr, "\n");
                                ExecAbort();
                        }

                        PopStack();             /* kill test condition */
                }
                while (x == TRUE);
                break;

/* default case for illegal or undefined operators */

default:
                PrintLine();
                fprintf(stderr, "internal error: ExecParse parse type = %d\n",
                        par->type);
                ExecAbort();
                break;

/* end of big, big switch statement */

        }

        /* restore file line number */

        LineNumber = oldline;
}


/*
MakeDynamic()

Make a stack entry dynamic.  That is, if it is not marked as "free", then
replace it with a "free" copy.
*/

MakeDynamic(n)
        int n;                          /* stack entry number, usually SP */
{
        if ((n < 0) || (n > SP))
        {
                PrintLine();
                fprintf(stderr,
                        "internal error: MakeDynamic has n = %d and SP = %d\n",
                        n, SP);
                ExecAbort();
        }

        if (Stack[n].free != YES)
        {
                Stack[n].dummy->this = CopyValue(Stack[n].dummy->this);
                Stack[n].free = YES;
                Stack[n].owner = NULL;
        }
}


/*
PopStack()

Pop one value off the stack.  If the stack is marked as "free" (dynamic), then
its memory will be released.
*/

PopStack()
{
        CheckStack();                   /* of course */

        /* free the value? */

```

```
        if (Stack[SP].free == YES)
        {
                FreeValue(Stack[SP].dummy->this);
        }

        /* clear the stack entry, just to be careful */

        Stack[SP].dummy->this = NULL;
        Stack[SP].free = YES;
        Stack[SP].owner = NULL;

        /* decrement stack pointer, and check it again */

        SP --;
        CheckStack();
}


/*
PushStack()

Push a NULL value onto the stack.  A NULL value has the attributes of being
dynamically allocated ("free") and no owner.  The stack field "dummy" points
to a dummy value structure that ends in a null pointer (dummy->this is NULL).
The caller should attach his value to the end of the dummy structure.

(All of the dummy stuff is necessary so that function local variables can be
assigned.)
*/

PushStack()
{
        /* increment the stack pointer, and check it */

        SP ++;
        CheckStack();

        /* put a dummy value structure onto the stack */

        if (Stack[SP].dummy == NULL)
        {
                Stack[SP].dummy = MakeValue(ValDUMMY);
        }
        Stack[SP].dummy->this = NULL;
        Stack[SP].free = YES;
        Stack[SP].owner = NULL;
}
```

```
/*

faglo.c -- Global Variables


Keith Fenske
Department of Computing Science
The University of Alberta
Edmonton, Alberta, Canada
T6G 2H1

December 1987

Copyright (c) 1987 by Keith Fenske.  All rights reserved.


This file defines the global variables declared by "fainc.h".  A global
variable is any variable shared by more than one module.  They are collected
here so that you don't have to go looking for initial values.

*/

#include "fainc.h"                          /* our standard includes */

        int FP = 0;                                 /* function frame pointer */
        SymbolThing * GlobalHead = NULL;            /* global symbol head */
        SymbolThing * GlobalTail = NULL;            /* global symbol tail */
        int LineNumber = -0999;                     /* file line number */
        SymbolThing * LocalHead = NULL;             /* local symbol head */
        SymbolThing * LocalTail = NULL;             /* local symbol tail */
        ParseThing * ParseTree = NULL;              /* YACC parse tree */
        int SP = 0;                                 /* stack pointer (index) */
        StackThing Stack[STACKSIZE+1];              /* execution stack */
        char * UserArg = "-p";                      /* argument for classifier */
        char * UserFile = "robert";                 /* name of user classifier */
        int UserTrace = NO;                         /* trace flag for classifier */
```

```
/*

fainc.h -- Include Standard Definitions


Keith Fenske
Department of Computing Science
The University of Alberta
Edmonton, Alberta, Canada
T6G 2H1

December 1987

Copyright (c) 1987 by Keith Fenske.  All rights reserved.


This file is #include'd by the other modules to declare:

        - values for common symbolic names
        - return value types for functions
        - data types for global variables

All global variables are defined in the "faglo.c" module.

*/


/* other includes */

#include <stdio.h>                  /* standard I/O */


/* definitions */

#define ANYBIT '#'                  /* "don't care" character in a bit string */
#define BADBIT '?'                  /* illegal character in a bit string */
#define CMPEQ 0                     /* compare: left equals right */
#define CMPGT 1                     /* compare: left greater than right */
#define CMPLT 2                     /* compare: left less than right */
#define CMPNE 3                     /* compare: left, right not comparable */
#define FALSE 0.0                   /* user-level logical false */
#define FORMAT "%.15g"              /* (f)printf format for numbers */
#define MAXSTRING 999               /* maximum bytes in a string */
#define NO 0                        /* "C" level logical false */
#define NUMBER double               /* user-level numeric type */
#define ONE 1.0                     /* user-level value of number one */
#define QUOTE '"'                   /* character for printing a string */
#define STACKSIZE 999               /* size of execution stack */
#define TRUE 1.0                    /* user-level logical true */
#define YES 1                       /* "C" level logical true */
#define ZERO 0.0                    /* user-level value of number zero */


/*
For each complete statement, a parse tree is built and then executed.  The
contents of a parse node vary depending upon the operator, which is some OpXXX
number.
*/

#define OpAND 101
#define OpASSIGN 102
#define OpCONCAT 103
#define OpDIV 104
#define OpEQ 105
#define OpEQP 106
#define OpFOR 107
#define OpFUNCTION 108
#define OpGE 109
#define OpGT 110
#define OpIF 111
```

```
#define OpINDEX 112
#define OpLE 113
#define OpLT 114
#define OpMINUS 115
#define OpMOD 116
#define OpNAME 117
#define OpNE 118
#define OpNEGATE 119
#define OpNEP 120
#define OpNOT 121
#define OpNULL 122
#define OpNUMBER 123
#define OpOR 124
#define OpPAR 125
#define OpPLUS 126
#define OpPOWER 127
#define OpREPEAT 128
#define OpRETURN 129
#define OpSET 130
#define OpSLASH 131
#define OpSTAR 132
#define OpSTMT 133
#define OpSTRING 134
#define OpWHILE 135

typedef struct ParseThing {
        struct ParseThing * one;        /* first part, if not NULL */
        struct ParseThing * two;        /* second part, if not NULL */
        struct ParseThing * three;      /* third part, if not NULL */
        struct ParseThing * four;       /* fourth part, if not NULL */
        int count;                      /* parameter count */
        int line;                       /* file line number */
        NUMBER number;                  /* if a number */
        char * string;                  /* if a string */
        struct SymbolThing * symbol;    /* if a name */
        int type;                       /* parse type: OpXXX */
} ParseThing;


/*
Expressions are evaluated on a stack.  To keep the stack entries consistent,
each entry points to a structure which contains the actual value.  Additional
information is kept on whether the stack values were created dynamically (and
are free to be assigned to a variable), or if the values are owned by a variable
that can be assigned a new value (used to simplify the assignment grammar).
*/

typedef struct StackThing {
        struct ValueThing * dummy;      /* pointer to dummy value structure */
        int free;                       /* non-zero if this value is "free" */
        struct ValueThing * owner;      /* owning ValueThing, if not NULL */
} StackThing;


/*
There are only two types of symbols: functions and variables.  Variables can be
global or local.  Global variables have exactly one value, which is pointed to
by the symbol table (via a dummy value structure).  Local variables are stack
offsets relative to a function call (which allows recursion).
*/

#define SpeABS 401                      /* pre-defined absolute function */
#define SpeACOS 402                     /* pre-defined arc cosine function */
#define SpeASIN 403                     /* pre-defined arc sine function */
#define SpeATAN 404                     /* pre-defined arc tangent function */
#define SpeATAN2 405                    /* pre-defined arc tangent function */
#define SpeCBRT 406                     /* pre-defined cube root function */
#define SpeCLOSE 407                    /* classifier close function */
#define SpeCOS 408                      /* pre-defined cosine function */
#define SpeEXIT 409                     /* pre-defined exit function */
```

```
#define SpeEXP 410                                /* pre-defined exponential function */
#define SpeFLAGMESS 411                           /* classifier flagmess function */
#define SpeFLAGRULE 412                           /* classifier flagrule function */
#define SpeLOAD 413                               /* pre-defined load function */
#define SpeLOG 414                                /* pre-defined logarithm function */
#define SpeLOG10 415                              /* pre-defined logarithm function */
#define SpeOPEN 416                               /* classifier open function */
#define SpePACK 417                               /* pre-defined pack function */
#define SpePOW 418                                /* pre-defined power function */
#define SpePRINTF 419                             /* pre-defined printf function */
#define SpeRANDOM 420                             /* pre-defined random function */
#define SpeRECEIVE 421                            /* classifier receive function */
#define SpeROUND 422                              /* pre-defined round function */
#define SpeSAVE 423                               /* pre-defined save function */
#define SpeSCANF 424                              /* pre-defined scanf function */
#define SpeSEND 425                               /* classifier send function */
#define SpeSIGN 426                               /* pre-defined sign function */
#define SpeSIN 427                                /* pre-defined sine function */
#define SpeSIZE 428                               /* pre-defined size function */
#define SpeSPRINTF 429                            /* pre-defined sprintf function */
#define SpeSQRT 430                               /* pre-defined square root function */
#define SpeSSCANF 431                             /* pre-defined sscanf function */
#define SpeSTOP 432                               /* pre-defined stop function */
#define SpeSYSTEM 433                             /* pre-defined system function */
#define SpeTAN 434                                /* pre-defined tangent function */
#define SpeTRUNC 435                              /* pre-defined truncate function */
#define SpeTYPE 436                               /* pre-defined type function */
#define SpeUNPACK 437                             /* pre-defined unpack function */
#define SpeVALUE 438                              /* pre-defined value function */
#define SpeWRITE 439                              /* pre-defined write function */

#define SymFUNCTION 501                           /* type if a function */
#define SymGLOBAL 502                             /* type if a global variable */
#define SymLOCAL 503                              /* type if a local variable */

typedef struct SymbolThing {
        int count;                                /* # of parameters, if a function */
        struct ValueThing * dummy;                /* dummy value if global */
        int free;                                 /* for local variables (parameters):
                                                     NO if passed by address,
                                                     YES if copy by value */
        struct SymbolThing * local;               /* local symbols if a function */
        char * name;                              /* name string */
        struct SymbolThing * next;                /* next symbol, or NULL */
        int offset;                               /* parameter list offset if local */
        struct ParseThing * parse;                /* parse tree if a function */
        int special;                              /* special processing code */
        int type;                                 /* symbol type: SymXXX */
} SymbolThing;


/*
The user's data is stored in linked "value" structures.  Basic data items are
numbers, strings, and the NULL value.  Composite items are sets of the basic
items.  A dummy value is inserted between variable entries in the symbol table
and the actual value structure so that the assignment operator doesn't need
separate productions for the lefthand and righthand sides.  (See the "owner"
field in "StackThing".)

NULL values are assumed when a pointer to a required value structure is NULL.
*/

#define ValDUMMY 701                              /* type if a dummy value structure */
#define ValELEMENT 702                            /* type if a set element */
#define ValNUMBER 703                             /* type if a number */
#define ValSET 704                                /* type if a set */
#define ValSTRING 705                             /* type if a string */

typedef struct ValueThing {
        struct ValueThing * next;                 /* first element if a set */
```

```
                                            /* next element if an element */
        NUMBER number;                      /* value if a number */
        int sign;                           /* +1 or -1 if set or string */
        char * string;                      /* value if a string */
        struct ValueThing * this;           /* value if dummy or element */
        int type;                           /* value type: ValXXX */
} ValueThing;


/* global variables */

        extern int FP;                              /* function frame pointer */
        extern SymbolThing * GlobalHead;            /* global symbol head */
        extern SymbolThing * GlobalTail;            /* global symbol tail */
        extern int LineNumber;                      /* file line number */
        extern SymbolThing * LocalHead;             /* local symbol head */
        extern SymbolThing * LocalTail;             /* local symbol tail */
        extern ParseThing * ParseTree;              /* YACC parse tree */
        extern int SP;                              /* stack pointer (index) */
        extern StackThing Stack[];                  /* execution stack */
        extern char * UserArg;                      /* argument for classifier */
        extern char * UserFile;                     /* name of user classifier */
        extern int UserTrace;                       /* trace flag for classifier */
        extern FILE * yyin;                         /* LEX input file pointer */


/* functions */

        int CompareValue();                 /* compare value structures */
        char * CopyString();                /* dynamic string copy */
        ValueThing * CopyValue();           /* dynamic value copy */
        NUMBER FindNumber();                /* find number in a string */
        char * GetMemory();                 /* memory allocation */
        SymbolThing * GlobalAdd();          /* global symbol addition */
        SymbolThing * GlobalLook();         /* global symbol look-up */
        SymbolThing * LocalAdd();           /* local symbol addition */
        SymbolThing * LocalLook();          /* local symbol look-up */
        ParseThing * MakeParse();           /* create parse tree node */
        SymbolThing * MakeSymbol();         /* create symbol table entry */
        ValueThing * MakeValue();           /* create value structure */
        char * PackValue();                 /* pack a value into a string */
        char * SkipSpace();                 /* skip white space in a string */
        ValueThing * StringToValue();       /* parse string into value structure */
        ValueThing * UnpackString();        /* unpack string into a value */
        ValueThing * UnpackValue();         /* unpack value into a bigger value */
        char * UserReceive();               /* receive from user classifier */
```

```
%{

/*

falex.l -- Lexical Tokens


Keith Fenske
Department of Computing Science
The University of Alberta
Edmonton, Alberta, Canada
T6G 2H1

December 1987

Copyright (c) 1987 by Keith Fenske.  All rights reserved.


This file breaks up the input stream into tokens that can be handled by YACC.
Reserved words are recognized in any combination of upper and lower case
letters.  Some special characters are converted into the equivalent reserved
word.  (For example, "&" is the same as the "AND" token.)  Strings are scanned
manually so that "\" escape sequences can be recognized.  Comments go from a
"#" character to the end of the line (not in a string, of course!).

The relational operators (EQ, LT, etc) are grouped together as a single token.
This simple change reduced an early version of the YACC machine in "y.output"
from 2,300 lines to 1,200 lines.

(Special characters are listed in ASCII order.)

*/

#include "fainc.h"                      /* our standard includes */
#include "y.tab.h"                      /* YACC token definitions */

#define LEXEOF 0                        /* lexical end-of-file character */

        double atof();                  /* ASCII to floating double */

%}


%%


[aA][nN][dD]                    { return(TokAND); }
[bB][rR][eE][aA][kK]            { return(TokBREAK); }
[bB][yY]                        { return(TokBY); }
[dD][iI][vV]                    { return(TokDIV); }
[dD][oO]                        { return(TokDO); }
[eE][lL][iI][fF]                { return(TokELIF); }
[eE][lL][sS][eE]                { return(TokELSE); }
[eE][nN][dD]                    { return(TokEND); }
[eE][qQ]                        { yylval.count = OpEQ; return(TokRELOP); }
[eE][qQ][pP]                    { yylval.count = OpEQP; return(TokRELOP); }
[fF][oO][rR]                    { return(TokFOR); }
[fF][rR][oO][mM]                { return(TokFROM); }
[fF][uU][nN][cC][tT][iI][oO][nN] { return(TokFUNCTION); }
[gG][eE]                        { yylval.count = OpGE; return(TokRELOP); }
[gG][tT]                        { yylval.count = OpGT; return(TokRELOP); }
[iI][fF]                        { return(TokIF); }
[lL][eE]                        { yylval.count = OpLE; return(TokRELOP); }
[lL][tT]                        { yylval.count = OpLT; return(TokRELOP); }
[mM][oO][dD]                    { return(TokMOD); }
[nN][eE]                        { yylval.count = OpNE; return(TokRELOP); }
[nN][eE][pP]                    { yylval.count = OpNEP; return(TokRELOP); }
[nN][oO][tT]                    { return(TokNOT); }
[nN][uU][lL][lL]                { return(TokNULL); }
[oO][rR]                        { return(TokOR); }
```

```
[pP][rR][oO][cC][eE][dD][uU][rR][eE] { return(TokFUNCTION); }
[rR][eE][pP][eE][aA][tT]          { return(TokREPEAT); }
[rR][eE][tT][uU][rR][nN]          { return(TokRETURN); }
[tT][oO]                          { return(TokTO); }
[tT][hH][eE][nN]                  { return(TokTHEN); }
[uU][nN][tT][iI][lL]              { return(TokUNTIL); }
[wW][hH][iI][lL][eE]              { return(TokWHILE); }

[a-zA-Z_][a-zA-Z_0-9]*            {
                                  yylval.string = CopyString(yytext);
                                  return(TokNAME);
                                  }

(([0-9]+)|([0-9]+"."[0-9]*)|("."[0-9]+))([eE][-+]?[0-9]+)?      {
                                  yylval.number = atof(yytext);
                                  return(TokNUMBER);
                                  }

"!="                              { yylval.count = OpNE; return(TokRELOP); }
\"                                { LexString('"'); return(TokSTRING); }
"%"                               { return(TokMOD); }
"&"+                              { return(TokAND); }
"'"                               { LexString('\''); return(TokSTRING); }
"**"                              { return(TokPOWER); }
":="                              { return(TokASSIGN); }
"<="                              { yylval.count = OpLE; return(TokRELOP); }
"<>"                              { yylval.count = OpNE; return(TokRELOP); }
"<"                               { yylval.count = OpLT; return(TokRELOP); }
"=<"                              { yylval.count = OpLE; return(TokRELOP); }
"=>"                              { yylval.count = OpGE; return(TokRELOP); }
"="+                              { yylval.count = OpEQ; return(TokRELOP); }
">="                              { yylval.count = OpGE; return(TokRELOP); }
">"                               { yylval.count = OpGT; return(TokRELOP); }
"@"                               { return(TokPOWER); }
"^"                               { return(TokPOWER); }
"└"                               { LexString('└'); return(TokSTRING); }
"|"+                              { return(TokOR); }
"¬"                               { return(TokNOT); }

\n                                { /* newline */ LineNumber++; }
[ \t]+                            { /* ignore spaces and tabs */ }
"#".*                             { /* comment, do nothing */ }

.                                 { /* default */ return(yytext[0]); }


%%


/*
LexString()

Given a delimiting character, look for a string ending with this character, and
which may include the following escape sequences:

                \n for newline
                \t for tab
                \\ for \

or \ followed by the delimiting character.
*/

LexString(delim)
        char delim;                     /* delimiting character */
{
        char buffer[MAXSTRING+2];       /* string buffer */
        char c;                         /* some input character */
        int length;                     /* bytes in string buffer */

        length = 0;                     /* nothing in buffer */
```

```

        while (length <= MAXSTRING)
        {
                c = input();                /* get an input character */
                if (c == LEXEOF)            /* lexical end-of-file? */
                {
                        PrintLine();
                        fprintf(stderr, "end-of-file in quoted string (%c)\n",
                                delim);
                        unput(c);
                        break;
                }
                else if (c == '\n')         /* newline? */
                {
                        PrintLine();
                        fprintf(stderr, "newline in quoted string (%c)\n",
                                delim);
                        unput(c);
                        break;
                }
                else if (c == '\\')         /* escape sequence? */
                {
                        char d;             /* another input character */

                        d = input();
                        if (d == 'n')
                                buffer[length++] = '\n';
                        else if (d == 't')
                                buffer[length++] = '\t';
                        else if (d == '\\')
                                buffer[length++] = '\\';
                        else if (d == delim)
                                buffer[length++] = delim;
                        else
                        {
                                buffer[length++] = '\\';
                                unput(d);
                        }
                }
                else if (c == delim)        /* end of the string? */
                {
                        break;
                }
                else                        /* just a text character */
                {
                        buffer[length++] = c;
                }
        }
        buffer[length] = '\0';              /* put null at the end */

        if (length > MAXSTRING)
        {
                PrintLine();
                fprintf(stderr,
                        "string longer than %d characters, begins with '%.9s'\n",
                        MAXSTRING, buffer);
        }

        yylval.string = CopyString(buffer);
}


/*
PrintLine()

If input is coming from a file, print an error message on stderr with the file
line number.  Otherwise, if input is from a terminal, don't bother.
*/

PrintLine()
```

```
{
        if (LineNumber > 0)
                fprintf(stderr, "at line %d: ", LineNumber);
}
```

```
/*

famem.c -- Memory Allocation


Keith Fenske
Department of Computing Science
The University of Alberta
Edmonton, Alberta, Canada
T6G 2H1

December 1987

Copyright (c) 1987 by Keith Fenske.  All rights reserved.


