GPPL: A Small Block-Structured Imperative Programming Language Implemented Using **Ox**

by Kurt M. Bischoff

TR#92-32
December, 1992

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011

Contents

1 Introduction

GPPL is a small, block-structured, imperative programming language, for which a compiler, **gc**, has been built using the **Yacc/Lex**-based attribute-grammar compiler **Ox** (see [Bischoff]). This paper describes **GPPL** and **gc**, and is directed mainly to those who would like to study and modify them. The implementation of **GPPL** may be considered as an example of the use of attribute grammars in general and as a nontrivial example application of **Ox**.

The syntax of **GPPL** bears some resemblance to that of **C** but, being described in only about eighty grammar rules, lacks many **C** constructs. **GPPL**'s semantics are similar to those of **Pascal**, with relatively strict error checking.

gc's target language is a very small subset of **C**. The source code for **gc** occupies about seventy kilobytes and constitutes about a dozen files.

2 Lexical Elements

The following are **GPPL**'s reserved words:

```
int float bool void var
if then else while return exit
input output EOF
TRUE FALSE
,  ;  (  )  {  }  [  ]
!
*  /
+  -
<  >  <=  >=
==  !=
&&
||
=
```

Identifiers are sequences of one or more letters or digits, the first character being a letter. Integer constants are sequences of one or more digits. Each floating point constant is a nonempty digit sequence followed by a period, followed by a nonempty digit sequence. Strings and comments are like **C** strings and comments.

3 Syntax

This section indicates the main features of **GPPL**'s syntax. The Y-file `gppl.Y` may be used as the exact syntactic specification for **GPPL**. Several example **GPPL** programs are included in an appendix, and these too may be studied to gain an appreciation of **GPPL**'s syntax.

3.1 Programs and Blocks

Each **GPPL** *program* is a *block*, which is {, followed by a possibly empty sequence of *block elements* followed by }. A block element is a *variable declaration*, a *function declaration*, or an *executable element*. A variable declaration is a *storage declaration*. The shortest **GPPL** program is {}.

3.2 Storage Declarations

Storage declarations are similar in general form to the *definitions* of **C**, with a type specifier (one of: `int`, `float`, `bool`) preceding a comma-separated, semicolon-terminated list of identifiers, each followed optionally by a list of square-bracketed constant integer expressions. Here are three examples of storage declarations:

```
int i,j,a[5][5*7],k;
float r,s[50],t,v;
bool p[10],q;
```

3.3 Function Declarations

A type specifier (defined above) or `void` begins a *heading* of a *function declaration*. There follows an identifier and a parenthesized list of *parameter declarations*, which are storage declarations optionally preceded by `var`. Each of these two lines is an example heading:

```
void qsort(var int v[10000]; int left,right;)

int mod (int i,m;)
```

A function declaration is such a heading followed by a block (described above). Here are two function declarations, the second of which has another function declaration nested within its block:

```
int fact(int i;)
  {if (i == 0)
      then return 1;
      else return i * fact(i - 1);
  }


void printData(var int a[10000]; int size;)
  {int i;

   i = 0;
   while (i < size)
     {output("%d   ",i); output("%d\n",a[i]);
      i = i + 1;
      if (mod(i,50) == 0) then output("\f");
     }

   int mod (int i,m;) {return i-((i/m)*m);}
  }
```

3.4 Executable Elements

An executable element is either a *statement* or a block. A statement is either a semicolon (*empty statement*), an *expression* followed by a semicolon (*expression statement*), or a *control statement.* The expressions of **GPPL** are rather ordinary. The exact rules for expressions may be found in `gppl.Y`. Each line below has an example expression. Terminating any one of these with a semicolon yields a statement.

```
7
a
7 * a
(a)
!p && !q || (b != c)
fact(3)
fun(1,fun(2,b))
input("%f\n",r[i][j])
i = int(r)
a = b[i] = fact(i)
c = (a = d) * (b + 88.3)
```

3.5 Function Invocations

Several of the example expressions given above contain *function invocations* as subexpressions. An identifier followed by a parenthesized, possibly empty, comma-separated list of expressions is a function invocation. **GPPL** also has several built-in functions: `input`, `output`, and the *cast functions*. `input` precedes a left parenthesis, a **C**-style format string, a comma, a storage reference, and a right parenthesis. There are two forms of `output` invocations, one being `output` followed by a parenthesized string. In its other form, `output` has both a string and an expression, parenthesized and separated by a comma, as arguments. A parenthesized expression preceded by `int`, `float`, or `bool` is a cast function invocation.