These routines perform some of the low-level memory allocation functions.  The
standard "malloc" routine is called to handle the most trivial details.

*/

#include "fainc.h"                      /* our standard includes */


/*
CopyString()

Given a pointer to a string (which may be in static memory), allocate enough
memory for a new string, and copy the contents.  This is used to save names
and strings found by the LEX routines.
*/

char * CopyString(old)
        char * old;                     /* pointer to old string */
{
        char * new;                     /* pointer to new string */

        if (old == NULL)
                new = NULL;
        else
        {
                new = GetMemory(strlen(old) + 1);
                strcpy(new, old);
        }
        return(new);
}


/*
CopyValue()

Recursively copy a value structure into a new dynamically allocated structure.
The contents of the new structure are identical to the old, and no attempt is
made to check them.
*/

ValueThing * CopyValue(val)
        ValueThing * val;               /* a value structure pointer */
{
        ValueThing * new;               /* new value structure pointer */

        if (val == NULL)
                new = NULL;
        else
        {
                new = MakeValue(val->type);
                new->next = CopyValue(val->next);
                new->number = val->number;
                new->sign = val->sign;
                new->string = CopyString(val->string);
```

```
                new->this = CopyValue(val->this);
        }
        return(new);
}


/*
FreeParse()

Recursively free all memory allocated to a parse tree.  This is used when a
function is re-defined, or after a complete statement has been executed.
*/

FreeParse(par)
        ParseThing * par;                       /* a parse tree pointer */
{
        if (par != NULL)
        {
                FreeParse(par->one);
                FreeParse(par->two);
                FreeParse(par->three);
                FreeParse(par->four);
                FreeString(par->string);

                /* DO NOT FREE PAR->SYMBOL ! */

                free(par);
        }
}


/*
FreeString()

Free the memory allocated to a string.  The string must have been previously
allocated by some dynamic routine, such as CopyString().
*/

FreeString(string)
        char * string;                          /* a string pointer */
{
        if (string != NULL)
                free(string);
}


/*
FreeSymbol()

Recursively free all memory allocated to a symbol table.  This is used when a
global symbol is re-defined as a function, and the old local symbol table (if
any) must be de-allocated.  The caller must ensure that there are no stray
pointers to the freed memory.
*/

FreeSymbol(sym)
        SymbolThing * sym;                      /* a symbol table pointer */
{
        SymbolThing * old;                      /* old symbol table pointer */
        SymbolThing * new;                      /* new symbol table pointer */

        old = sym;
        while (old != NULL)
        {
                FreeValue(old->dummy);
                FreeSymbol(old->local);
                FreeString(old->name);
                FreeParse(old->parse);

                new = old->next;        /* don't use a freed pointer! */
```

```
			free(old);
			old = new;
		}
}


/*
FreeValue()

Recursively free all memory allocated to a value structure.  This is used when
a variable is assigned, or when an operator finishes with its operands.
*/

FreeValue(val)
	ValueThing * val;			/* a value structure pointer */
{
	ValueThing * old;			/* old value structure pointer */
	ValueThing * new;			/* new value structure pointer */

	old = val;
	while (old != NULL)
	{
		FreeString(old->string);
		FreeValue(old->this);

		new = old->next;		/* don't use a freed pointer! */
		free(old);
		old = new;
	}
}


/*
GetMemory()

Given a size in bytes, allocate enough memory to hold a thing that big.  If
this fails, print a nasty error message and abort.
*/

char * GetMemory(size)
	int size;				/* size in bytes */
{
	char * malloc();			/* must declare function type */
	char * new;				/* pointer to new string */

	new = malloc((unsigned) size);
	if (new == NULL)
	{
		PrintLine();
		fprintf(stderr, "GetMemory failed to allocate %d bytes\n",
			size);
		abort();			/* be nasty */
	}
	return(new);
}


/*
GlobalAdd()

Add a new symbol to the global symbol table.  A warning is issued if this name
is already on the global symbol table.
*/

SymbolThing * GlobalAdd(name, type)
	char * name;				/* some name string */
	int type;				/* symbol type: SymXXX */
{
	SymbolThing * sym;			/* symbol table pointer */

```

```
        sym = GlobalLook(name);
        if (sym != NULL)
        {
                PrintLine();
                fprintf(stderr, "warning: duplicate global symbol '%s'\n",
                        name);
        }
        sym = MakeSymbol(type);
        sym->name = name;
        if (GlobalTail != NULL)
                GlobalTail->next = sym;
        GlobalTail = sym;
        if (GlobalHead == NULL)
                GlobalHead = sym;

        return(sym);
}


/*
GlobalLook()

Given a name, look for this symbol in the global symbol table.  If an entry is
found, return the address.  Otherwise, return NULL.
*/

SymbolThing * GlobalLook(name)
        char * name;                    /* some name string */
{
        SymbolThing * sym;              /* symbol table pointer */

        sym = GlobalHead;
        while (sym != NULL)
        {
                if (strcmp(name, sym->name) == 0)
                        break;
                sym = sym->next;
        }
        return(sym);
}


/*
LocalAdd()

Add a new symbol to the local symbol table.  A warning is issued if this name
is already on the local symbol table.
*/

SymbolThing * LocalAdd(name)
        char * name;                    /* some name string */
{
        SymbolThing * sym;              /* symbol table pointer */

        sym = LocalLook(name);
        if (sym != NULL)
        {
                PrintLine();
                fprintf(stderr, "warning: duplicate local symbol '%s'\n",
                        name);
        }
        sym = MakeSymbol(SymLOCAL);
        sym->name = name;
        if (LocalTail != NULL)
                LocalTail->next = sym;
        LocalTail = sym;
        if (LocalHead == NULL)
                LocalHead = sym;

        return(sym);
```

```
}


/*
LocalLook()

Given a name, look for this symbol in the local symbol table.  If an entry is
found, return the address.  Otherwise, return NULL.
*/

SymbolThing * LocalLook(name)
        char * name;                    /* some name string */
{
        SymbolThing * sym;              /* symbol table pointer */

        sym = LocalHead;
        while (sym != NULL)
        {
                if (strcmp(name, sym->name) == 0)
                        break;
                sym = sym->next;
        }
        return(sym);
}



/*
MakeParse()

Allocate space for a new parse tree node, given the operator type and pointers
to the first four parse sub-trees.  All other fields are cleared; the caller
must insert the correct contents.
*/

ParseThing * MakeParse(type, one, two, three, four)
        int type;                       /* operator type: OpXXX */
        ParseThing * one;               /* first part */
        ParseThing * two;               /* second part */
        ParseThing * three;             /* third part */
        ParseThing * four;              /* fourth part */
{
        ParseThing * new;               /* new parse pointer */

        new = (ParseThing *) GetMemory(sizeof(ParseThing));

        new->one = one;
        new->two = two;
        new->three = three;
        new->four = four;
        new->count = 0;
        new->line = LineNumber;
        new->number = ZERO;
        new->string = NULL;
        new->symbol = NULL;
        new->type = type;

        return(new);
}


/*
MakeSymbol()

Allocate space for a new symbol table entry, given the symbol type.  All of the
fields are cleared, and the caller must insert the correct contents.
*/

SymbolThing * MakeSymbol(type)
        int type;                       /* symbol type: SymXXX */
```

```
{
        SymbolThing * new;              /* new symbol pointer */

        new = (SymbolThing *) GetMemory(sizeof(SymbolThing));

        new->count = 0;
        new->dummy = NULL;
        new->free = YES;
        new->local = NULL;
        new->name = NULL;
        new->next = NULL;
        new->offset = 0;
        new->parse = NULL;
        new->special = 0;
        new->type = type;

        return(new);
}


/*
MakeValue()

Allocate space for a new value structure, given the value type.  All of the
fields are cleared, and the caller must insert the correct contents.
*/

ValueThing * MakeValue(type)
        int type;                       /* value type: ValXXX */
{
        ValueThing * new;               /* new value pointer */

        new = (ValueThing *) GetMemory(sizeof(ValueThing));

        new->next = NULL;
        new->number = ZERO;
        new->sign = 0;
        new->string = NULL;
        new->this = NULL;
        new->type = type;

        return(new);
}
```

```
/*

fapre.c -- Pre-Defined Functions


Keith Fenske
Department of Computing Science
The University of Alberta
Edmonton, Alberta, Canada
T6G 2H1

December 1987

Copyright (c) 1987 by Keith Fenske.  All rights reserved.


These are the pre-defined language functions.  Many are just calls to support
subroutines in the "fasub.c" module.  (Classifier pre-defined functions are in
the "fause.c" module.)

*/

#include "fainc.h"                          /* our standard includes */
#include <math.h>                           /* math routines */
#include <setjmp.h>                         /* system long jump */
#include <signal.h>                         /* system signals */
#include <varargs.h>                        /* system variable arguments */

        jmp_buf MathJump;                   /* for errors in math routines */


/*
Define a type for the parameters to the formatted I/O routines ("printf",
"scanf", etc).  Each parameter can be a number pointer, a number value, or a
string pointer.  We cheat here and assume that parameters are of two types:
pointers and numbers.  All pointers are assumed to have the same size.

Note:  There is a problem on VAX computers.  Parameters are pushed onto a
common stack.  For an arbitrary union to be acceptable for all parameters,
members of the union would have to have the same size.  Otherwise, if numbers
are double-precision, and "double" is twice the size of a pointer, then for:

                        printf("%s %f", "hello", 23);

the following would get put on the machine stack for "hello" and 23:

        +--------------+--------------+--------------+--------------+
        | char pointer | garbage      | double                      |
        +--------------+--------------+--------------+--------------+

when "printf" is expecting:

        +--------------+--------------+--------------+
        | char pointer | double                      |
        +--------------+--------------+--------------+
*/

#define FioNUMBER 801                       /* type if a number */
#define FioPOINTER 802                      /* type if a pointer */

        typedef struct {
                NUMBER number;              /* number value */
                char * pointer;             /* number or string pointer */
                int type;                   /* type */
        } FioThing;

        struct { NUMBER dummy[10]; } va_alist;
                                            /* 10-variable argument list */
                                            /* don't change the name */
        va_list va_pvar;                    /* pointer to variable list */
```

```
#define va_put(p) if (p.type == FioNUMBER) \
                va_arg(va_pvar, NUMBER) = p.number; \
        else \
                va_arg(va_pvar, char *) = p.pointer;


/*
FioCheck()

Check that a parameter on the stack is legal for use with the formatted I/O
subroutines ("printf", "scanf", etc).  Return the appropriate value for use
with the real system subroutine.
*/

FioThing FioCheck(n, string, assign, name)
        int n;                          /* offset from frame pointer */
        int string;                     /* YES if must be a string */
        int assign;                     /* YES if must be assignable */
        char * name;                    /* name of pre-defined function */
{
        FioThing result;                /* our result thingy */

        result.number = ZERO;           /* default to zero */
        result.pointer = NULL;          /* default to nothing */
        result.type = FioPOINTER;       /* assume a pointer */

        if ((FP + n) > SP)
        {
                /* this parameter is missing in function call */

                if (string)
                {
                        PrintLine();
                        fprintf(stderr,
                                "%s failed: string parameter #%d is missing\n",
                                name, n);
                        ExecAbort();
                }
                else
                        result.pointer = NULL;
        }
        else if (Stack[FP+n].dummy->this == NULL)
        {
                /* explicit NULL parameters are illegal */
                /* happens when you use an undefined variable name */

                PrintLine();
                fprintf(stderr, "%s failed: parameter #%d is NULL\n", name, n);
                ExecAbort();
        }
        else if (assign && (Stack[FP+n].owner == NULL))
        {
                PrintLine();
                fprintf(stderr,
                        "%s failed: parameter #%d can not be assigned: ",
                        name, n);
                PrintValue(stderr, Stack[FP+n].dummy, YES);
                fprintf(stderr, "\n");
                ExecAbort();
        }
        else if (Stack[FP+n].dummy->this->type == ValNUMBER)
        {
                if (string)
                {
                        PrintLine();
                        fprintf(stderr,
                                "%s failed: parameter #%d must be a string: ",
                                name, n);
                        PrintValue(stderr, Stack[FP+n].dummy, YES);
```

```
                        fprintf(stderr, "\n");
                        ExecAbort();
                }
                else if (assign)
                {
                        Stack[FP+n].dummy->this->number = ZERO;
                        result.pointer = (char *) &
                                Stack[FP+n].dummy->this->number;
                }
                else
                {
                        result.number = Stack[FP+n].dummy->this->number;
                        result.type = FioNUMBER;
                }
        }
        else if (Stack[FP+n].dummy->this->type == ValSTRING)
        {
                if (assign)
                {
                        char * cp;
                        int i;

                        /* free old string */

                        FreeString(Stack[FP+n].dummy->this->string);

                        /* allocate and clear a new string */
                        /* clearing it makes %c input safe */

                        cp = GetMemory(MAXSTRING + 1);
                        for (i = 0 ; i <= MAXSTRING ; i ++)
                                (*(cp+i)) = '\0';

                        /* link back to original value structure */

                        Stack[FP+n].dummy->this->string = cp;
                }
                result.pointer = Stack[FP+n].dummy->this->string;
        }
        else
        {
                PrintLine();
                fprintf(stderr,
                        "%s failed: parameter #%d must be a number or string: ",
                        name, n);
                PrintValue(stderr, Stack[FP+n].dummy, YES);
                fprintf(stderr, "\n");
                ExecAbort();
        }
        return(result);
}


/*
PreAbs()

Pre-defined function to return the absolute value of an expression.
*/

PreAbs(par)
        ParseThing * par;               /* function call in parse tree */
{
        if (Stack[FP+1].dummy->this == NULL)
        {
                /* abs(NULL) is just NULL */
        }
        else
        {
                CheckSign(Stack[FP+1].dummy->this);

```

```
		switch (Stack[FP+1].dummy->this->type)
		{
		case (ValNUMBER):
			Stack[FP].dummy->this = MakeValue(ValNUMBER);
			Stack[FP].dummy->this->number =
				fabs(Stack[FP+1].dummy->this->number);
			break;
		case (ValSET):
		case (ValSTRING):
			Stack[FP].dummy->this =
				CopyValue(Stack[FP+1].dummy->this);
			Stack[FP].dummy->this->sign = 1;
			break;
		default:
			PrintLine();
			fprintf(stderr,
				"internal error: PreAbs value type = %d\n",
				Stack[FP+1].dummy->this->type);
			ExecAbort();
			break;
		}
	}
}


/*
PreDefine()

Define the pre-defined symbols.  Try to be alphabetical here.

For functions, "sym->count" is the minimum number of parameters.  If negative,
then any number of parameters is acceptable, and the pre-defined function must
be smart enough to use only those values that have been actually pushed onto
the stack.
*/

PreDefine()
{
	SymbolThing * sym;		/* a symbol table pointer */

	sym = GlobalAdd("abs", SymFUNCTION);
	sym->count = 1;
	sym->special = SpeABS;

	sym = GlobalAdd("acos", SymFUNCTION);
	sym->count = 2;
	sym->special = SpeACOS;

	sym = GlobalAdd("asin", SymFUNCTION);
	sym->count = 2;
	sym->special = SpeASIN;

	sym = GlobalAdd("atan", SymFUNCTION);
	sym->count = 2;
	sym->special = SpeATAN;

	sym = GlobalAdd("atan2", SymFUNCTION);
	sym->count = 2;
	sym->special = SpeATAN2;

	sym = GlobalAdd("cbrt", SymFUNCTION);
	sym->count = 2;
	sym->special = SpeCBRT;

	sym = GlobalAdd("cos", SymFUNCTION);
	sym->count = 2;
	sym->special = SpeCOS;

	sym = GlobalAdd("exit", SymFUNCTION);
	sym->count = 0;
```

```
	sym->special = SpeEXIT;

	sym = GlobalAdd("exp", SymFUNCTION);
	sym->count = 2;
	sym->special = SpeEXP;

	sym = GlobalAdd("false", SymGLOBAL);
	sym->dummy = MakeValue(ValDUMMY);
	sym->dummy->this = MakeValue(ValNUMBER);
	sym->dummy->this->number = FALSE;

	sym = GlobalAdd("load", SymFUNCTION);
	sym->count = 1;
	sym->special = SpeLOAD;

	sym = GlobalAdd("log", SymFUNCTION);
	sym->count = 2;
	sym->special = SpeLOG;

	sym = GlobalAdd("log10", SymFUNCTION);
	sym->count = 2;
	sym->special = SpeLOG10;

	sym = GlobalAdd("pack", SymFUNCTION);
	sym->count = 2;
	sym->special = SpePACK;

	sym = GlobalAdd("pi", SymGLOBAL);
	sym->dummy = MakeValue(ValDUMMY);
	sym->dummy->this = MakeValue(ValNUMBER);
	sym->dummy->this->number = 3.14159265358979323846;

	sym = GlobalAdd("pow", SymFUNCTION);
	sym->count = 2;
	sym->special = SpePOW;

	sym = GlobalAdd("printf", SymFUNCTION);
	sym->count = 1;
	sym->special = SpePRINTF;

	sym = GlobalAdd("quit", SymFUNCTION);
	sym->count = 0;
	sym->special = SpeEXIT;

	sym = GlobalAdd("random", SymFUNCTION);
	sym->count = 1;
	sym->special = SpeRANDOM;
	srandom(getpid());			/* initialize random seed */

	sym = GlobalAdd("round", SymFUNCTION);
	sym->count = 2;
	sym->special = SpeROUND;

	sym = GlobalAdd("save", SymFUNCTION);
	sym->count = 1;
	sym->special = SpeSAVE;

	sym = GlobalAdd("scanf", SymFUNCTION);
	sym->count = 1;
	sym->special = SpeSCANF;

	sym = GlobalAdd("sign", SymFUNCTION);
	sym->count = 1;
	sym->special = SpeSIGN;

	sym = GlobalAdd("sin", SymFUNCTION);
	sym->count = 2;
	sym->special = SpeSIN;

	sym = GlobalAdd("size", SymFUNCTION);
```

```
        sym->count = 1;
        sym->special = SpeSIZE;

        sym = GlobalAdd("sprintf", SymFUNCTION);
        sym->count = 2;
        sym->special = SpeSPRINTF;

        sym = GlobalAdd("sqrt", SymFUNCTION);
        sym->count = 2;
        sym->special = SpeSQRT;

        sym = GlobalAdd("sscanf", SymFUNCTION);
        sym->count = 2;
        sym->special = SpeSSCANF;

        sym = GlobalAdd("stop", SymFUNCTION);
        sym->count = 0;
        sym->special = SpeSTOP;

        sym = GlobalAdd("system", SymFUNCTION);
        sym->count = 1;
        sym->special = SpeSYSTEM;

        sym = GlobalAdd("tan", SymFUNCTION);
        sym->count = 2;
        sym->special = SpeTAN;

        sym = GlobalAdd("true", SymGLOBAL);
        sym->dummy = MakeValue(ValDUMMY);
        sym->dummy->this = MakeValue(ValNUMBER);
        sym->dummy->this->number = TRUE;

        sym = GlobalAdd("trunc", SymFUNCTION);
        sym->count = 2;
        sym->special = SpeTRUNC;

        sym = GlobalAdd("type", SymFUNCTION);
        sym->count = 2;
        sym->special = SpeTYPE;

        sym = GlobalAdd("unpack", SymFUNCTION);
        sym->count = 2;
        sym->special = SpeUNPACK;

        sym = GlobalAdd("value", SymFUNCTION);
        sym->count = 1;
        sym->special = SpeVALUE;

        sym = GlobalAdd("write", SymFUNCTION);
        sym->count = -1;
        sym->special = SpeWRITE;
}


/*
PreExit()

Pre-defined function to exit (quit) from this classifier interface system, and
to return to the UNIX command level.
*/

PreExit(par)
        ParseThing * par;               /* function call in parse tree */
{
        fprintf(stderr, "\n");
        PrintLine();
        fprintf(stderr, "exit() called; returning to UNIX\n");

        /* close the pipe to the user classifier system (if any) */

```

```
        UserClose();

        exit(1);
}


/*
PreLoad()

Pre-defined function to "load" (read and execute) the statements or definitions
in a file.
*/

PreLoad(par)
        ParseThing * par;               /* function call in parse tree */
{
        if ((Stack[FP+1].dummy->this != NULL)
        && (Stack[FP+1].dummy->this->type == ValSTRING))
        {
                ExecFile(Stack[FP+1].dummy->this->string);
        }
        else
        {
                PrintLine();
                fprintf(stderr,
                        "load failed: parameter must be a file name string: ");
                PrintValue(stderr, Stack[FP+1].dummy, YES);
                fprintf(stderr, "\n");
                ExecAbort();
        }
}


/*
PreMath()

Pre-defined math functions.  The caller must give us the address of a real
system math routine (like "sqrt") and a character string with the routine's
name.  We pass up to two number parameters, return one number result, and trap
errors.
*/

PreMath(par, func, name)
        ParseThing * par;               /* function call in parse tree */
        double (* func)();              /* pointer to function returns double */
        char * name;                    /* function name */
{
        int error;                      /* YES means bad values */
        extern int PreMathTrap();       /* trap handler for error signal */
        int (* status)();               /* status of "signal" function */
        NUMBER w, x, y;                 /* numbers */

        error = NO;                     /* assume no errors */

        /* get first number parameter */

        if ((Stack[FP+1].dummy->this != NULL)
        && (Stack[FP+1].dummy->this->type == ValNUMBER))
        {
                x = Stack[FP+1].dummy->this->number;
        }
        else
                error = YES;

        /* get second number parameter (if any) */

        if (Stack[FP+2].dummy->this == NULL)
        {
                y = ZERO;
        }
```

```
        else if (Stack[FP+2].dummy->this->type == ValNUMBER)
        {
                y = Stack[FP+2].dummy->this->number;
        }
        else
                error = YES;

        /* add protection and call the math routine */

        if (!error)
        {
                /* set long jump for PreMathTrap abort */

                if (setjmp(MathJump) == 0)
                {
                        /* trap signal for math errors */

                        status = signal(SIGILL, PreMathTrap);
                        if (status == BADSIG)
                        {
                                perror("PreMath signal trap failed");
                                exit(-1);
                        }

                        /* call the math routine */

                        w = (*func)(x, y);

                        /* return the result (if we get this far) */

                        Stack[FP].dummy->this = MakeValue(ValNUMBER);
                        Stack[FP].dummy->this->number = w;
                }
                else
                {
                        /* must be PreMathTrap aborting */

                        error = YES;
                }
        }

        /* restore default signal trap */

        status = signal(SIGILL, SIG_DFL);
        if (status == BADSIG)
        {
                perror("PreMath signal default failed");
                exit(-1);
        }

        if (error)
        {
                PrintLine();
                fprintf(stderr, "%s failed: bad number parameters: ", name);
                PrintValue(stderr, Stack[FP+1].dummy, YES);
                fprintf(stderr, " and ");
                PrintValue(stderr, Stack[FP+2].dummy, YES);
                fprintf(stderr, "\n");
                ExecAbort();
        }
}


/*
PreMathTrap()

Trap the error signal that happens when bad parameters are given to a math
routine.
*/
```

```
PreMathTrap()
{
        longjmp(MathJump, YES);
}


/*
PrePack()

Pack a value structure into a string.  This is used to convert some general set
into a message or rule string suitable for a user classifier machine.
*/

PrePack(par)
        ParseThing * par;               /* function call in parse tree */
{
        char buffer[MAXSTRING+1];       /* temporary buffer for result */
        char * cp;                      /* pointer into "buffer" */

        cp = PackValue(buffer, Stack[FP+1].dummy->this,
                Stack[FP+2].dummy->this);

        Stack[FP].dummy->this = MakeValue(ValSTRING);
        Stack[FP].dummy->this->sign = 1;
        Stack[FP].dummy->this->string = CopyString(buffer);
}


/*
PrePrintf()

Formatted output with the "printf" subroutine to standard output.  We use
variable argument lists because the size of a string pointer may be different
than the size of a number.
*/

PrePrintf(par)
        ParseThing * par;               /* function call in parse tree */
{
        FioThing format;                /* format string */
        FioThing p1, p2, p3, p4, p5, p6, p7, p8, p9;    /* parameters */

        format = FioCheck(1, YES, NO, "printf");
        p1 = FioCheck(2, NO, NO, "printf");
        p2 = FioCheck(3, NO, NO, "printf");
        p3 = FioCheck(4, NO, NO, "printf");
        p4 = FioCheck(5, NO, NO, "printf");
        p5 = FioCheck(6, NO, NO, "printf");
        p6 = FioCheck(7, NO, NO, "printf");
        p7 = FioCheck(8, NO, NO, "printf");
        p8 = FioCheck(9, NO, NO, "printf");
        p9 = FioCheck(10, NO, NO, "printf");

        va_start(va_pvar);              /* start list */
        va_put(p1);
        va_put(p2);
        va_put(p3);
        va_put(p4);
        va_put(p5);
        va_put(p6);
        va_put(p7);
        va_put(p8);
        va_put(p9);
        va_end(va_pvar);                /* end list */

        _doprnt(format.pointer, &va_alist, stdout);
}


/*
```

```
PreRandom()

Generate a pseudo-random number from 0 and less than the caller's modulus.
*/

PreRandom(par)
        ParseThing * par;               /* function call in parse tree */
{
        int error;                      /* YES means bad value */
        long i;                         /* integral caller's parameter */
        long random();                  /* system random number routine */
        NUMBER x;                       /* caller's original parameter */

        error = NO;                     /* assume no errors */

        if ((Stack[FP+1].dummy->this == NULL)
        || (Stack[FP+1].dummy->this->type != ValNUMBER))
        {
                error = YES;
        }
        else
        {
                x = Stack[SP].dummy->this->number;
                i = (long) x;

                if (x != (NUMBER) i)
                {
                        /* truncation changes value */

                        error = YES;
                }
                else if (i <= 0)
                {
                        /* modulus must be positive */

                        error = YES;
                }
        }

        if (error)
        {
                PrintLine();
                fprintf(stderr,
                        "random failed: parameter must be a positive integer: ");
                PrintValue(stderr, Stack[FP+1].dummy, YES);
                fprintf(stderr, "\n");
                ExecAbort();
        }

        Stack[FP].dummy->this = MakeValue(ValNUMBER);
        Stack[FP].dummy->this->number = (NUMBER) (random() % i);
}


/*
PreRound()

Pre-defined function to round a number to the closest integral value.  If a
second parameter is given, then it must be a scale factor.  (For example, a
factor of 0.01 would round to the nearest penny.)
*/

PreRound(par)
        ParseThing * par;               /* function call in parse tree */
{
        int error;                      /* YES means bad values */
        long i;                         /* integral part */
        NUMBER x;                       /* number to be converted */
        NUMBER y;                       /* scale factor */

```

```
	error = NO;				/* assume no errors */

	/* get number to be converted */

	if ((Stack[FP+1].dummy->this != NULL)
	&& (Stack[FP+1].dummy->this->type == ValNUMBER))
	{
		x = Stack[FP+1].dummy->this->number;
	}
	else
		error = YES;

	/* get scale factor */

	if (Stack[FP+2].dummy->this == NULL)
	{
		y = ONE;
	}
	else if (Stack[FP+2].dummy->this->type == ValNUMBER)
	{
		y = Stack[FP+2].dummy->this->number;
		if (y == ZERO)
			error = YES;
	}
	else
		error = YES;

	/* do the conversion */

	if (!error)
	{
		if (x < ZERO)
			i = (long) ((x / y) - 0.5);
		else
			i = (long) ((x / y) + 0.5);
		Stack[FP].dummy->this = MakeValue(ValNUMBER);
		Stack[FP].dummy->this->number = y * (NUMBER) i;
	}

	if (error)
	{
		PrintLine();
		fprintf(stderr, "round failed: bad number parameters: ");
		PrintValue(stderr, Stack[FP+1].dummy, YES);
		fprintf(stderr, " and ");
		PrintValue(stderr, Stack[FP+2].dummy, YES);
		fprintf(stderr, "\n");
		ExecAbort();
	}
}


/*
PreSave()

Pre-defined function to save all global variables into a file named by the
user, or else print them on the terminal if no file name is given.
*/

PreSave(par)
	ParseThing * par;			/* function call in parse tree */
{
	char * cp;				/* pointer to file name string */
	FILE * fp;				/* file pointer */
	SymbolThing * sym;			/* a symbol table pointer */

	if (Stack[FP+1].dummy->this == NULL)
	{
		cp = NULL;
		fp = stdout;
```

```
	}
	else if (Stack[FP+1].dummy->this->type == ValSTRING)
	{
		cp = Stack[FP+1].dummy->this->string;
		fp = fopen(cp, "w");
		if (fp == NULL)
		{
			PrintLine();
			fprintf(stderr,
				"save failed: can't open file '%s' for writing\n",
				cp);
			ExecAbort();
		}
	}
	else
	{
		PrintLine();
		fprintf(stderr,
			"save failed: parameter must be a file name string or NULL: ");
		PrintValue(stderr, Stack[FP+1].dummy, YES);
		fprintf(stderr, "\n");
		ExecAbort();
	}

	sym = GlobalHead;
	while (sym != NULL)
	{
		if (sym->type == SymGLOBAL)
		{
			if (sym->special == 0)
			{
				fprintf(fp, "%s := ", sym->name);
				PrintValue(fp, sym->dummy, YES);
				fprintf(fp, ";\n");
			}
			else
			{
				UserSave(fp, sym);
			}
		}
		sym = sym->next;
	}

	if (cp != NULL)
		fclose(fp);
}


/*
PreScanf()

Formatted input with the "scanf" subroutine from wherever it is that we are
reading input ("yyin").
*/

PreScanf(par)
	ParseThing * par;		/* function call in parse tree */
{
	int count;			/* result from fscanf */
	FioThing format;		/* format string */
	FioThing p1, p2, p3, p4, p5, p6, p7, p8, p9;	/* parameters */

	format = FioCheck(1, YES, NO, "scanf");
	p1 = FioCheck(2, NO, YES, "scanf");
	p2 = FioCheck(3, NO, YES, "scanf");
	p3 = FioCheck(4, NO, YES, "scanf");
	p4 = FioCheck(5, NO, YES, "scanf");
	p5 = FioCheck(6, NO, YES, "scanf");
	p6 = FioCheck(7, NO, YES, "scanf");
	p7 = FioCheck(8, NO, YES, "scanf");
```

```
        p8 = FioCheck(9, NO, YES, "scanf");
        p9 = FioCheck(10, NO, YES, "scanf");

        count = fscanf(yyin, format.pointer, p1.pointer, p2.pointer,
                p3.pointer, p4.pointer, p5.pointer, p6.pointer, p7.pointer,
                p8.pointer, p9.pointer);

        Stack[FP].dummy->this = MakeValue(ValNUMBER);
        Stack[FP].dummy->this->number = (NUMBER) count;
}


/*
PreSign()

Pre-defined function to return the sign of an expression.
*/

PreSign(par)
        ParseThing * par;               /* function call in parse tree */
{
        if (Stack[FP+1].dummy->this == NULL)
        {
                /* sign(NULL) is just NULL */
        }
        else
        {
                NUMBER sign;

                CheckSign(Stack[FP+1].dummy->this);

                switch (Stack[FP+1].dummy->this->type)
                {
                case (ValNUMBER):
                        sign = Stack[FP+1].dummy->this->number;
                        if (sign < ZERO)
                                sign = - ONE;
                        else if (sign > ZERO)
                                sign = ONE;
                        break;
                case (ValSET):
                case (ValSTRING):
                        sign = (NUMBER) Stack[FP+1].dummy->this->sign;
                        break;
                default:
                        PrintLine();
                        fprintf(stderr,
                                "internal error: PreSign value type = %d\n",
                                Stack[FP+1].dummy->this->type);
                        ExecAbort();
                        break;
                }
                Stack[FP].dummy->this = MakeValue(ValNUMBER);
                Stack[FP].dummy->this->number = sign;
        }
}


/*
PreSize()

Pre-defined function to return the size of an expression.
*/

PreSize(par)
        ParseThing * par;               /* function call in parse tree */
{
        ValueThing * val;               /* a value structure pointer */
        NUMBER size;                    /* where we put the size */

```

```
        if (Stack[FP+1].dummy->this == NULL)
        {
                size = ZERO;
        }
        else switch (Stack[FP+1].dummy->this->type)
        {
        case (ValNUMBER):
                size = ONE;
                break;
        case (ValSET):
                size = ZERO;
                val = Stack[FP+1].dummy->this->next;
                while (val != NULL)
                {
                        size += ONE;
                        val = val->next;
                }
                break;
        case (ValSTRING):
                size = (NUMBER) strlen(Stack[FP+1].dummy->this->string);
                break;
        default:
                PrintLine();
                fprintf(stderr, "internal error: PreSize value type = %d\n",
                        Stack[FP+1].dummy->this->type);
                ExecAbort();
                break;
        }
        Stack[FP].dummy->this = MakeValue(ValNUMBER);
        Stack[FP].dummy->this->number = size;
}


/*
PreSprintf()

Formatted output with the "sprintf" subroutine to a text string.  Because of
problems with variable arguments, this routine is restricted to two data
parameters.

(Calling "_doprnt" with variable arguments into a string is too difficult.)
*/

PreSprintf(par)
        ParseThing * par;               /* function call in parse tree */
{
        FioThing format;                /* format string */
        FioThing p1, p2;                /* parameters */
        FioThing string;                /* a text string */

        string = FioCheck(1, YES, YES, "sprintf");
        format = FioCheck(2, YES, NO, "sprintf");
        p1 = FioCheck(3, NO, NO, "sprintf");
        p2 = FioCheck(4, NO, NO, "sprintf");

        if (SP > (FP + 4))
        {
                PrintLine();
                fprintf(stderr,
                        "warning: sprintf called with too many parameters (%d)\n",
                        (SP-FP));
        }

        if (p1.type == FioNUMBER)
        {
                if (p2.type == FioNUMBER)
                        sprintf(string.pointer, format.pointer, p1.number,
                                p2.number);
                else
                        sprintf(string.pointer, format.pointer, p1.number,
```

```
                                        p2.pointer);
        }
        else
        {
                if (p2.type == FioNUMBER)
                        sprintf(string.pointer, format.pointer, p1.pointer,
                                p2.number);
                else
                        sprintf(string.pointer, format.pointer, p1.pointer,
                                p2.pointer);
        }
}


/*
PreSscanf()

Formatted input with the "sscanf" subroutine from a text string.
*/

PreSscanf(par)
        ParseThing * par;               /* function call in parse tree */
{
        int count;                      /* result from sscanf */
        FioThing format;                /* format string */
        FioThing p1, p2, p3, p4, p5, p6, p7, p8, p9;    /* parameters */
        FioThing string;                /* a text string */

        string = FioCheck(1, YES, NO, "sscanf");
        format = FioCheck(2, YES, NO, "sscanf");
        p1 = FioCheck(3, NO, YES, "sscanf");
        p2 = FioCheck(4, NO, YES, "sscanf");
        p3 = FioCheck(5, NO, YES, "sscanf");
        p4 = FioCheck(6, NO, YES, "sscanf");
        p5 = FioCheck(7, NO, YES, "sscanf");
        p6 = FioCheck(8, NO, YES, "sscanf");
        p7 = FioCheck(9, NO, YES, "sscanf");
        p8 = FioCheck(10, NO, YES, "sscanf");
        p9 = FioCheck(11, NO, YES, "sscanf");

        count = sscanf(string.pointer, format.pointer, p1.pointer, p2.pointer,
                p3.pointer, p4.pointer, p5.pointer, p6.pointer, p7.pointer,
                p8.pointer, p9.pointer);

        Stack[FP].dummy->this = MakeValue(ValNUMBER);
        Stack[FP].dummy->this->number = (NUMBER) count;
}


/*
PreStop()

Pre-defined function to stop the current user program or statement.  We do
this by the crude method of forcing an execution abort (just like some sort
of fatal error).
*/

PreStop(par)
        ParseThing * par;               /* function call in parse tree */
{
        fprintf(stderr, "\n");
        PrintLine();
        fprintf(stderr, "stop() called; ready for more input\n");
        ExecAbort();
}


/*
PreSystem()

```

```
Pre-defined function to call the UNIX "sh" shell with a command line.
*/

PreSystem(par)
        ParseThing * par;                   /* function call in parse tree */
{
        if ((Stack[FP+1].dummy->this != NULL)
        && (Stack[FP+1].dummy->this->type == ValSTRING))
        {
                system(Stack[FP+1].dummy->this->string);
        }
        else
        {
                PrintLine();
                fprintf(stderr,
                        "system failed: parameter must be a command string: ");
                PrintValue(stderr, Stack[FP+1].dummy, YES);
                fprintf(stderr, "\n");
                ExecAbort();
        }
}


/*
PreTrunc()

Pre-defined function to truncate a number to its integral part.  If a second
parameter is given, then it must be a scale factor.  (For example, a factor of
0.01 would truncate to the penny.)
*/

PreTrunc(par)
        ParseThing * par;                   /* function call in parse tree */
{
        int error;                          /* YES means bad values */
        long i;                             /* integral part */
        NUMBER x;                           /* number to be converted */
        NUMBER y;                           /* scale factor */

        error = NO;                         /* assume no errors */

        /* get number to be converted */

        if ((Stack[FP+1].dummy->this != NULL)
        && (Stack[FP+1].dummy->this->type == ValNUMBER))
        {
                x = Stack[FP+1].dummy->this->number;
        }
        else
                error = YES;

        /* get scale factor */

        if (Stack[FP+2].dummy->this == NULL)
        {
                y = ONE;
        }
        else if (Stack[FP+2].dummy->this->type == ValNUMBER)
        {
                y = Stack[FP+2].dummy->this->number;
                if (y == ZERO)
                        error = YES;
        }
        else
                error = YES;

        /* do the conversion */

        if (!error)
        {
```

```
                        i = (long) (x / y);
                        Stack[FP].dummy->this = MakeValue(ValNUMBER);
                        Stack[FP].dummy->this->number = y * (NUMBER) i;
                }

                if (error)
                {
                        PrintLine();
                        fprintf(stderr, "trunc failed: bad number parameters: ");
                        PrintValue(stderr, Stack[FP+1].dummy, YES);
                        fprintf(stderr, " and ");
                        PrintValue(stderr, Stack[FP+2].dummy, YES);
                        fprintf(stderr, "\n");
                        ExecAbort();
                }
}


/*
PreType()

Pre-defined function to return the type of an expression.  If a second
parameter is given, then it must be a string.  We compare the type of the
first parameter with this string, and return TRUE if the types are the same,
and FALSE if they are different.
*/

PreType(par)
        ParseThing * par;               /* function call in parse tree */
{
        char * type;                    /* type string pointer */

        if (Stack[FP+1].dummy->this == NULL)
        {
                type = "null";
        }
        else switch (Stack[FP+1].dummy->this->type)
        {
        case (ValDUMMY):
                type = "dummy";
                break;
        case (ValELEMENT):
                type = "element";
                break;
        case (ValNUMBER):
                type = "number";
                break;
        case (ValSET):
                type = "set";
                break;
        case (ValSTRING):
                type = "string";
                break;
        default:
                PrintLine();
                fprintf(stderr, "internal error: PreType value type = %d\n",
                        Stack[FP+1].dummy->this->type);
                ExecAbort();
                break;
        }

        /* do we return the type, or compare it with another string? */

        if (Stack[FP+2].dummy->this == NULL)
        {
                Stack[FP].dummy->this = MakeValue(ValSTRING);
                Stack[FP].dummy->this->sign = 1;
                Stack[FP].dummy->this->string = CopyString(type);
        }
        else if (Stack[FP+2].dummy->this->type == ValSTRING)
```

```
		{
			Stack[FP].dummy->this = MakeValue(ValNUMBER);
			if (strcmp(Stack[FP+2].dummy->this->string, type) == 0)
				Stack[FP].dummy->this->number = TRUE;
			else
				Stack[FP].dummy->this->number = FALSE;
		}
		else
		{
			PrintLine();
			fprintf(stderr,
				"type failed: second parameter must be a string or NULL: ");
			PrintValue(stderr, Stack[FP+2].dummy, YES);
			fprintf(stderr, "\n");
			ExecAbort();
		}
	}


	/*
	PreUnpack()

	Pre-defined function to return a value that looks like the first parameter,
	but with all strings replaced by a value that looks like the second parameter.
	For each string, this is the reverse operation of "pack()".
	*/

	PreUnpack(par)
		ParseThing * par;		/* function call in parse tree */
	{
		Stack[FP].dummy->this = UnpackValue(Stack[FP+1].dummy->this,
			Stack[FP+2].dummy->this);
	}


	/*
	PreValue()

	Pre-defined value function.  Manually try to parse a string into a value
	structure.

	WARNING:  This function was written to read back message and rule lists from
	the user classifier system.  It is not documented, and is subject to change.
	*/

	PreValue(par)
		ParseThing * par;		/* function call in parse tree */
	{
		char * cp;			/* a character pointer */
		int error;			/* YES means bad string */

		error = NO;			/* assume no errors */

		if ((Stack[FP+1].dummy->this == NULL)
		|| (Stack[FP+1].dummy->this->type != ValSTRING))
		{
			error = YES;
		}
		else
		{
			cp = Stack[FP+1].dummy->this->string;
			Stack[FP].dummy->this = StringToValue(cp, cp, &cp);

			if ((*cp) != '\0')	/* did we use entire string? */
				error = YES;
		}

		if (error)
		{
			PrintLine();
```

```
                        fprintf(stderr, "value failed: bad string parameter: ");
                        PrintValue(stderr, Stack[FP+1].dummy, YES);
                        fprintf(stderr, "\n");
                        ExecAbort();
                }
}


/*
PreWrite()

Pre-defined write function.  Write out all of the parameters without adding
spaces, newlines, or quotes to strings.
*/

PreWrite(par)
        ParseThing * par;                       /* function call in parse tree */
{
        int i;                                  /* index variable */

        for (i = (FP + 1) ; i <= SP ; i ++)
                PrintValue(stdout, Stack[i].dummy, NO);
}
```

```
/*

fasub.c -- Support Subroutines


Keith Fenske
Department of Computing Science
The University of Alberta
Edmonton, Alberta, Canada
T6G 2H1

December 1987

Copyright (c) 1987 by Keith Fenske.  All rights reserved.


These are the support subroutines for various nifty things.  Most are called
by the pre-defined functions in the "fapre.c" module.

*/

#include "fainc.h"                          /* our standard includes */
#include <ctype.h>                          /* character types */
#include <math.h>                           /* math routines */


/*
CheckSign()

Check the "sign" of a value structure.  It should be zero for some value types,
and +1 or -1 for others.

This function could be generalized to check (for each value type) that the
fields which should be unused have their default values.  The additional CPU
time is probably not warranted in a production version.
*/

CheckSign(val)
        ValueThing * val;                   /* a value structure pointer */
{
        if (val == NULL)
        {
                PrintLine();
                fprintf(stderr,
                        "internal error: CheckSign value pointer = NULL\n");
                ExecAbort();
        }
        else switch (val->type)
        {
        case (ValDUMMY):
        case (ValNUMBER):
                if (val->sign != 0)
                {
                        PrintLine();
                        fprintf(stderr,
                                "internal error: CheckSign sign = %d not zero\n",
                                val->sign);
                        ExecAbort();
                }
                break;
        case (ValSET):
        case (ValSTRING):
                if (val->number != ZERO)
                {
                        PrintLine();
                        fprintf(stderr, "internal error: CheckSign number = ");
                        fprintf(stderr, FORMAT, val->number);
                        fprintf(stderr, " not zero\n");
                        ExecAbort();
                }
```

```
			if ((val->sign != -1) && (val->sign != 1))
			{
				PrintLine();
				fprintf(stderr,
					"internal error: CheckSign sign = %d not +-1\n",
					val->sign);
				ExecAbort();
			}
			break;
		default:
			PrintLine();
			fprintf(stderr, "internal error: CheckSign value type = %d\n",
				val->type);
			ExecAbort();
			break;
	}
}


/*
CompareValue()

Given two value structures, compare them or give up.  An integer is returned:

			CMPEQ if left = right
			CMPGT if left > right
			CMPLT if left < right
			CMPNE if left and right are not comparable

We keep track of the sign of "left" and "right" so that for sets and strings
we can compare the positive parts and then reverse CMPGT and CMPLT results if
necessary.
*/

int CompareValue(left, right, pattern)
	ValueThing * left;		/* left value structure */
	ValueThing * right;		/* right value structure */
	int pattern;			/* YES if "#" allowed in strings */
{
	int result;			/* result of comparison */
	int sign;			/* sign of comparison: -1 or +1 */

	result = CMPEQ;			/* assume equal */
	sign = 1;			/* don't reverse result */

	if (left == NULL)
	{
		if (right == NULL)
			result = CMPEQ;
		else
			result = CMPNE;
	}
	else if (right == NULL)
	{
		result = CMPNE;
	}
	else if ((left->type == ValNUMBER) && (right->type == ValNUMBER))
	{
		if (left->number < right->number)
			result = CMPLT;
		else if (left->number > right->number)
			result = CMPGT;
		else
			result = CMPEQ;
	}
	else if ((left->type == ValSET) && (right->type == ValSET))
	{
		CheckSign(left);
		CheckSign(right);
```

```
			sign = left->sign;

			if (left->sign != right->sign)
			{
				result = CMPGT;		/* sign may get reversed */
			}
			else
			{
				ValueThing * lp;	/* left element */
				ValueThing * rp;	/* right element */

				lp = left->next;	/* first element on left */
				rp = right->next;	/* first element on right */

				result = CMPEQ;		/* assume equal */

				while ((lp != NULL) && (rp != NULL))
				{
					result = CompareValue(lp->this, rp->this,
						pattern);
					if (result != CMPEQ)
						break;
					lp = lp->next;
					rp = rp->next;
				}
				if (result == CMPEQ)
				{
					if (lp != NULL)
						result = CMPGT;
					else if (rp != NULL)
						result = CMPLT;
				}
			}
		}
		else if ((left->type == ValSTRING) && (right->type == ValSTRING))
		{
			CheckSign(left);
			CheckSign(right);

			sign = left->sign;

			if (left->sign != right->sign)
			{
				result = CMPGT;		/* sign may get reversed */
			}
			else
			{
				char * lp, * rp;	/* left, right pointers */

				lp = left->string;
				rp = right->string;

				if (pattern)
				{
					result = CMPEQ;		/* assume equal */

					while (((*lp) != '\0') && ((*rp) != '\0'))
					{
						if (((*lp) != ANYBIT)
						&& ((*rp) != ANYBIT))
						{
							if ((*lp) < (*rp))
							{
								result = CMPLT;
								break;
							}
							else if ((*lp) > (*rp))
							{
								result = CMPGT;
								break;
```

```
								}
							}
							lp ++;
							rp ++;
						}

						/* at end-of-string, '\0' < '#' */

						if (result == CMPEQ)
						{
							if ((*lp) != '\0')
								result = CMPGT;
							if ((*rp) != '\0')
								result = CMPLT;
						}
					}
					else
					{
						int status;	/* result from strcmp */

						status = strcmp(lp, rp);

						if (status < 0)
							result = CMPLT;
						else if (status > 0)
							result = CMPGT;
						else
							result = CMPEQ;
					}
				}
		}
		else
		{
			/* not comparable */

			result = CMPNE;
		}

		/* reverse the meaning of "less than" and "greater than"? */

		if (sign < 0)
		{
			switch (result)
			{
			case (CMPEQ):
				return(CMPEQ);
				break;
			case (CMPGT):
				return(CMPLT);
				break;
			case (CMPLT):
				return(CMPGT);
				break;
			case (CMPNE):
				return(CMPNE);
				break;
			default:
				PrintLine();
				fprintf(stderr,
					"internal error: CompareValue result = %d\n",
					result);
				ExecAbort();
				break;
			}
		}
		else
		{
			return(result);
		}
	}
```

```c


/*
FindNumber()

Given the address of a string, try to find a number.  We return the address
in the string after the number.

Note:  This routine does NOT accept a leading sign ("+" or "-"), because of
who calls it (StringToValue, UnpackString).
*/

NUMBER FindNumber(cp, newcp)
        char * cp;                      /* current string pointer */
        char * * newcp;                 /* returned string pointer */
{
        double atof();                  /* ASCII to floating double */
        char * new;                     /* our string pointer */
        NUMBER result;                  /* our resulting number */

        new = cp;                       /* starting address */

        /* skip digits before decimal point */

        while (isascii(*new) && isdigit(*new))
                new ++;

        /* skip decimal point (if any) and trailing digits */

        if ((*new) == '.')
        {
                new ++;
                while (isascii(*new) && isdigit(*new))
                        new ++;
        }

        /* skip exponent (if any) */

        if (((*new) == 'e') || ((*new) == 'E'))
        {
                new ++;
                if (((*new) == '-') || ((*new) == '+'))
                        new ++;
                while (isascii(*new) && isdigit(*new))
                        new ++;
        }

        /* convert ASCII to number and return new string pointer */

        result = atof(cp);
        (*newcp) = new;
        return(result);
}


/*
PackValue()

Pack a value structure into a string.  This is used to compress a set into a
true bit string.  It has been defined to be quite general, although this
generality may not be useful.

The "value" parameter may be NULL, a number, a set, or a string.  The "pattern"
parameter must have the same structure.  If "value" is a set, then "pattern"
may be a set with a similar but simpler structure (fewer levels), such as a
number or string.

Assume that "value" and "pattern" are both sets with identical structures.
Then, for each NULL element in "value", the corresponding element in "pattern"
is converted into a string and concatenated to this function's result.
```

```
Non-NULL elements in "value" are converted to a string that looks like the
"pattern" element.  If a "value" element has a negative sign, then the sign is
only preserved if the "pattern" element is also negative.
*/

char * PackValue(cp, value, pattern)
        char * cp;                          /* where output goes */
        ValueThing * value;                 /* value to be packed */
        ValueThing * pattern;               /* pattern to be used */
{
        char * new;                         /* new output pointer */

        new = cp;                           /* default to nothing */

        if (value == NULL)
        {
                if (pattern == NULL)
                {
                        /* do nothing */
                }
                else
                {
                        /* use pattern element instead */

                        new = PackValue(new, pattern, NULL);
                }
        }
        else if ((value->type == ValNUMBER) && (pattern == NULL))
        {
                sprintf(new, FORMAT, value->number);
                new += strlen(new);                 /* find new string end */
        }
        else if ((value->type == ValNUMBER) && (pattern->type == ValNUMBER))
        {
                if (pattern->number < ZERO)
                        sprintf(new, FORMAT, value->number);
                else
                        sprintf(new, FORMAT, fabs(value->number));
                new += strlen(new);                 /* find new string end */
        }
        else if ((value->type == ValSET) && (pattern == NULL))
        {
                ValueThing * val;                   /* current part of "value" */

                CheckSign(value);
                if (value->sign < 0)
                        (*new ++) = '-';

                val = value->next;                  /* first value element */

                while (val != NULL)
                {
                        new = PackValue(new, val->this, pattern);
                        val = val->next;
                }
        }
        else if ((value->type == ValSET) && (pattern->type == ValNUMBER))
        {
                ValueThing * val;                   /* current part of "value" */

                CheckSign(value);
                if ((pattern->number < ZERO) && (value->sign < 0))
                        (*new ++) = '-';

                val = value->next;                  /* first value element */

                while (val != NULL)
                {
                        new = PackValue(new, val->this, pattern);
                        val = val->next;
```

```
		}
	}
	else if ((value->type == ValSET) && (pattern->type == ValSTRING))
	{
		ValueThing * val;		/* current part of "value" */

		CheckSign(pattern);
		CheckSign(value);
		if ((pattern->sign < 0) && (value->sign < 0))
			(*new ++) = '-';

		val = value->next;		/* first value element */

		while (val != NULL)
		{
			new = PackValue(new, val->this, pattern);
			val = val->next;
		}
	}
	else if ((value->type == ValSET) && (pattern->type == ValSET))
	{
		ValueThing * pat;		/* current part of "pattern" */
		ValueThing * val;		/* current part of "value" */

		CheckSign(pattern);
		CheckSign(value);
		if ((pattern->sign < 0) && (value->sign < 0))
			(*new ++) = '-';

		pat = pattern->next;		/* first pattern element */
		val = value->next;		/* first value element */

		while ((pat != NULL) || (val != NULL))
		{
			if (pat == NULL)
			{
				new = PackValue(new, val->this, NULL);
				val = val->next;
			}
			else if (val == NULL)
			{
				new = PackValue(new, pat->this, NULL);
				pat = pat->next;
			}
			else
			{
				new = PackValue(new, val->this, pat->this);
				pat = pat->next;
				val = val->next;
			}
		}
	}
	else if ((value->type == ValSTRING) && (pattern == NULL))
	{
		CheckSign(value);
		if (value->sign < 0)
			(*new ++) = '-';

		strcpy(new, value->string);
		new += strlen(new);		/* find new string end */
	}
	else if ((value->type == ValSTRING) && (pattern->type == ValSTRING))
	{
		int j, k;			/* string lengths */
		char * pp;			/* pattern string pointer */
		char * vp;			/* value string pointer */

		CheckSign(pattern);
		CheckSign(value);
		if ((pattern->sign < 0) && (value->sign < 0))
```

```
					(*new ++) = '-';

			pp = pattern->string;
			vp = value->string;

			j = strlen(vp);				/* value length */
			k = strlen(pp);				/* pattern length */

			if (j <= k)				/* if value <= pattern */
			{
				strcpy(new, vp);		/* copy value string */
				new += j;			/* new output pointer */
				pp += j;			/* get extra from pattern */
				for ( ; j < k ; j ++)
					(*new ++) = (*pp ++);
			}
			else					/* if value > pattern */
			{
				for (j = 0 ; j < k ; j ++)
					(*new ++) = (*vp ++);
			}
		}
		else
		{
			PrintLine();
			fprintf(stderr, "pack failed: can't pack ");
			PrintValue(stderr, value, YES);
			fprintf(stderr, " into pattern ");
			PrintValue(stderr, pattern, YES);
			fprintf(stderr, "\n");
			ExecAbort();
		}

		/* put an end-of-string marker and return */

		(*new) = '\0';
		return(new);
}


/*
PrintString()

Print a string (possibly quoted) using a given file pointer.
*/

PrintString(fp, cp, quote)
		FILE * fp;			/* a file pointer */
		char * cp;			/* string pointer */
		int quote;			/* YES if we quote strings */
{
		if (quote)
		{
			fprintf(fp, "%c", QUOTE);
			while (*cp)
			{
				if ((*cp) == '\n')
					fprintf(fp, "\\n");
				else if ((*cp) == '\t')
					fprintf(fp, "\\t");
				else if ((*cp) == '\\')
					fprintf(fp, "\\\\");
				else if ((*cp) == QUOTE)
					fprintf(fp, "\\%c", QUOTE);
				else
					fprintf(fp, "%c", (*cp));
				cp ++;
			}
			fprintf(fp, "%c", QUOTE);
		}
```

```
        else
                fprintf(fp, "%s", cp);
}


/*
PrintValue()

Recursively print a value structure.  The caller gives us a file pointer (like
"stdout"), and is responsible for any newlines before or after the output.

Unlike most subroutines, the caller may pass us a dummy value structure.  This
is provided as a courtesy only, due to the large number of PrintValue() calls.
*/

PrintValue(fp, val, quote)
        FILE * fp;                      /* a file pointer */
        ValueThing * val;               /* a value structure pointer */
        int quote;                      /* YES if we quote strings */
{
        int comma;                      /* "comma required" flag */
        ValueThing * new;               /* new value structure pointer */

        if (val == NULL)
        {
                fprintf(fp, "NULL");
        }
        else
        {
                CheckSign(val);

                switch (val->type)
                {
                case (ValDUMMY):
                        PrintValue(fp, val->this, quote);
                        break;
                case (ValNUMBER):
                        fprintf(fp, FORMAT, val->number);
                        break;
                case (ValSET):
                        if (val->sign < 0)
                                fprintf(fp, "-");
                        fprintf(fp, "{");
                        comma = NO;
                        new = val->next;
                        while (new != NULL)
                        {
                                if (comma)
                                {
                                        if (quote)
                                                fprintf(fp, ", ");
                                        else
                                                fprintf(fp, ",");
                                }
                                PrintValue(fp, new->this, quote);
                                comma = YES;
                                new = new->next;
                        }
                        fprintf(fp, "}");
                        break;
                case (ValSTRING):
                        if (val->sign < 0)
                                fprintf(fp, "-");
                        PrintString(fp, val->string, quote);
                        break;
                default:
                        PrintLine();
                        fprintf(stderr,
                                "internal error: PrintValue value type = %d\n",
                                val->type);
```

```
                                ExecAbort();
                                break;
                        }
                }
}


/*
SkipSpace()

Given a string pointer, return a new pointer which skips over any white space
(blanks, tabs, or newlines).
*/

char * SkipSpace(cp)
        char * cp;                              /* string pointer */
{
        char * new;                             /* new string pointer */

        new = cp;
        while (isascii(*new) && isspace(*new))
        {
                new ++;
        }
        return(new);
}


/*
StringToValue()

Given a string pointer, try to parse the string as the definition of a single
data object (number, string, set, or NULL value).  This is a crude version of
the real YACC parser, and is used to take apart message and rule lists from
the user classifier system.

The following features are not supported:

        - missing set elements are NOT converted to NULL values
        - escape sequences are NOT allowed in strings

WARNING:  This function was written to read back message and rule lists from
the user classifier system.  It is not documented, and is subject to change.
*/

ValueThing * StringToValue(start, cp, newcp)
        char * start;                           /* start of complete string */
        char * cp;                              /* where we start looking */
        char * * newcp;                         /* where we return new pointer */
{
        int error;                              /* YES means bad string */
        char * new;                             /* new string pointer */
        ValueThing * result;                    /* resulting value structure */
        int sign;                               /* sign of result */

        error = NO;                             /* assume no errors */
        new = cp;                               /* start looking here */
        result = NULL;                          /* assume no result */
        sign = 1;                               /* assume positive sign */

        new = SkipSpace(new);                   /* skip white space */

        /* look for a sign */

        if ((*new) == '\0')
        {
                error = YES;
        }
        else if ((*new) == '+')
        {
```

```
			new = SkipSpace(new+1);
		}
		else if ((*new) == '-')
		{
			new = SkipSpace(new+1);
			sign = -1;
		}

		/* find the data type */

		if (error)
		{
			/* do nothing at this level */
		}
		else if (((*new) == 'n') || ((*new) == 'N'))
		{
			/* must be the NULL value */

			if ((((*++new) == 'u') || ((*new) == 'U'))
			&& (((*++new) == 'l') || ((*new) == 'L'))
			&& (((*++new) == 'l') || ((*new) == 'L')))
			{
				new ++;
				result = NULL;
			}
			else
				error = YES;
		}
		else if (isascii(*new) && (isdigit(*new) || ((*new) == '.')))
		{
			/* must be a number */

			result = MakeValue(ValNUMBER);
			result->number = FindNumber(new, &new) * sign;
		}
		else if ((*new) == '{')
		{
			/* must be a set */

			ValueThing * val;

			new = SkipSpace(new+1);
			result = val = MakeValue(ValSET);
			val->sign = sign;

			if ((*new) == '}')			/* end of set? */
			{
				new ++;
			}
			else
			{
				while (YES)
				{
					val->next = MakeValue(ValELEMENT);
					val = val->next;
					val->this = StringToValue(start, new, &new);

					if ((*new) == ',')
					{
						new ++;		/* more elements */
					}
					else if ((*new) == '}')
					{
						new ++;		/* end of set */
						break;
					}
					else
					{
						error = YES;
						break;
```

```
                                }
                        }
                }
        }
        else if (((*new) == '"') || ((*new) == '\'') || ((*new) == 'L'))
        {
                /* must be a string */

                char delim;             /* string delimiter */
                char * old;             /* starting address */

                delim = *new ++;
                old = new;
                while (YES)
                {
                        if ((*new) == '\0')
                        {
                                error = YES;    /* missing string delimiter */
                                break;
                        }
                        else if ((*new) == delim)
                        {
                                break;
                        }
                        else
                        {
                                new ++;
                        }
                }

                if (!error)
                {
                        char * copy;    /* copy of string */

                        result = MakeValue(ValSTRING);
                        result->sign = sign;
                        result->string = copy = GetMemory(new - old + 1);

                        while (old != new)
                        {
                                (*copy ++) = (*old ++);
                        }
                        (*copy) = '\0';

                        new ++;         /* skip over final delimiter */
                }
        }
        else
                error = YES;

        if (error)
        {
                PrintLine();
                fprintf(stderr,
                        "StringToValue failed: bad string parameter: ");
                PrintString(stderr, start, YES);
                fprintf(stderr, "\n");
                ExecAbort();
        }

        (*newcp) = SkipSpace(new);
        return(result);
}


/*
UnpackString()

Given a string and a pattern value, attempt to take the string apart and create
a value structure that looks like the pattern.  The caller must give us the
```

```
starting string pointer, and we return (as a parameter) the updated pointer.
*/

ValueThing * UnpackString(cp, pattern, newcp)
	char * cp;			/* starting string pointer */
	ValueThing * pattern;		/* pattern to be used */
	char * * newcp;			/* where we return new pointer */
{
	ValueThing * result;		/* our result */
	int sign;			/* sign of a value */

	if ((*cp) == '\0')
	{
		result = CopyValue(pattern);
	}
	else if (pattern == NULL)
	{
		result = NULL;
	}
	else if (pattern->type == ValNUMBER)
	{
		sign = 1;
		if (pattern->number < ZERO)
		{
			if ((*cp) == '+')
				cp ++;
			else if ((*cp) == '-')
			{
				sign = -1;
				cp ++;
			}
		}
		result = MakeValue(ValNUMBER);
		result->number = FindNumber(cp, &cp) * sign;
	}
	else if (pattern->type == ValSET)
	{
		ValueThing * new;		/* a new set */
		ValueThing * pat;		/* current part of "pattern" */

		sign = 1;
		CheckSign(pattern);
		if (pattern->sign < 0)
		{
			if ((*cp) == '+')
				cp ++;
			else if ((*cp) == '-')
			{
				sign = -1;
				cp ++;
			}
		}
		new = result = MakeValue(ValSET);	/* a new set */
		result->sign = sign;

		pat = pattern->next;		/* first pattern element */
		while (pat != NULL)
		{
			new->next = MakeValue(ValELEMENT);
			new->next->this = UnpackString(cp, pat->this, &cp);
			new = new->next;
			pat = pat->next;
		}
	}
	else if (pattern->type == ValSTRING)
	{
		int j, k;			/* string lengths */
		char * pp;			/* pattern pointer */
		char * rp;			/* result pointer */

```

```
                sign = 1;
                CheckSign(pattern);
                if (pattern->sign < 0)
                {
                        if ((*cp) == '+')
                                cp ++;
                        else if ((*cp) == '-')
                        {
                                sign = -1;
                                cp ++;
                        }
                }
                result = MakeValue(ValSTRING);
                result->sign = sign;

                pp = pattern->string;
                j = strlen(cp);                 /* remaining input */
                k = strlen(pp);                 /* pattern length */

                result->string = rp = GetMemory(k + 1);

                if (j <= k)                     /* input smaller than pattern */
                {
                        strcpy(rp, cp);         /* copy all input */
                        cp += j;
                        pp += j;                /* get more from pattern */
                        rp += j;
                        for ( ; j < k ; j ++)
                                (*rp ++) = (*pp ++);
                }
                else /* j > k */                /* more input than pattern */
                {
                        for (j = 0 ; j < k ; j ++)
                                (*rp ++) = (*cp ++);
                }
                (*rp) = '\0';                   /* end result string */
        }
        else
        {
                PrintLine();
                fprintf(stderr,
                        "internal error: UnpackString pattern type = %d\n",
                        pattern->type);
                ExecAbort();
        }

        /* pass back new string pointer, and return value */

        (*newcp) = cp;
        return(result);
}


/*
UnpackValue()

Given a value structure, return a new value structure which looks like the
old value, except that all strings have been broken apart by a "pattern".
The result of this function is dynamic, and may be assigned freely.
*/

ValueThing * UnpackValue(value, pattern)
        ValueThing * value;             /* original value */
        ValueThing * pattern;           /* pattern structure */
{
        ValueThing * result;            /* our result */

        if (pattern == NULL)
        {
                /* can't unpack without a pattern */
```

```
                result = NULL;
        }
        else if (value == NULL)
        {
                /* replace NULL value with the pattern */

                result = CopyValue(pattern);
        }
        else if (value->type == ValNUMBER)
        {
                /* numbers are unchanged */

                result = CopyValue(value);
        }
        else if (value->type == ValSET)
        {
                /* sets are unpacked recursively */

                ValueThing * new;               /* new set */
                ValueThing * val;               /* current part of "value" */

                new = result = MakeValue(ValSET);
                CheckSign(value);
                result->sign = value->sign;

                val = value->next;              /* first value element */
                while (val != NULL)
                {
                        new->next = MakeValue(ValELEMENT);
                        new->next->this = UnpackValue(val->this, pattern);
                        new = new->next;
                        val = val->next;
                }
        }
        else if (value->type == ValSTRING)
        {
                /* strings are so much work, we call somebody else */

                char * cp;                      /* dummy string pointer */

                result = UnpackString(value->string, pattern, &cp);

                CheckSign(value);
                if ((result != NULL) && (value->sign < 0))
                {
                        CheckSign(result);
                        if (result->type == ValNUMBER)
                                result->number = - result->number;
                        else
                                result->sign = - result->sign;
                }
        }
        else
        {
                PrintLine();
                fprintf(stderr, "unpack failed: can't unpack ");
                PrintValue(stderr, value, YES);
                fprintf(stderr, " using pattern ");
                PrintValue(stderr, pattern, YES);
                fprintf(stderr, "\n");
                ExecAbort();
        }

        return(result);
}
```

```
/*

fause.c -- User Support


Keith Fenske
Department of Computing Science
The University of Alberta
Edmonton, Alberta, Canada
T6G 2H1

December 1987

Copyright (c) 1987 by Keith Fenske.  All rights reserved.


All knowledge and special code necessary to support an interface to the user's
classifier system is buried in this module.  These routines can be safely
omitted without affecting the language itself.  (There are a few pre-defined
function references in the main code that would need to be commented out.)

*/

#include "fainc.h"                          /* our standard includes */

#define MESSLIST 301                /* special code for "messlist" variable */
#define READY "ready"               /* user classifier prompt for input */
#define RULELIST 302                /* special code for "rulelist" variable */

        char UserBuffer[MAXSTRING+1];       /* buffer for UserReceive() */
        int UserFlagMess = YES;             /* YES if "messlist" is invalid */
        int UserFlagRule = YES;             /* YES if "rulelist" is invalid */
        int UserReadFD = -1;                /* user pipe read file descriptor */
        int UserWriteFD = -1;               /* user pipe write file descriptor */


/*
PreClose()

Pre-defined function to close the connection to the user classifier system, if
it isn't already closed.  There are no parameters, and no result.
*/

PreClose(par)
        ParseThing * par;                   /* function call in parse tree */
{
        UserClose();
}


/*
PreFlagMess()

Pre-defined function to flag the user classifier message list ("messlist") as
invalid, so that it will be re-fetched upon the next reference.  This is
necessary after sending any command that MAY change the message list.
*/

PreFlagMess(par)
        ParseThing * par;                   /* function call in parse tree */
{
        UserFlagMess = YES;
}


/*
PreFlagRule()

Pre-defined function to flag the user classifier rule list ("rulelist") as
invalid, so that it will be re-fetched upon the next reference.  This is
```

```
necessary after sending any command that MAY change the rule list.
*/

PreFlagRule(par)
        ParseThing * par;               /* function call in parse tree */
{
        UserFlagRule = YES;
}


/*
PreOpen()

Pre-defined function to open a connection to the user classifier system, if it
isn't already open.  If the first parameter is given, then it must be the name
of the executable classifier file.  If the second parameter is given, then it
must be the first argument to the user classifier.  If either parameter is
missing, then the defaults are used.
*/

PreOpen(par)
        ParseThing * par;               /* function call in parse tree */
{
        char * arg;                     /* argument to user classifier */
        int error;                      /* YES means bad values */
        char * file;                    /* file name of user classifier */

        error = NO;                     /* assume no errors */

        /* get file name */

        if (Stack[FP+1].dummy->this == NULL)
        {
                file = UserFile;        /* default */
        }
        else if (Stack[FP+1].dummy->this->type == ValSTRING)
        {
                file = Stack[FP+1].dummy->this->string;
        }
        else
                error = YES;

        /* get first argument to classifier */

        if (Stack[FP+2].dummy->this == NULL)
        {
                arg = UserArg;          /* default */
        }
        else if (Stack[FP+2].dummy->this->type == ValSTRING)
        {
                arg = Stack[FP+2].dummy->this->string;
        }
        else
                error = YES;

        /* open connection */

        if (!error)
        {
                UserOpen(file, arg);
        }

        if (error)
        {
                PrintLine();
                fprintf(stderr, "open failed: bad string parameters: ");
                PrintValue(stderr, Stack[FP+1].dummy, YES);
                fprintf(stderr, " and ");
                PrintValue(stderr, Stack[FP+2].dummy, YES);
                fprintf(stderr, "\n");
```

```
                        ExecAbort();
                }
        }


        /*
        PreReceive()

        Pre-defined function to receive a string from the user classifier system.  If
        a parameter is given, then we keep waiting until we get this "expected"
        response string, and nothing is returned to the caller.  (May get abused by the
        user.)
        */

        PreReceive(par)
                ParseThing * par;               /* function call in parse tree */
        {
                char * cp;                      /* a string pointer */

                if (Stack[FP+1].dummy->this == NULL)
                {
                        cp = UserReceive();
                        Stack[FP].dummy->this = MakeValue(ValSTRING);
                        Stack[FP].dummy->this->sign = 1;
                        Stack[FP].dummy->this->string = CopyString(cp);
                }
                else if (Stack[FP+1].dummy->this->type == ValSTRING)
                {
                        UserReady(Stack[FP+1].dummy->this->string);
                }
                else
                {
                        PrintLine();
                        fprintf(stderr,
                                "receive failed: parameter must be a string or NULL: ");
                        PrintValue(stderr, Stack[FP+1].dummy, YES);
                        fprintf(stderr, "\n");
                        ExecAbort();
                }
        }


        /*
        PreSend()

        Pre-defined function to send a string to the user classifier system.  (May get
        abused by the user.)
        */

        PreSend(par)
                ParseThing * par;               /* function call in parse tree */
        {
                if ((Stack[FP+1].dummy->this != NULL)
                && (Stack[FP+1].dummy->this->type == ValSTRING))
                {
                        UserSend(Stack[FP+1].dummy->this->string);
                }
                else
                {
                        PrintLine();
                        fprintf(stderr,
                                "send failed: parameter must be a string: ");
                        PrintValue(stderr, Stack[FP+1].dummy, YES);
                        fprintf(stderr, "\n");
                        ExecAbort();
                }
        }


        /*
```

```
UserClose()

Tell the user classifier system to exit, and then close the pipe.
*/

UserClose()
{
        int status;                         /* status of called function */

        if ((UserReadFD < 0) && (UserWriteFD < 0))
                return;                     /* already closed */

        if (UserTrace)
                fprintf(stderr,
                        "trace: closing pipe to user classifier system\n");

        /* tell user classifer system to close up and exit */
        /* we don't care if there is a "ready" reply */

        UserSend("close");
        sleep(1);                           /* wait for child to exit */

        /* invalidate our file descriptors */

        status = close(UserReadFD);
        if (status < 0)
        {
                perror("UserClose close UserReadFD failed");
                exit(-1);
        }
        status = close(UserWriteFD);
        if (status < 0)
        {
                perror("UserClose close UserWriteFD failed");
                exit(-1);
        }

        UserReadFD = UserWriteFD = -1;
}


/*
UserDefine()

Define and initialize any functions to support the user classifer system.  (See
also PreDefine() in the "fapre.c" file.)
*/

UserDefine()
{
        SymbolThing * sym;                  /* a symbol table pointer */

        sym = GlobalAdd("close", SymFUNCTION);
        sym->count = 0;
        sym->special = SpeCLOSE;

        sym = GlobalAdd("messlist", SymGLOBAL);
        sym->dummy = MakeValue(ValDUMMY);
        sym->dummy->this = MakeValue(ValSET);
        sym->dummy->this->sign = 1;
        sym->special = MESSLIST;

        sym = GlobalAdd("flagmess", SymFUNCTION);
        sym->count = 0;
        sym->special = SpeFLAGMESS;

        sym = GlobalAdd("flagrule", SymFUNCTION);
        sym->count = 0;
        sym->special = SpeFLAGRULE;
```

```
        sym = GlobalAdd("open", SymFUNCTION);
        sym->count = 2;
        sym->special = SpeOPEN;

        sym = GlobalAdd("receive", SymFUNCTION);
        sym->count = 1;
        sym->special = SpeRECEIVE;

        sym = GlobalAdd("rulelist", SymGLOBAL);
        sym->dummy = MakeValue(ValDUMMY);
        sym->dummy->this = MakeValue(ValSET);
        sym->dummy->this->sign = 1;
        sym->special = RULELIST;

        sym = GlobalAdd("send", SymFUNCTION);
        sym->count = 1;
        sym->special = SpeSEND;
}


/*
UserError()

Print an unidentified string as an error message from the user classifier
system.
*/

UserError(cp)
        char * cp;                      /* a string pointer */
{
        fprintf(stderr, "classifier error: '%s'\n", cp);
}


/*
UserFetchList()

The caller must have sent a command to the user classifier asking for the
message or rule list.  We fetch all elements (either list), and attach a new
set structure to the symbol table entry.
*/

UserFetchList(sym)
        SymbolThing * sym;              /* a symbol table pointer */
{
        char * cp;                      /* string pointer */
        char * new;                     /* new string pointer */
        ValueThing * val;               /* a value structure pointer */

        /* free old value */

        FreeValue(sym->dummy->this);

        /* create a new value */

        sym->dummy->this = val = MakeValue(ValSET);
        val->sign = 1;

        while (YES)
        {
                cp = UserReceive();
                if (strcmp(cp, READY) == 0)
                        break;

                val->next = MakeValue(ValELEMENT);
                val = val->next;
                val->this = StringToValue(cp, cp, &new);

                if ((*new) != '\0')     /* did we use entire string? */
                        UserError(cp);  /* no */
```

```
		}
}


/*
UserFetchMess()

Fetch a new copy of the message list from the user classifier system.
*/

UserFetchMess(sym)
	SymbolThing * sym;			/* a symbol table pointer */
{
	/* reset fetch flag */

	UserFlagMess = NO;

	/* ask for the message list */

	UserSend("messlist");

	/* call common routine for fetching a set */

	UserFetchList(sym);
}


/*
UserFetchRule()

Fetch a new copy of the rule list from the user classifier system.
*/

UserFetchRule(sym)
	SymbolThing * sym;			/* a symbol table pointer */
{
	/* reset fetch flag */

	UserFlagRule = NO;

	/* ask for the rule list */

	UserSend("rulelist");

	/* call common routine for fetching a set */

	UserFetchList(sym);
}


/*
UserOpen()

Open a pipe to the user classifier system.  We use two UNIX pipes, because
it's simple, and it works.
*/

UserOpen(file, arg)
	char * file;				/* file name of user classifier */
	char * arg;				/* argument to user classifier */
{
	int fda[2], fdb[2];			/* pipe file descriptors */
	int status;				/* status of called function */

	if ((UserReadFD > 0) && (UserWriteFD > 0))
		return;				/* already open */

	if (UserTrace)
		fprintf(stderr,
			"trace: opening pipe to user classifier '%s'\n", file);
```

```

	/* create the necessary pipes (two) */

	status = pipe(fda);		/* get face-to-classifier pipe */
	if (status < 0)
	{
		perror("UserOpen pipe fda failed");
		exit(-1);
	}
	status = pipe(fdb);		/* get classifier-to-face pipe */
	if (status < 0)
	{
		perror("UserOpen pipe fdb failed");
		exit(-1);
	}

	/* fork a copy of "face" to become the classifier */

	status = fork();
	if (status < 0)
	{
		perror("UserOpen fork failed");
		exit(-1);
	}
	else if (status == 0)
	{
		/* child process */

		/* close unused ends of the pipes */

		status = close(fda[1]);
		if (status < 0)
		{
			perror("UserOpen close fda[1] failed");
			exit(-1);
		}
		status = close(fdb[0]);
		if (status < 0)
		{
			perror("UserOpen close fdb[0] failed");
			exit(-1);
		}

		/* replace standard input and output */

		close(0);			/* close stdin */
		status = dup(fda[0]);		/* replace with pipe */
		if (status < 0)
		{
			perror("UserOpen dup stdin failed");
			exit(-1),
		}
		close(1);			/* close stdout */
		status = dup(fdb[1]);		/* replace with pipe */
		if (status < 0)
		{
			perror("UserOpen dup stdout failed");
			exit(-1);
		}

		/* execute the user classifier system */
		/* if "execl" returns, it's an error */

		execl(file, file, arg, 0);
		perror("UserOpen execl failed");
		PrintLine();
		fprintf(stderr, "open failed: parameters were '%s' and '%s'\n",
			file, arg);
		exit(-1);
	}
```

```
	else
	{
		/* parent process */

		/* close unused ends of the pipes */

		status = close(fda[0]);
		if (status < 0)
		{
			perror("UserOpen close fda[0] failed");
			exit(-1);
		}
		status = close(fdb[1]);
		if (status < 0)
		{
			perror("UserOpen close fdb[1] failed");
			exit(-1);
		}

		UserReadFD = fdb[0];	/* our read file descriptor */
		UserWriteFD = fda[1];	/* our write file descriptor */

		sleep(1);		/* let child start running */

		/* UserSend("open"); */ /* tell child to open up */
		UserReady(READY);	/* should be ready now */
	}
}


/*
UserOpName()

The "name" executable operator found a global variable marked "special".  This
must be a special user classifier variable.  Re-fetch it from the classifier
system if necessary.  (A dummy stack element is ready for the value.)
*/

UserOpName(sym)
	SymbolThing * sym;		/* a symbol table pointer */
{
	/* only "messlist" and "rulelist" are legal now */

	switch (sym->special)
	{
	case (MESSLIST):
		if (UserFlagMess)
			UserFetchMess(sym);
		Stack[SP].dummy->this = sym->dummy->this;
		Stack[SP].free = NO;
		Stack[SP].owner = NULL;		/* not assignable */
		break;
	case (RULELIST):
		if (UserFlagRule)
			UserFetchRule(sym);
		Stack[SP].dummy->this = sym->dummy->this;
		Stack[SP].free = NO;
		Stack[SP].owner = NULL;		/* not assignable */
		break;
	default:
		PrintLine();
		fprintf(stderr,
			"internal error: UserOpName symbol special = %d\n",
			sym->special);
		ExecAbort();
		break;
	}
}
```

```
/*
UserReady()

Call this subroutine after sending a command to the user classifier system,
and the only acceptable response is "ready" (or whatever the final prompt
string is chosen to be).
*/

UserReady(string)
	char * string;			/* expected response string */
{
	char * cp;			/* a string pointer */

	while (YES)
	{
		cp = UserReceive();	/* receive a line */
		if (strcmp(cp, string) == 0)
			break;
		UserError(cp);		/* must be an error message */
	}
}


/*
UserReceive()

Receive a line from the user classifier system.  Return the address of the
string (in our static buffer) to the caller.  The string does not include the
newline character.
*/

char * UserReceive()
{
	char * cp;			/* a string pointer */
	int status;			/* status of called function */

	if (UserReadFD < 0)
		UserOpen(UserFile, UserArg);

	/* read the pipe, one byte at a time, until a newline */

	/* sleep(1); */
	cp = UserBuffer;		/* start here */
	while (YES)
	{
		status = read(UserReadFD, cp, 1);
		if (status != 1)
		{
			perror("UserReceive read failed");
			exit(-1);
		}
		if ((*cp) == '\n')
			break;
		else
			cp ++;
	}
	(*cp) = '\0';			/* terminate string with null */

	if (UserTrace)
		fprintf(stderr, "trace: received '%s'\n", UserBuffer);
	/* sleep(1); */

	/* return address of buffered string */

	return(UserBuffer);
}


/*
UserSave()
```

```

The pre-defined save() function found a global variable that it does not
understand (because the "special" flag is non-zero).  We convert this variable
into the proper format for a user classifier function call.
*/

UserSave(fp, sym)
	FILE * fp;			/* file pointer */
	SymbolThing * sym;		/* a symbol table pointer */
{
	char * fun;			/* pointer to function name */

	/* only "messlist" and "rulelist" are legal now */

	switch (sym->special)
	{
	case (MESSLIST):
		fun = "message";
		break;
	case (RULELIST):
		fun = "rule";
		break;
	default:
		PrintLine();
		fprintf(stderr,
			"internal error: UserSave symbol special = %d\n",
			sym->special);
		ExecAbort();
		break;
	}

	/* both "messlist" and "rulelist" must be sets */

	if ((sym->dummy == NULL)
	|| (sym->dummy->this == NULL)
	|| (sym->dummy->this->type != ValSET))
	{
		/* must be some mistake, save unchanged */

		fprintf(fp, "%s := ", sym->name);
		PrintValue(fp, sym->dummy, YES);
		fprintf(fp, "; # warning, expecting a set\n");
	}
	else
	{
		ValueThing * val;		/* pointer to set element */

		/* we have a legal set to work with */

		fprintf(fp, "\n# saving '%s'\n", sym->name);

		val = sym->dummy->this->next;
		while (val != NULL)
		{
			fprintf(fp, "%s(", fun);
			PrintValue(fp, val->this, YES);
			fprintf(fp, ");\n");
			val = val->next;
		}

		fprintf(fp, "# end of '%s'\n\n", sym->name);
	}
}


/*
UserSend()

Send a string followed by a newline to the user classifer system.  We add the
newline, not the caller.
```

```
*/

UserSend(string)
        char * string;                          /* some text string */
{
        int length;                             /* length of string */
        int status;                             /* status of called function */

        if (UserWriteFD < 0)
                UserOpen(UserFile, UserArg);

        /* sleep(1); */
        if (UserTrace)
                fprintf(stderr, "trace: sending '%s'\n", string);

        /* send the string, followed by a newline */

        length = strlen(string);
        status = write(UserWriteFD, string, length);
        if (status != length)
        {
                perror("UserSend write string failed");
                exit(-1);
        }
        status = write(UserWriteFD, "\n", 1);
        if (status != 1)
        {
                perror("UserSend write newline failed");
                exit(-1);
        }
        /* sleep(1); */
}
```

```
%{

/*

fayac.y -- YACC Grammar


Keith Fenske
Department of Computing Science
The University of Alberta
Edmonton, Alberta, Canada
T6G 2H1

December 1987

Copyright (c) 1987 by Keith Fenske.  All rights reserved.


This is the YACC grammar.  It has been greatly simplified by the following
assumptions:

This program will be used in an interactive environment.

Exactly one function definition or complete executable statement is parsed on
each call to yyparse().

When a syntax error is found, a message is printed, and all input is skipped
until the next semicolon (;).  At this point, it should be safe to call the
parser for the next statement.

Statements always end with a semicolon.  The semicolon is not used anywhere
else in the grammar.  Hence, YACC will safely parse a statement without
fetching a look-ahead token from LEX (which would be lost on the next call).

Anything and everything can be re-defined at any time.  There is no required
ordering for the input statements.  This means that the semantic actions can
not and should not do type checking.  Also, this removes the compile-time
distinction between functions and procedures.

Operators are assigned priorities with the %left and %right declarations.
This avoids having to separate the rules into long unambiguous productions
(as is typical in Pascal language definitions).

Assignments are just the lowest-priority operator.  They are implemented by
keeping track of the owner of each data value, so that one extra level of
indirection can always be removed.

The only control statements are "for", "if", "repeat", and "while".  Everything
else is done by pre-defined functions.

*/

#include "fainc.h"                          /* our standard includes */

        SymbolThing * sym;                  /* a symbol table pointer */
        SymbolThing * ThisFunction;         /* current function symbol */

%}

%start program

%token TokAND TokASSIGN TokBREAK TokBY TokDIV TokDO TokELIF TokELSE TokEND
%token TokERROR TokFOR TokFROM TokFUNCTION TokIF TokMOD
%token <string> TokNAME
%token TokNOT TokNULL
%token <number> TokNUMBER
%token TokOR TokPOWER
%token <count> TokRELOP
%token TokREPEAT TokRETURN
%token <string> TokSTRING
```

```
%token TokTO TokTHEN TokUNTIL TokWHILE

%right TokASSIGN
%left TokOR
%left TokAND
%left TokRELOP
%left '+' '-'
%left '*' '/' TokDIV TokMOD
%right TokNOT
%right TokPOWER
%right '['

%type <parse> expr expr_index expr_pars expr_plist expr_set expr_slist expr_term
%type <parse> for_by for_do for_from for_stmt for_to
%type <parse> func_body
%type <count> func_head func_pars
%type <parse> if_else if_stmt if_then
%type <parse> repeat_stmt
%type <parse> return_stmt
%type <parse> statement stmt_list
%type <parse> while_do while_stmt

%union {                                /* YACC stack value */
                int count;              /* if an integer */
                NUMBER number;          /* if a number */
                ParseThing * parse;     /* if a parse tree node */
                char * string;          /* if a string */
        }


%%


program         :
                        { /* prevents an error on end-of-file */ }
                | function ';'
                        { YYACCEPT; }
                | statement ';'
                        {
                        ParseTree = MakeParse(OpSTMT, NULL, $1, NULL, NULL);
                        YYACCEPT;
                        }
                | error err_program ';'
                        { YYABORT; }
                ;


function        : TokFUNCTION func_name func_head func_body TokEND
                        {
                        ThisFunction->count = $3;
                        ThisFunction->parse = $4;
                        LocalHead = LocalTail = NULL;
                        }
                | TokFUNCTION error err_function ';'
                        { YYABORT; }
                ;

func_name       : TokNAME
                        {
                        LocalHead = LocalTail = MakeSymbol(SymLOCAL);
                        LocalHead->name = CopyString($1);

                        ThisFunction = GlobalLook($1);
                        if (ThisFunction == NULL)
                                ThisFunction = GlobalAdd($1, SymFUNCTION);
                        else
                        {
                                /* free old definition (if any) */

                                FreeValue(ThisFunction->dummy);
```

```
                                FreeSymbol(ThisFunction->local);
                                FreeString(ThisFunction->name);
                                FreeParse(ThisFunction->parse);
                        }
                        ThisFunction->count = 0;
                        ThisFunction->dummy = NULL;
                        ThisFunction->free = YES;
                        ThisFunction->local = LocalHead;
                        ThisFunction->name = $1;
                        ThisFunction->offset = 0;
                        ThisFunction->parse = NULL;
                        ThisFunction->special = 0;
                        ThisFunction->type = SymFUNCTION;
                        }
                ;

func_head       :
                        { $$ = 0; }
                | '(' ')'
                        { $$ = 0; }
                | '(' func_pars ')'
                        { $$ = $2; }
                | '(' error err_func_paren ';'
                        { YYABORT; }
                ;

func_pars       : TokNAME
                        {
                        /* first parameter passed by value */
                        $$ = 1;
                        sym = LocalAdd($1);
                        sym->free = YES;
                        sym->offset = $$;
                        }
                | '*' TokNAME
                        {
                        /* first parameter passed by address */
                        $$ = 1;
                        sym = LocalAdd($2);
                        sym->free = NO;
                        sym->offset = $$;
                        }
                | '*' error err_func_star ';'
                        { YYABORT; }
                | func_pars ',' TokNAME
                        {
                        /* following parameter passed by value */
                        $$ = $1 + 1;
                        sym = LocalAdd($3);
                        sym->free = YES;
                        sym->offset = $$;
                        }
                | func_pars ',' '*' TokNAME
                        {
                        /* following parameter passed by address */
                        $$ = $1 + 1;
                        sym = LocalAdd($4);
                        sym->free = NO;
                        sym->offset = $$;
                        }
                | func_pars ',' '*' error err_func_star ';'
                        { YYABORT; }
                | func_pars ',' error err_func_comma ';'
                        { YYABORT; }
                ;

func_body       :
                        { $$ = NULL; }
                | TokASSIGN stmt_list
                        { $$ = $2; }
```

```
                | TokDO stmt_list
                        { $$ = $2; }
                ;


stmt_list       : statement
                        {
                        if ($1 == NULL)
                                $$ = NULL;
                        else
                                $$ = MakeParse(OpSTMT, NULL, $1, NULL, NULL);
                        }
                | stmt_list ';' statement
                        {
                        if ($1 == NULL)
                                $$ = $3;
                        else if ($3 == NULL)
                                $$ = $1;
                        else
                                $$ = MakeParse(OpSTMT, $1, $3, NULL, NULL);
                        }
                ;

statement       :
                        { $$ = NULL; }
                | expr
                        { $$ = $1; }
                | for_stmt
                        { $$ = $1; }
                | if_stmt
                        { $$ = $1; }
                | repeat_stmt
                        { $$ = $1; }
                | return_stmt
                        { $$ = $1; }
                | while_stmt
                        { $$ = $1; }
                ;


expr            : expr_term
                        { $$ = $1; }
                | TokNOT expr
                        { $$ = MakeParse(OpNOT, $2, NULL, NULL, NULL); }
                | TokNOT error err_expr_not ';'
                        { YYABORT; }
                | '+' expr                      %prec TokNOT
                        { $$ = $2; }
                | '+' error err_expr_plus ';'
                        { YYABORT; }
                | '-' expr                      %prec TokNOT
                        { $$ = MakeParse(OpNEGATE, $2, NULL, NULL, NULL); }
                | '-' error err_expr_minus ';'
                        { YYABORT; }
                | expr '*' expr
                        { $$ = MakeParse(OpSTAR, $1, $3, NULL, NULL); }
                | expr '*' error err_expr_star ';'
                        { YYABORT; }
                | expr '+' expr
                        { $$ = MakeParse(OpPLUS, $1, $3, NULL, NULL); }
                | expr '+' error err_expr_plus ';'
                        { YYABORT; }
                | expr '-' expr
                        { $$ = MakeParse(OpMINUS, $1, $3, NULL, NULL); }
                | expr '-' error err_expr_minus ';'
                        { YYABORT; }
                | expr '/' expr
                        { $$ = MakeParse(OpSLASH, $1, $3, NULL, NULL); }
                | expr '/' error err_expr_slash ';'
                        { YYABORT; }
```

```
                | expr TokAND expr
                        { $$ = MakeParse(OpAND, $1, $3, NULL, NULL); }
                | expr TokAND error err_expr_and ';'
                        { YYABORT; }
                | expr TokASSIGN expr
                        { $$ = MakeParse(OpASSIGN, $1, $3, NULL, NULL); }
                | expr TokASSIGN error err_expr_assign ';'
                        { YYABORT; }
                | expr TokDIV expr
                        { $$ = MakeParse(OpDIV, $1, $3, NULL, NULL); }
                | expr TokDIV error err_expr_div ';'
                        { YYABORT; }
                | expr TokMOD expr
                        { $$ = MakeParse(OpMOD, $1, $3, NULL, NULL); }
                | expr TokMOD error err_expr_mod ';'
                        { YYABORT; }
                | expr TokOR expr
                        { $$ = MakeParse(OpOR, $1, $3, NULL, NULL); }
                | expr TokOR error err_expr_or ';'
                        { YYABORT; }
                | expr TokPOWER expr
                        { $$ = MakeParse(OpPOWER, $1, $3, NULL, NULL); }
                | expr TokPOWER error err_expr_power ';'
                        { YYABORT; }
                | expr TokRELOP expr
                        { $$ = MakeParse($2, $1, $3, NULL, NULL); }
                | expr TokRELOP error err_expr_relop ';'
                        { YYABORT; }
                | '(' expr ')'
                        { $$ = $2; }
                | '(' error err_expr_paren ';'
                        { YYABORT; }
                | '{' expr_set '}'
                        { $$ = MakeParse(OpSET, $2, NULL, NULL, NULL); }
                | '{' error err_expr_set ';'
                        { YYABORT; }
                | expr_index ']'
                        { $$ = $1; }
                | expr_index error err_expr_index ';'
                        { YYABORT; }
                ;

expr_set        :
                        { $$ = NULL; }
                | expr_slist
                        { $$ = $1; }
                ;

expr_slist      : expr
                        { $$ = MakeParse(OpCONCAT, $1, NULL, NULL, NULL); }
                | expr ','
                        {
                        /* "{expr,}" must be the same as "{expr,NULL}" */
                        $$ = MakeParse(OpCONCAT, $1,
                                MakeParse(OpCONCAT,
                                        MakeParse(OpNULL, NULL, NULL, NULL, NULL),
                                        NULL, NULL, NULL),
                                NULL, NULL);
                        }
                | expr ',' expr_slist
                        { $$ = MakeParse(OpCONCAT, $1, $3, NULL, NULL); }
                | ',' expr_slist
                        {
                        /* "{,expr_slist}" must be "{NULL,expr_slist}" */
                        $$ = MakeParse(OpCONCAT,
                                MakeParse(OpNULL, NULL, NULL, NULL, NULL),
                                $2, NULL, NULL);
                        }
                | ','
                        {
```

```
                        /* "{,}" must be the same as "{NULL,NULL}" */
                        $$ = MakeParse(OpCONCAT,
                                MakeParse(OpNULL, NULL, NULL, NULL, NULL),
                                MakeParse(OpCONCAT,
                                        MakeParse(OpNULL, NULL, NULL, NULL, NULL),
                                        NULL, NULL, NULL),
                                NULL, NULL);
                        }
                ;

expr_index      : expr '[' expr
                        { $$ = MakeParse(OpINDEX, $1, $3, NULL, NULL); }
                | expr '[' expr ':' expr
                        { $$ = MakeParse(OpINDEX, $1, $3, $5, NULL); }
                | expr '[' error err_expr_index ';'
                        { YYABORT; }
                | expr_index ',' expr
                        { $$ = MakeParse(OpINDEX, $1, $3, NULL, NULL); }
                | expr_index ',' expr ':' expr
                        { $$ = MakeParse(OpINDEX, $1, $3, $5, NULL); }
                | expr_index ',' error err_expr_index ';'
                        { YYABORT; }
                ;

expr_term       : TokNAME
                        {
                        sym = LocalLook($1);
                        if (sym == NULL)
                                sym = GlobalLook($1);
                        if (sym == NULL)
                                sym = GlobalAdd($1, SymGLOBAL);
                        $$ = MakeParse(OpNAME, NULL, NULL, NULL, NULL);
                        $$->symbol = sym;
                        }
                | TokNAME '(' expr_pars ')'
                        {
                        sym = GlobalLook($1);
                        if (sym == NULL)
                                sym = GlobalAdd($1, SymGLOBAL);
                        $$ = MakeParse(OpFUNCTION, $3, NULL, NULL, NULL);
                        if ($3 != NULL)
                                $$->count = $3->count;
                        $$->symbol = sym;
                        }
                | TokNAME '(' error err_func_paren ';'
                        { YYABORT; }
                | TokNULL
                        { $$ = MakeParse(OpNULL, NULL, NULL, NULL, NULL); }
                | TokNUMBER
                        {
                        $$ = MakeParse(OpNUMBER, NULL, NULL, NULL, NULL);
                        $$->number = $1;
                        }
                | TokSTRING
                        {
                        $$ = MakeParse(OpSTRING, NULL, NULL, NULL, NULL);
                        $$->string = $1;
                        }
                ;

expr_pars       :
                        { $$ = NULL; }
                | expr_plist
                        { $$ = $1; }
                ;

expr_plist      : expr
                        {
                        $$ = MakeParse(OpPAR, NULL, $1, NULL, NULL);
                        $$->count = 1;
```

```
                        }
                | ',' expr
                        {
                        /* ",expr" is "NULL,expr" */
                        $$ = MakeParse(OpNULL, NULL, NULL, NULL, NULL);
                        $$ = MakeParse(OpPAR, NULL, $$, NULL, NULL);
                        $$->count = 1;
                        $$ = MakeParse(OpPAR, $$, $2, NULL, NULL);
                        $$->count = 2;
                        }
                | expr_plist ',' expr
                        {
                        $$ = MakeParse(OpPAR, $1, $3, NULL, NULL);
                        $$->count = $1->count + 1;
                        }
                | expr_plist ','
                        {
                        /* "expr_plist," is "expr_plist,NULL" */
                        $$ = MakeParse(OpPAR, $1,
                                MakeParse(OpNULL, NULL, NULL, NULL, NULL),
                                NULL, NULL);
                        $$->count = $1->count + 1;
                        }
                | ','
                        {
                        /* "," is "NULL,NULL" */
                        $$ = MakeParse(OpNULL, NULL, NULL, NULL, NULL);
                        $$ = MakeParse(OpPAR, NULL, $$, NULL, NULL);
                        $$->count = 1;
                        $$ = MakeParse(OpPAR, $$,
                                MakeParse(OpNULL, NULL, NULL, NULL, NULL),
                                NULL, NULL);
                        $$->count = 2;
                        }
                ;


for_stmt        : TokFOR TokNAME for_from for_to for_by for_do TokEND
                        {
                        sym = LocalLook($2);
                        if (sym == NULL)
                                sym = GlobalLook($2);
                        if (sym == NULL)
                                sym = GlobalAdd($2, SymGLOBAL);
                        $$ = MakeParse(OpFOR, $3, $4, $5, $6);
                        $$->symbol = sym;
                        }
                | TokFOR error err_for ';'
                        { YYABORT; }
                ;

for_from        :
                        { $$ = NULL; }
                | TokASSIGN expr
                        { $$ = $2; }
                | TokASSIGN error err_for_from ';'
                        { YYABORT; }
                | TokFROM expr
                        { $$ = $2; }
                | TokFROM error err_for_from ';'
                        { YYABORT; }
                ;

for_to          :
                        { $$ = NULL; }
                | TokTO expr
                        { $$ = $2; }
                | TokTO error err_for_to ';'
                        { YYABORT; }
                ;
```

```

for_by          :
                        { $$ = NULL; }
                | TokBY expr
                        { $$ = $2; }
                | TokBY error err_for_by ';'
                        { YYABORT; }
                ;

for_do          :
                        { $$ = NULL; }
                | TokDO stmt_list
                        { $$ = $2; }
                ;


if_stmt         : TokIF expr if_then if_else TokEND
                        { $$ = MakeParse(OpIF, $2, $3, $4, NULL); }
                | TokIF error err_if ';'
                        { YYABORT; }
                ;

if_then         :
                        { $$ = NULL; }
                | TokTHEN stmt_list
                        { $$ = $2; }
                ;

if_else         :
                        { $$ = NULL; }
                | TokELSE stmt_list
                        { $$ = $2; }
                | TokELIF expr if_then if_else
                        { $$ = MakeParse(OpIF, $2, $3, $4, NULL); }
                | TokELIF error err_if_elif ';'
                        { YYABORT; }
                ;


repeat_stmt     : TokREPEAT stmt_list TokUNTIL expr
                        { $$ = MakeParse(OpREPEAT, $2, $4, NULL, NULL); }
                | TokREPEAT stmt_list TokUNTIL error err_rep_until ';'
                        { YYABORT; }
                | TokREPEAT error err_repeat ';'
                        { YYABORT; }
                ;


return_stmt     : TokRETURN
                        { $$ = MakeParse(OpRETURN, NULL, NULL, NULL, NULL); }
                ;


while_stmt      : TokWHILE expr while_do TokEND
                        { $$ = MakeParse(OpWHILE, $2, $3, NULL, NULL); }
                | TokWHILE error err_while ';'
                        { YYABORT; }
                ;

while_do        :
                        { $$ = NULL; }
                | TokDO stmt_list
                        { $$ = $2; }
                ;


err_expr_and    :
                        { skippy("error after 'and' or '&' in expression"); }
                ;

```

```
err_expr_assign :
                        { skippy("error after ':=' in expression"); }
                ;

err_expr_div    :
                        { skippy("error after 'div' in expression"); }
                ;

err_expr_index  :
                        { skippy("error after '[' or ',' in subscripted expression"); }
                ;

err_expr_minus  :
                        { skippy("error after '-' in expression"); }
                ;

err_expr_mod    :
                        { skippy("error after 'mod' or '%' in expression"); }
                ;

err_expr_not    :
                        { skippy("error after 'not' or '¬' in expression"); }
                ;

err_expr_or     :
                        { skippy("error after 'or' or '|' in expression"); }
                ;

err_expr_paren  :
                        { skippy("error after '(' in expression"); }
                ;

err_expr_plus   :
                        { skippy("error after '+' in expression"); }
                ;

err_expr_power  :
                        { skippy("error after '**' or '^' in expression"); }
                ;

err_expr_relop  :
                        { skippy("error after relational operator in expression"); }
                ;

err_expr_set    :
                        { skippy("error after '{' in set expression"); }
                ;

err_expr_slash  :
                        { skippy("error after '/' in expression"); }
                ;

err_expr_star   :
                        { skippy("error after '*' in expression"); }
                ;

err_for         :
                        { skippy("error after 'for' in for statement"); }
                ;

err_for_by      :
                        { skippy("error after 'by' in for statement"); }
                ;

err_for_from    :
                        { skippy("error after 'from' in for statement"); }
                ;

err_for_to      :
                        { skippy("error after 'to' in for statement"); }
```

```
                    ;

err_function        :
                            { skippy("error after 'function' in function definition"); }
                    ;

err_func_comma      :
                            { skippy("error after ',' in function parameters"); }
                    ;

err_func_paren      :
                            { skippy("error after '(' in function parameters"); }
                    ;

err_func_star       :
                            { skippy("error after '*' in function parameters"); }
                    ;

err_if              :
                            { skippy("error after 'if' in if statement"); }
                    ;

err_if_elif         :
                            { skippy("error after 'elif' in if statement"); }
                    ;

err_program         :
                            { skippy("input must be an executable statement or function definit
                    ;

err_repeat          :
                            { skippy("error after 'repeat' in repeat statement"); }
                    ;

err_rep_until       :
                            { skippy("error after 'until' in repeat statement"); }
                    ;

err_while           :
                            { skippy("error after 'while' in while statement"); }
                    ;


%%


/*
skippy()

Because of a syntax error, we will now start skipping until we find a semicolon
in the input.  The caller gives us an error message to print first.
*/

skippy(cp)
        char * cp;              /* character string pointer */
{
        yyerror(cp);
        yyerror("skipping to next semicolon ';'");
}


/*
yyerror()

Print a YACC-generated error message.
*/

yyerror(string)
        char * string;          /* some error message */
{
```

```
        PrintLine();
        fprintf(stderr, "%s\n", string);
}
```

```
/*

kpr.c -- Keith's Print Program


Keith Fenske
Department of Computing Science
The University of Alberta
Edmonton, Alberta, Canada
T6G 2H1

December 1987

Copyright (c) 1987 by Keith Fenske.  All rights reserved.


Print a file listing in a format acceptable to the Faculty of Graduate Studies
and Research.  This is similar to the standard UNIX "pr" utility, but has fewer
options:

        -n
                number the lines.  Default is no line numbers.

        -p<number>
                first page number.  Default is 1.

        -s
                skip to a new sheet of paper for each file.  Default is to skip
                only to the next side.

All other arguments must be file names.  The output from this program should be
fed directly into "mpr" without running "xp" first.  The following MTS carriage
control characters are used (first column):

        ;       skip to physical top of next page
        9       single spacing, ignore MTS lines per page count

*/

#include <stdio.h>                      /* standard I/O */
#include <sys/types.h>                  /* for last modify time */
#include <sys/stat.h>                   /* for last modify time */
#include <time.h>                       /* time buffers */

#define PAGEGAP 0                       /* lines before page number */
#define PAGELEFT 93                     /* spaces before page number */
#define TITLEGAP 1                      /* lines after page before title */
#define TITLELEFT 32                    /* spaces before title */
#define TEXTGAP 2                       /* lines after title before text */
#define TEXTLEFT 6                      /* spaces before text */
#define TEXTLINES 70                    /* text lines per page */


/* global variables */

        int NumberFlag = 0;             /* non-zero if line numbers */
        int PageNumber = 1;             /* first page number */
        int SkipFlag = 0;               /* non-zero if skip sheet */


/* main program */

main(argc, argv)
        int argc;                       /* number of arguments */
        char * argv[];                  /* argument strings */
{
        int i;                          /* index variable */

        /* set page printer font and format */

```

```
	printf("$**$ OVERLAY=NONE\n");
	printf("$**$ FORMAT=FMTGH1 FONTNEXTIMAGE=MEDIUM SKIPTO=NEXTSHEET\n");

	/* process command line arguments */

	for (i = 1 ; i < argc ; i ++)
	{
		if (argv[i][0] == '-')
		{
			switch (argv[i][1])
			{
			case ('n'):
			case ('N'):
				NumberFlag = 1;
				break;
			case ('p'):
			case ('P'):
				PageNumber = atoi(&argv[i][2]);
				break;
			case ('s'):
			case ('S'):
				SkipFlag = 1;
				break;
			default:
				fprintf(stderr, "kpr: unknown flag: '%s'\n",
					argv[i]);
				exit(-1);
				break;
			}
		}
		else
		{
			ListFile(argv[i]);
		}
	}
}


/*
ListFile()

Given a file name, list the contents of the file.
*/

ListFile(name)
	char * name;			/* file name */
{
	int c;				/* input character */
	int column;			/* current output column (tabs) */
	FILE * fp;			/* file pointer */
	int i, j;			/* index variables */
	int line;			/* file line number */
	struct stat modify;		/* last modify buffer */
	int page;			/* local page number */
	int status;			/* status of called function */
	char time_day[9];		/* time string buffers */
	char time_month[9];
	char time_time[9];
	char time_week[9];
	char time_year[9];
	char time[99];			/* last modify time string */

	/* open file */

	fp = fopen(name, "r");
	if (fp == NULL)
	{
		fprintf(stderr, "kpr: can't open '%s' for reading\n", name);
		return;
	}
```

```

	/* get last modify time and reformat it */

	status = stat(name, &modify);
	if (status != 0)
	{
		fprintf(stderr, "kpr: can't get file status for '%s'\n", name);
		fclose(fp);
		return;
	}
	sscanf(asctime(localtime(&modify.st_mtime)), "%s %s %s %s %s",
		time_week, time_month, time_day, time_time, time_year);
	sprintf(time, "%s %s %s %s", time_week, time_day, time_month,
		time_year);

	/* skip to a new sheet of paper? */

	if (SkipFlag)
		printf("$**$ SKIPTO=NEXTSHEET\n");

	/* do the pages of text */

	line = page = 1;		/* local line and page numbers */

	while (!feof(fp))
	{
		/* start a new page */

		printf(";\n");

		/* do the page nu~ber */

		SkipLines(PAGEGAP);
		printf("9");
		SkipSpaces(PAGELEFT);
		printf("%4d\n", PageNumber);

		/* do the title */

		SkipLines(TITLEGAP);
		printf("9");
		SkipSpaces(TITLELEFT);
		printf("%s     %s     page %d\n", time, name, page);

		/* do enough text lines to fill this page */

		SkipLines(TEXTGAP);

		for (i = 0 ; i < TEXTLINES ; i ++)
		{
			if (feof(fp))		/* end-of-file? */
				break;

			column = 0;		/* starting column */

			do
			{
				c = fgetc(fp);

				if ((c == EOF) || (c == '\f'))
				{
					if (column > 0)
						putchar('\n');
					break;
				}

				if (column == 0)
				{
					printf("9");
					SkipSpaces(TEXTLEFT);
```

```
                                        if (NumberFlag)
                                                printf("%5d   ", line);
                                        else
                                                SkipSpaces(8);
                                }

                                if (c == 0x07)          /* bell */
                                {
                                        putchar(0xf0);  /* translate */
                                        column ++;
                                }
                                else if (c == '\t')     /* tab */
                                {
                                        j = 8 - (column % 8);
                                        SkipSpaces(j);
                                        column += j;
                                }
                                else if (c == '^')      /* circumflex */
                                {
                                        putchar(0xbb);  /* translate */
                                        column ++;
                                }
                                else if (c == '`')      /* grave */
                                {
                                        putchar(0xd3);  /* translate */
                                        column ++;
                                }
                                else if (c == 0x7f)     /* delete */
                                {
                                        putchar(0xd0);  /* translate */
                                        column ++;
                                }
                                else
                                {
                                        putchar(c);
                                        column ++;
                                }
                        }
                        while (c != '\n');

                        if ((c == EOF) || (c == '\f'))
                                break;

                        line ++;
                }

                /* increment page number */

                page ++;                        /* local page number */
                PageNumber ++;                  /* global page number */
        }

        /* close file */

        fclose(fp);
}


/*
SkipLines()

The caller tells us how many blank lines to put in the output.
*/

SkipLines(count)
        int count;                              /* number of blank lines */
{
        int k;                                  /* index variable */

        for (k = 0 ; k < count ; k ++)
```

```
                printf("9\n");
}


/*
SkipSpaces()

The caller tells us how many blank spaces to put in the output.
*/

SkipSpaces(count)
        int count;                      /* number of blank spaces */
{
        int k;                          /* index variable */

        for (k = 0 ; k < count ; k ++)
                printf(" ");
}
```

```c
/*

telex.c -- Test Lexical Routines


Keith Fenske
Department of Computing Science
The University of Alberta
Edmonton, Alberta, Canada
T6G 2H1

December 1987

Copyright (c) 1987 by Keith Fenske.  All rights reserved.


This is a quick-and-dirty program to test the lexical routines.  As you type
tokens, this program will tell you the token number, and possibly the value
(if it recognizes the token).

*/

#include <ctype.h>                          /* character types */
#include "fainc.h"                          /* our standard includes */
#include "y.tab.h"                          /* YACC token definitions */

        YYSTYPE yylval;                     /* where LEX returns information */

main()
{
        int token;                          /* token returned by LEX */

        do
        {
                token = yylex();            /* get a token */
                if (token == TokAND)
                        printf("token AND\n");
                else if (token == TokASSIGN)
                        printf("token ASSIGN\n");
                else if (token == TokBREAK)
                        printf("token BREAK\n");
                else if (token == TokBY)
                        printf("token BY\n");
                else if (token == TokDIV)
                        printf("token DIV\n");
                else if (token == TokDO)
                        printf("token DO\n");
                else if (token == TokELIF)
                        printf("token ELIF\n");
                else if (token == TokELSE)
                        printf("token ELSE\n");
                else if (token == TokEND)
                        printf("token END\n");
                else if (token == TokERROR)
                        printf("token ERROR\n");
                else if (token == TokFOR)
                        printf("token FOR\n");
                else if (token == TokFROM)
                        printf("token FROM\n");
                else if (token == TokFUNCTION)
                        printf("token FUNCTION\n");
                else if (token == TokIF)
                        printf("token IF\n");
                else if (token == TokMOD)
                        printf("token MOD\n");
                else if (token == TokNAME)
                        printf("token NAME = '%s' at %x\n", yylval.string,
                                yylval.string);
                else if (token == TokNOT)
                        printf("token NOT\n");
```

```
		else if (token == TokNULL)
			printf("token NULL\n");
		else if (token == TokNUMBER)
		{
			printf("token NUMBER = ");
			printf(FORMAT, yylval.number);
			printf("\n");
		}
		else if (token == TokOR)
			printf("token OR\n");
		else if (token == TokPOWER)
			printf("token POWER\n");
		else if (token == TokRELOP)
		{
			int op;
			op = yylval.count;
			printf("token RELOP = ");
			if (op == OpEQ)
				printf("OpEQ\n");
			else if (op == OpEQP)
				printf("OpEQP\n");
			else if (op == OpGE)
				printf("OpGE\n");
			else if (op == OpGT)
				printf("OpGT\n");
			else if (op == OpLE)
				printf("OpLE\n");
			else if (op == OpLT)
				printf("OpLT\n");
			else if (op == OpNE)
				printf("OpNE\n");
			else if (op == OpNEP)
				printf("OpNEP\n");
			else
				printf("unknown %d\n", op);
		}
		else if (token == TokREPEAT)
			printf("token REPEAT\n");
		else if (token == TokRETURN)
			printf("token RETURN\n");
		else if (token == TokSTRING)
			printf("token STRING = '%s' at %x\n", yylval.string,
				yylval.string);
		else if (token == TokTO)
			printf("token TO\n");
		else if (token == TokTHEN)
			printf("token THEN\n");
		else if (token == TokUNTIL)
			printf("token UNTIL\n");
		else if (token == TokWHILE)
			printf("token WHILE\n");
		else if (isascii(token) && isprint(token))
			printf("token character '%c'\n", token);
		else
			printf("token decimal %d\n", token);
	}
	while (token > 0);
	printf("\nend-of-file received from LEX\n");
}
```

```
/*

tepar.c -- Test Parse Routines


Keith Fenske
Department of Computing Science
The University of Alberta
Edmonton, Alberta, Canada
T6G 2H1

December 1987

Copyright (c) 1987 by Keith Fenske.  All rights reserved.


This is a quick-and-dirty routine to dump the parse tree built up by the YACC
grammar.  It replaces the "main()" routine normally found in "face.c".

*/

#include "fainc.h"                      /* our standard includes */


/*
main()

Fake main program to keep calling YACC until an end-of-file.
*/

main()
{
        int flag;                       /* status flag */

        do
        {
                ParseTree = NULL;

                printf("\ncalling yyparse()\n");
                flag = yyparse();
                printf("\nyyparse() returns %d\n", flag);

                if (ParseTree == NULL)
                        printf("ParseTree is NULL\n");
                else
                        DumpParse(ParseTree, 0);
        }
        while (!feof(stdin));

        printf("\nend-of-file on standard input\n");
}


/*
DumpParse()

Dump a parse tree in a crude indented format.
*/

DumpParse(parse, level)
        ParseThing * parse;             /* pointer to a parse tree */
        int level;                      /* indenting level */
{
        int i;                          /* index variable */

        for (i = 0 ; i < level ; i ++)
                printf("  ");
        printf("at %x ", parse);

        if (parse->type == OpAND)
```

```
			printf("AND");
		else if (parse->type == OpASSIGN)
			printf("ASSIGN");
		else if (parse->type == OpCONCAT)
			printf("CONCAT");
		else if (parse->type == OpDIV)
			printf("DIV");
		else if (parse->type == OpEQ)
			printf("EQ");
		else if (parse->type == OpEQP)
			printf("EQP");
		else if (parse->type == OpFOR)
			printf("FOR");
		else if (parse->type == OpFUNCTION)
			printf("FUNCTION");
		else if (parse->type == OpGE)
			printf("GE");
		else if (parse->type == OpGT)
			printf("GT");
		else if (parse->type == OpIF)
			printf("IF");
		else if (parse->type == OpINDEX)
			printf("INDEX");
		else if (parse->type == OpLE)
			printf("LE");
		else if (parse->type == OpLT)
			printf("LT");
		else if (parse->type == OpMINUS)
			printf("MINUS");
		else if (parse->type == OpMOD)
			printf("MOD");
		else if (parse->type == OpNAME)
			printf("NAME");
		else if (parse->type == OpNE)
			printf("NE");
		else if (parse->type == OpNEGATE)
			printf("NEGATE");
		else if (parse->type == OpNEP)
			printf("NEP");
		else if (parse->type == OpNOT)
			printf("NOT");
		else if (parse->type == OpNULL)
			printf("NULL");
		else if (parse->type == OpNUMBER)
			printf("NUMBER");
		else if (parse->type == OpOR)
			printf("OR");
		else if (parse->type == OpPAR)
			printf("PAR");
		else if (parse->type == OpPLUS)
			printf("PLUS");
		else if (parse->type == OpPOWER)
			printf("POWER");
		else if (parse->type == OpREPEAT)
			printf("REPEAT");
		else if (parse->type == OpRETURN)
			printf("RETURN");
		else if (parse->type == OpSET)
			printf("SET");
		else if (parse->type == OpSLASH)
			printf("SLASH");
		else if (parse->type == OpSTAR)
			printf("STAR");
		else if (parse->type == OpSTMT)
			printf("STMT");
		else if (parse->type == OpSTRING)
			printf("STRING");
		else if (parse->type == OpWHILE)
			printf("WHILE");
		else
```

```
			printf("decimal %d", parse->type);

		if (parse->one != NULL)
			printf(" one = %x", parse->one);
		if (parse->two != NULL)
			printf(" two = %x", parse->two);
		if (parse->three != NULL)
			printf(" three = %x", parse->three);
		if (parse->four != NULL)
			printf(" four = %x", parse->four);
		if (parse->count != 0)
			printf(" count = %d", parse->count);
		if (parse->number != ZERO)
		{
			printf(" number = ");
			printf(FORMAT, parse->number);
		}
		if (parse->string != NULL)
			printf(" string = '%s'", parse->string);
		if (parse->symbol != NULL)
			printf(" symbol = %x '%s'", parse->symbol, parse->symbol->name);
		printf("\n");

		if (parse->one != NULL)
			DumpParse(parse->one, (level+1));
		if (parse->two != NULL)
			DumpParse(parse->two, (level+1));
		if (parse->three != NULL)
			DumpParse(parse->three, (level+1));
		if (parse->four != NULL)
			DumpParse(parse->four, (level+1));
}
```

```c
/*

terob.c -- Test Robert Program


Keith Fenske
Department of Computing Science
The University of Alberta
Edmonton, Alberta, Canada
T6G 2H1

December 1987

Copyright (c) 1987 by Keith Fenske.  All rights reserved.


This is a dummy program which is connected to the user classifier end of a
pipe to dump exactly what the "face" program is sending to the classifier
system.  This program reads a line from standard input, traces it onto the
terminal, repeatedly asks you for what to send back (until you type the magic
word "ready"), and then waits to read more input from standard input.

(The name "terob" comes from "test Robert", where "Robert" is Robert Andrew
Chai, who is writing the real user classifier machine.)

*/

#include "fainc.h"                      /* our standard includes */

#define READY "ready"                   /* user classifier prompt for input */


main(argc, argv)
        int argc;                       /* number of arguments */
        char * argv[];                  /* argument strings */
{
        char buffer[MAXSTRING+1];       /* input/output buffer */
        int i;                          /* index variable */
        FILE * ttin;                    /* input from terminal */
        FILE * ttout;                   /* output to terminal */

        ttin = fopen("/dev/tty", "r");
        if (ttin == NULL)
        {
                fprintf(stderr, "can't open /dev/tty for input\n");
                exit(-1);
        }

        ttout = fopen("/dev/tty", "w");
        if (ttout == NULL)
        {
                fprintf(stderr, "can't open /dev/tty for output\n");
                exit(-1);
        }

        /* introduce ourself and echo command line arguments */

        fprintf(ttout, "\nterob: running\n");
        for (i = 0 ; i < argc ; i ++)
        {
                fprintf(ttout, "terob: argument #%d is '%s'\n", i, argv[i]);
        }
        fflush(ttout);

        /* main loop */

        while ((!feof(stdin)) && (!feof(ttin)))
        {
                fprintf(ttout, "terob: sending '%s'\n", READY);
                fflush(ttout);
```

```
                fprintf(stdout, "%s\n", READY);
                fflush(stdout);

                gets(buffer);           /* drops newline */
                if (feof(stdin))
                        break;

                fprintf(ttout, "terob: received '%s'\n", buffer);
                fflush(ttout);

                if (strcmp(buffer, "close") == 0)
                        break;

                while (YES)
                {
                        fprintf(ttout, "terob: send what reply? ");
                        fflush(ttout);

                        fscanf(ttin, " %[^\n]s", buffer);

                        if (feof(ttin))
                                break;

                        if (strcmp(buffer, READY) == 0)
                                break;

                        fprintf(ttout, "terob: sending '%s'\n", buffer);
                        fflush(ttout);
                        fprintf(stdout, "%s\n", buffer);
                        fflush(stdout);
                }
                if (feof(ttin))
                        break;
        }

        fprintf(ttout, "\nterob: exiting\n");
        fflush(ttout);

        fclose(ttin);
        fclose(ttout);
}
```

```
#
# pretty.f
#
#
# Keith Fenske
# Department of Computing Science
# The University of Alberta
# Edmonton, Alberta, Canada
# T6G 2H1
#
# December 1987
#
#
# This "face" function prints a value in a pretty format.  The first
# parameter must be NULL, a number, a set, or a string.  The second
# parameter should have the same set structure as the first parameter,
# but with one extra level of sets to provide a mapping from "value"
# elements to name strings.
#
# For example, suppose that there is a field in a classifier message
# that has the string "00" for NO, "11" for YES, and "01" for MAYBE.
# Then a mapping picture would be:
#
#       picture := {"00", "NO", "11", "YES", "01", "MAYBE"};
#
# The function call:
#
#       pretty("00", picture);
#
# would print "NO" (without quotes or newlines).  The function call:
#
#       pretty("#1", picture);
#
# would print "(YES or MAYBE)".  Finally, the function call:
#
#       pretty("10", picture);
#
# would print "[10]" since there is no legal mapping.
#
# (This function is recursively defined for sets.  The caller is
# responsible for writing any newlines before or after the "pretty"
# output.  Bad parameters will result in obscure error messages.)
#

function pretty(value, picture,          # real parameters
                found, i)                # local variables
do

# if "value" is a set, then call ourself for each element

        if type(value, "set")
        then
                if sign(value) < 0
                then write("-");
                end;

                write("{");
                for i from 1 to size(value)
                do
                        if i > 1
                        then write(", ");
                        end;

                        pretty(value[i], picture[i]);
                end;
                write("}");

# else look for an identical mapping (no pattern matching)

        else
```

```
                    found := 0;
                    for i from 1 to size(picture) by 2
                    do
                            if value = picture[i]
                            then
                                    if found > 0
                                    then write(" and ");
                                    end;

                                    write(picture[i+1]);

                                    found := found + 1;
                            end;
                    end;

# try pattern matching if nothing identical was found

                    if found = 0
                    then
                            for i from 1 to size(picture) by 2
                            do
                                    if value eqp picture[i]
                                    then
                                            if found = 0
                                            then write("(");
                                            else write(" or ");
                                            end;

                                            write(picture[i+1]);

                                            found := found + 1;
                                    end;
                            end;

                            if found = 0
                            then write("[", value, "]");
                            else write(")");
                            end;
                    end;
            end;
end;
```

```
#
# user.f
#
#
# Keith Fenske
# Department of Computing Science
# The University of Alberta
# Edmonton, Alberta, Canada
# T6G 2H1
#
# December 1987
#
#
# These functions provide most of the support for Robert Chai's
# classifier system.  Since they are user-defined, they are easy to
# change.
#
# close(), flagmess(), flagrule(), open(), receive(), and send() are
# written in "C" in the "fause.c" module.
#


# clear ( )

function clear
do
        flagmess();                 # messlist is invalid
        flagrule();                 # rulelist is invalid
        send("clear");
        receive("ready");
end;


# crossover ( )

function crossover
do
        flagrule();                 # rulelist is invalid
        send("crossover");
        receive("ready");
end;


# generate ( )

function generate
do
        flagmess();                 # messlist is invalid
        flagrule();                 # rulelist is invalid
        send("generate");
        receive("ready");
end;


# invert ( )

function invert
do
        flagrule();                 # rulelist is invalid
        send("invert");
        receive("ready");
end;


# message ( string )

function message ( string ,         # string parameter
                   buffer )         # local string buffer
do
        flagmess();                 # messlist is invalid
```

```

        buffer := "message " + pack(string);
        send(buffer);
        receive("ready");
end;


# mutate ( )

function mutate
do
        flagrule();                 # rulelist is invalid
        send("mutate");
        receive("ready");
end;


# payoff ( number )

function payoff ( number ,          # number parameter
                  buffer )          # local string buffer
do
        flagrule();                 # rulelist is invalid

        buffer := "payoff " + pack(number);
        send(buffer);
        receive("ready");
end;


# rule ( set )

function rule ( set ,               # set parameter
                buffer ,            # local string buffer
                conditions ,        # local set of conditions
                i )                 # local index variable
do
        flagrule();                 # rulelist is invalid

        if type(set) ne "set"
        then
                write("rule failed: bad set parameter: ");
                set;
                stop();
        end;

        # do condition strings

        buffer := "rule ";

        conditions := set[1];
        if type(conditions, "set")
        then
                for i from 1 to size(conditions)
                do
                        if i > 1
                        then buffer := buffer + " , ";
                        end;

                        buffer := buffer + pack(conditions[i]);
                end;
        else
                buffer := buffer + pack(conditions);
        end;

        # do action part

        buffer := buffer + " / " + pack(set[2]);

        # do any remaining elements
```

```
        for i from 3 to size(set)
        do
                buffer := buffer + " " + pack(set[i]);
        end;

        # send to classifier system

        send(buffer);
        receive("ready");
end;


# switch ( number )

function switch ( number ,         # number parameter
                buffer )           # local string buffer
do
        flagmess();                # messlist is invalid
        flagrule();                # rulelist is invalid

        if type(number, "null")
        then switchnumber := 1;
        else
                if type(number, "number")
                then switchnumber := number;
                else
                        write("switch failed: bad number parameter: ");
                        number;
                        stop();
                end;
        end;

        buffer := "switch " + pack(switchnumber);
        send(buffer);
        receive("ready");
end;

switchnumber := 1;                 # initial value
```

```
#
# prime.f
#
#
# Keith Fenske
# Department of Computing Science
# The University of Alberta
# Edmonton, Alberta, Canada
# T6G 2H1
#
# December 1987
#
#
# Find all prime numbers from 1 to 100.  This is used to collect data
# about CPU times for the various operators -- hence this program is
# not written to be "efficient".  Yes, I know that 1 is not really a
# prime number!
#

primes := {};

for i from 1 to 100
do
        j := true;
        k := 2;
        while (j and (k <= size(primes)))
        do
                if i mod primes[k] eq 0
                then j := false;
                end;
                k := k + 1;
        end;

        if (j)
        then
                write(i, " is a prime number\n");
                primes := primes + {i};
        end;
end;

primes;
```

```
Script started on Tue Dec 22 19:02:41 1987

cavell fenske % face prime.f

face da class: an interactive classifier programming language

loading file 'prime.f'

1 is a prime number
2 is a prime number
3 is a prime number
5 is a prime number
7 is a prime number
11 is a prime number
13 is a prime number
17 is a prime number
19 is a prime number
23 is a prime number
29 is a prime number
31 is a prime number
37 is a prime number
41 is a prime number
43 is a prime number
47 is a prime number
53 is a prime number
59 is a prime number
61 is a prime number
67 is a prime number
71 is a prime number
73 is a prime number
79 is a prime number
83 is a prime number
89 is a prime number
97 is a prime number

{1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97}

end-of-file on file 'prime.f'

ready for input

exit();

exit() called; returning to UNIX

cavell fenske % exit

script done on Tue Dec 22 19:03:06 1987
```

call graph profile:
The sum of self and descendents is the major sort for this listing.

function entries:

index the index of the function in the call graph listing, as an aid to locating it (see below).

%time the percentage of the total time of the program accounted for by this function and its descendents.

self the number of seconds spent in this function itself.

descendents
the number of seconds spent in the descendents of this function on behalf of this function.

called the number of times this function is called (other than recursive calls).

self the number of times this function calls itself recursively.

name the name of the function, with an indication of its membership in a cycle, if any.

index the index of the function in the call graph listing, as an aid to locating it.

parent listings:

self* the number of seconds of this function's self time which is due to calls from this parent.

descendents*
the number of seconds of this function's descendent time which is due to calls from this parent.

called** the number of times this function is called by this parent. This is the numerator of the fraction which divides up the function's time to its parents.

total* the number of times this function was called by all of its parents. This is the denominator of the propagation fraction.

parents the name of this parent, with an indication of the parent's membership in a cycle, if any.

index the index of this parent in the call graph listing, as an aid in locating it.

children listings:

self* the number of seconds of this child's self time which is due to being called by this function.

descendent*

the number of seconds of this child's descendent's time which is due to being called by this function.

called** the number of times this child is called by this function. This is the numerator of the propagation fraction for this child.

total* the number of times this child is called by all functions. This is the denominator of the propagation fraction.

children the name of this child, and an indication of its membership in a cycle, if any.

index the index of this child in the call graph listing, as an aid to locating it.

* these fields are omitted for parents (or children) in the same cycle as the function. If the function (or child) is a member of a cycle, the propagated times and propagation denominator represent the self time and descendent time of the cycle as a whole.

** static-only parents and children are indicated by a call count of 0.

cycle listings:
the cycle as a whole is listed with the same fields as a function entry. Below it are listed the members of the cycle, and their contributions to the time and call counts of the cycle.

```
granularity: each sample hit covers 4 byte(s) for 0.22% of 4.56 seconds

                                       called/total       parents
index  %time    self descendents  called+self    name           index
                                       called/total       children

                                                     <spontaneous>
[1]     27.7    1.26        0.00                 _ExecParse [1]
                0.00        0.00       1/108         _fprintf [9]

-----------------------------------------------

                                                     <spontaneous>
[2]     12.7    0.58        0.00                 _CheckStack [2]

-----------------------------------------------

                                                     <spontaneous>
[3]      9.4    0.14        0.29                 _GetMemory [3]
                0.25        0.04    4809/4812        _malloc [7]

-----------------------------------------------

                                                     <spontaneous>
[4]      9.4    0.30        0.13                 _FreeValue [4]
                0.13        0.00    4590/4681        _free [12]

-----------------------------------------------

                                                     <spontaneous>
[5]      8.8    0.40        0.00                 _PopStack [5]

-----------------------------------------------

                                                     <spontaneous>
[6]      7.9    0.36        0.00                 _PushStack [6]

-----------------------------------------------

                0.00        0.00       1/4812        __flsbuf [13]
                0.00        0.00       2/4812        __filbuf [43]
                0.25        0.04    4809/4812        _GetMemory [3]
[7]      6.4    0.25        0.04    4812         _malloc [7]
                0.03        0.01      15/15          _morecore [23]
                0.00        0.00       2/17          _sbrk [38]
                0.00        0.00       1/1           _getpagesize [92]

-----------------------------------------------

                0.00        0.01       5/113         _printf [33]
                0.08        0.11     108/113         _fprintf [9]
[8]      4.3    0.08        0.11     113         __doprnt [8]
                0.01        0.10      38/38          __flsbuf [13]

-----------------------------------------------

                0.00        0.00       1/108         _ExecParse [1]
                0.00        0.00       2/108         _PreExit [42]
                0.00        0.05      26/108         _PrintString [22]
                0.00        0.14      79/108         _PrintValue [11]
[9]      4.1    0.00        0.19     108         _fprintf [9]
                0.08        0.11     108/113         __doprnt [8]
                0.00        0.00       2/4           _fflush [41]

-----------------------------------------------

                                                     <spontaneous>
[10]     3.1    0.14        0.00                 _MakeValue [10]
```

```
-----------------------------------------------

                                                 <spontaneous>
[11]     3.0    0.00        0.14               _PrintValue [11]
                0.00        0.14      79/108        _fprintf [9]

-----------------------------------------------

                0.00        0.00       1/4681       _fclose [36]
                0.00        0.00      27/4681       _FreeString [20]
                0.00        0.00      63/4681       _FreeParse [35]
                0.13        0.00    4590/4681       _FreeValue [4]
[12]     2.9    0.13        0.00    4681           _free [12]

-----------------------------------------------

                0.01        0.10      38/38         __doprnt [8]
[13]     2.5    0.01        0.10      38           __flsbuf [13]
                0.10        0.00      36/38         _write [14]
                0.00        0.00       1/4812       _malloc [7]
                0.00        0.00       1/3          _fstat [88]
                0.00        0.00       1/1          _isatty [95]

-----------------------------------------------

                0.01        0.00       2/38         _fflush [41]
                0.10        0.00      36/38         __flsbuf [13]
[14]     2.4    0.11        0.00      38           _write [14]

-----------------------------------------------

                                                 <spontaneous>
[15]     2.2    0.10        0.00               _ExecFunction [15]

-----------------------------------------------

                                                 <spontaneous>
[16]     2.0    0.09        0.00               _PreSize [16]

-----------------------------------------------

                                                 <spontaneous>
[17]     1.8    0.08        0.00               _yylook [17]
                0.00        0.00       3/3          __filbuf [43]

-----------------------------------------------

                                                 <spontaneous>
[18]     1.8    0.08        0.00               _CopyValue [18]

-----------------------------------------------

                                                 <spontaneous>
[19]     1.3    0.06        0.00               _ExecCompare [19]

-----------------------------------------------

                                                 <spontaneous>
[20]     1.1    0.05        0.00               _FreeString [20]
                0.00        0.00      27/4681       _free [12]

-----------------------------------------------

                                                 <spontaneous>
[21]     1.1    0.04        0.01               _GlobalLook [21]
                0.01        0.00    1120/1120       _strcmp [37]

-----------------------------------------------
```

```
                                                        <spontaneous>
[22]    1.0    0.00        0.05                      _PrintString [22]
               0.00        0.05     26/108              _fprintf [9]

-----------------------------------------------

               0.03        0.01     15/15               _malloc [7]
[23]    0.9    0.03        0.01     15               _morecore [23]
               0.01        0.00     15/17               _sbrk [38]

-----------------------------------------------

                                                        <spontaneous>
[24]    0.7    0.01        0.02                      _ExecFile [24]
               0.00        0.01      1/1                _fclose [36]
               0.01        0.01      3/5                _printf [33]
               0.00        0.00      5/5                _setjmp [85]
               0.00        0.00      1/1                _fopen [91]

-----------------------------------------------

                                                        <spontaneous>
[25]    0.7    0.03        0.00                      _ClearStack [25]

-----------------------------------------------

                                                        <spontaneous>
[26]    0.7    0.03        0.00                      _CompareValue [26]

-----------------------------------------------

                                                        <spontaneous>
[27]    0.7    0.03        0.00                      _CopyString [27]
               0.00        0.00     50/50               _strlen [83]
               0.00        0.00     50/50               _strcpy [82]

-----------------------------------------------

                                                        <spontaneous>
[28]    0.7    0.01        0.02                      _PreDefine [28]
               0.02        0.00      1/1                _srandom [30]
               0.00        0.00      1/1                _getpid [93]

-----------------------------------------------

                                                        <spontaneous>
[29]    0.7    0.03        0.00                      _yyparse [29]

-----------------------------------------------

               0.02        0.00      1/1                _PreDefine [28]
[30]    0.4    0.02        0.00      1               _srandom [30]
               0.00        0.00    310/310              _random [81]

-----------------------------------------------

                                                        <spontaneous>
[31]    0.4    0.02        0.00                      _MakeDynamic [31]

-----------------------------------------------

                                                        <spontaneous>
[32]    0.4    0.02        0.00                      _yylex [32]
               0.00        0.00      5/5                _atoi [84]

-----------------------------------------------

               0.00        0.00      2/5                _main [34]
               0.01        0.01      3/5                _ExecFile [24]
[33]    0.4    0.01        0.01      5               _printf [33]
```

```
                0.00        0.01       5/113          __doprnt [8]

-------------------------------------------------

                                                       <spontaneous>
[34]     0.4    0.01        0.01                   _main [34]
                0.00        0.00       2/5            _printf [33]

-------------------------------------------------

                                                       <spontaneous>
[35]     0.3    0.01        0.00                   _FreeParse [35]
                0.00        0.00      63/4681         _free [12]

-------------------------------------------------

                0.00        0.01       1/1            _ExecFile [24]
[36]     0.3    0.00        0.01       1           _fclose [36]
                0.01        0.00       1/1            _close [39]
                0.00        0.00       1/4            _fflush [41]
                0.00        0.00       1/4681         _free [12]

-------------------------------------------------

                0.01        0.00    1120/1120         _GlobalLook [21]
[37]     0.2    0.01        0.00    1120           _strcmp [37]

-------------------------------------------------

                0.00        0.00       2/17           _malloc [7]
                0.01        0.00      15/17           _morecore [23]
[38]     0.2    0.01        0.00      17           _sbrk [38]

-------------------------------------------------

                0.01        0.00       1/1            _fclose [36]
[39]     0.2    0.01        0.00       1           _close [39]

-------------------------------------------------

                                                       <spontaneous>
[40]     0.2    0.01        0.00                   _CheckSign [40]

-------------------------------------------------

                0.00        0.00       1/4            __filbuf [43]
                0.00        0.00       1/4            _fclose [36]
                0.00        0.00       2/4            _fprintf [9]
[41]     0.1    0.00        0.01       4           _fflush [41]
                0.01        0.00       2/38           _write [14]

-------------------------------------------------

                                                       <spontaneous>
[42]     0.1    0.00        0.00                   _PreExit [42]
                0.00        0.00       2/106          _fprintf [9]

-------------------------------------------------

                0.00        0.00       3/3            _yylook [17]
[43]     0.0    0.00        0.00       3           __filbuf [43]
                0.00        0.00       1/4            _fflush [41]
                0.00        0.00       2/4812         _malloc [7]
                0.00        0.00       3/3            _read [89]
                0.00        0.00       2/3            _fstat [88]

-------------------------------------------------

                0.00        0.00     310/310          _srandom [30]
[81]     0.0    0.00        0.00     310           _random [81]
```

BLANK PAGE INSERTED

```
-----------------------------------------------

                0.00       0.00       1/1          __flsbuf [13]
[95]     0.0    0.00       0.00       1        _isatty [95]
                0.00       0.00       1/1          _ioctl [94]

-----------------------------------------------

                0.00       0.00       1/1          _fopen [91]
[96]     0.0    0.00       0.00       1        _open [96]

-----------------------------------------------

                0.00       0.00       1/1          _moncontrol [180]
[97]     0.0    0.00       0.00       1        _profil [97]

-----------------------------------------------
```

```
flat profile:

 %          the percentage of the total running time of the
time        program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds    for by this function and those listed above it.

 self       the number of seconds accounted for by this
seconds     function alone.  This is the major sort for this
            listing.

calls       the number of times this function was invoked, if
            this function is profiled, else blank.

 self       the average number of milliseconds spent in this
ms/call     function per call, if this function is profiled,
            else blank.

 total      the average number of milliseconds spent in this
ms/call     function and its descendents per call, if this
            function is profiled, else blank.

name        the name of the function.  This is the minor sort
            for this listing. The index shows the location of
            the function in the gprof listing. If the index is
            in parenthesis it shows where it would appear in
            the gprof listing if it were to be printed.
```

granularity: each sample hit covers 4 byte(s) for 0.20% of 5.06 seconds

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
24.9	1.26	1.26				_ExecParse [1]
11.5	1.84	0.58				_CheckStack [2]
9.9	2.34	0.50				mcount (68)
7.9	2.74	0.40				_PopStack [5]
7.1	3.10	0.36				_PushStack [6]
5.9	3.40	0.30				_FreeValue [4]
4.9	3.65	0.25	4812	0.05	0.06	_malloc [7]
2.8	3.79	0.14				_GetMemory [3]
2.8	3.93	0.14				_MakeValue [10]
2.6	4.06	0.13	4681	0.03	0.03	_free [12]
2.2	4.17	0.11	38	2.89	2.89	_write [14]
2.0	4.27	0.10				_ExecFunction [15]
1.8	4.36	0.09				_PreSize [16]
1.6	4.44	0.08	113	0.71	1.72	__doprnt [8]
1.6	4.52	0.08				_CopyValue [18]
1.6	4.60	0.08				_yylook [17]
1.2	4.66	0.06				_ExecCompare [19]
1.0	4.71	0.05				_FreeString [20]
0.8	4.75	0.04				_GlobalLook [21]
0.6	4.78	0.03	15	2.00	2.59	_morecore [23]
0.6	4.81	0.03				_ClearStack [25]
0.6	4.84	0.03				_CompareValue [26]
0.6	4.87	0.03				_CopyString [27]
0.6	4.90	0.03				_yyparse [29]
0.4	4.92	0.02	1	20.00	20.00	_srandom [30]
0.4	4.94	0.02				_MakeDynamic [31]
0.4	4.96	0.02				_yylex [32]
0.2	4.97	0.01	1120	0.01	0.01	_strcmp [37]
0.2	4.98	0.01	38	0.26	3.01	__flsbuf [13]
0.2	4.99	0.01	17	0.59	0.59	_sbrk [38]
0.2	5.00	0.01	5	2.00	3.72	_printf [33]
0.2	5.01	0.01	1	10.00	10.00	_close [39]
0.2	5.02	0.01				_CheckSign [40]
0.2	5.03	0.01				_ExecFile [24]
0.2	5.04	0.01				_FreeParse [35]
0.2	5.05	0.01				_PreDefine [28]
0.2	5.06	0.01				_main [34]
0.0	5.06	0.00	310	0.00	0.00	_random [81]
0.0	5.06	0.00	108	0.00	1.75	_fprintf [9]
0.0	5.06	0.00	50	0.00	0.00	_strcpy [82]
0.0	5.06	0.00	50	0.00	0.00	_strlen [83]
0.0	5.06	0.00	5	0.00	0.00	_atoi [84]
0.0	5.06	0.00	5	0.00	0.00	_setjmp [85]
0.0	5.06	0.00	5	0.00	0.00	_sigblock [86]
0.0	5.06	0.00	5	0.00	0.00	_sigstack [87]
0.0	5.06	0.00	4	0.00	1.45	_fflush [41]
0.0	5.06	0.00	3	0.00	0.52	__filbuf [43]
0.0	5.06	0.00	3	0.00	0.00	_fstat [88]
0.0	5.06	0.00	3	0.00	0.00	_read [89]
0.0	5.06	0.00	1	0.00	0.00	__findiop [90]
0.0	5.06	0.00	1	0.00	11.48	_fclose [36]
0.0	5.06	0.00	1	0.00	0.00	_fopen [91]
0.0	5.06	0.00	1	0.00	0.00	_getpagesize [92]
0.0	5.06	0.00	1	0.00	0.00	_getpid [93]
0.0	5.06	0.00	1	0.00	0.00	_ioctl [94]
0.0	5.06	0.00	1	0.00	0.00	_isatty [95]
0.0	5.06	0.00	1	0.00	0.00	_open [96]
0.0	5.06	0.00	1	0.00	0.00	_profil [97]

Index by function name