3.6 Control Statements

A control statement is either an *if statement*, a *while statement*, a *return statement*, or an *exit statement*. An if statement is `if` followed by an expression, `then`, and an executable element, optionally followed by `else` and another executable element. `while` followed by a parenthesized expression and an executable element is a while statement. An exit statement is `exit` followed by an integer expression and a semicolon. A return statement is `return` followed either by an expression and a semicolon, or by a semicolon alone.

4 Comparison with Other Languages

The reader may have noticed that many favorite constructs and facilities of some popular languages (notably **C**) are not mentioned above. **GPPL** lacks compound data types, unions, enumerated types, pointer variables, `for`, `do-while`, `continue`, `break`, `switch`, `goto` and labels, increment and decrement operators, conditional expressions, bit operations, automatic type conversions, dynamic memory allocation, separate compilation, and a macro facility. The absence of these features keeps the compiler small and easy to understand, and leaves many interesting opportunities for exercises that expand the language.

5 Semantics

The following is an informal and incomplete description of the meanings of **GPPL**'s constructs. The reader should follow experience and intuition, and if these fail, **gc**'s source code, in resolving semantics-related questions not addressed in this section.

5.1 Scope of Identifiers

As is common in block-structured languages, **GPPL** employs static scoping rules for resolution of storage references and function invocations. When an identifier is seen in the context of a storage reference, the list of variables declared in the block and (if the block is the block of a function) the formal parameter list are searched. When seen in the context of a function invocation, the list of functions declared in the current block is searched. If the identifier is not found in a given block, the search proceeds outward to enclosing blocks until there is a match or the search fails in the *program block* (the outermost block). In a correct input, for a given block, the sets of identifiers of variables, formal parameters, and functions declared for the block are each disjoint from one another.

An identifier's uses need not precede its declaration.

5.2 Operators

GPPL's operators are as follows:

```
!
*  /
+  -
<  >  <=  >=
==  !=
&&
||
=
```

Their precedences and associativities are as in **C**. Which of the two subexpressions of a given binary expression is executed first is implementation-dependent.

5.3 Types and Casts

Each expression has one of the types enumerated in section 3.2. There are no automatic type conversions. The cast functions are provided for changing types of expressions. `int` and `float` expressions cast to `bool` are `FALSE` for `0` and `0.0`, `TRUE` otherwise. A `TRUE` expression cast to `int` or `float` becomes either `1` or `1.0`, while `FALSE` becomes `0` or `0.0`. Casts between `int` and `float` behave as in **C**.

The arithmetic and relational operators are overloaded, in that they apply to both `int` and `float` operands. The logical operators apply only to operands of type `bool`. The operands of a binary operation must be of the same type. The type of an arithmetic or logical expression is the same as that of its operand(s). Relational expressions are of type `bool`.

The type specifier of a function must match the type of any expressions whose value it `return`s. If the function contains an empty `return` statement, its type specifier must be `void`.

5.4 Input and Output

The built-in I/O functions behave, insofar as the syntax of each permits, like the **C** functions `printf` and `scanf`, except that (since **GPPL** has no pointer variables) `input`'s storage reference argument is not a pointer.

As in **C**'s standard library, `EOF` is an integer constant, and may appear anywhere an integer constant may appear. Its value is defined only in relation to values returned by `input`.

It is illegal to call `input` with a storage reference of type `bool`.

5.5 Storage Items

In this section, k is some non-negative integer, and E is a sequence of k constant integer expressions such that $E_i > 0$ for $1 \leq i \leq k$.

5.5.1 Storage Declarations

The storage declaration:

$$type_0\ id_0\ \texttt{[}\ E_1\ \texttt{]}\ \cdots\ \texttt{[}\ E_k\ \texttt{]}\ \texttt{;}$$

can appear in the two distinct contexts treated below.

Variable Declarations When the context of the above declaration is that of a variable declaration (see section 3.1), id_0 is the name of a storage location in the current block, with storage for $\prod_{i=1}^{k} E_i$ items of type $type_0$ that may be read and written in references permitted by the scope rules of section 5.1. If $k > 0$, id_0 is called an *array*.

Parameter Declarations If the above storage declaration appears in a formal parameter list (see section 3.3) there are two cases:

1. The declaration is not preceded by `var`. In this case, $k = 0$ and id_0 has storage for one item. Upon calling the function, the result of the expression matched to the formal parameter is written to that location, which is otherwise inaccessible to the caller. id_0's item may be read and written in references permitted by the scope rules. The matched actual parameter is said to be *passed by value*.

2. The declaration is preceded by `var`. In this case no storage is reserved for the parameter. The actual parameter matched to the given formal parameter is a reference to storage in the scope of the caller, and references permitted by the scope rules read or write that storage. The actual parameter is said to be *passed by reference*. If $k > 0$, id_0 is called an *array*.

5.5.2 Storage References

Assume a valid declaration of id_0 as in the preceding section. If e is a sequence of k integer expressions such that

$$0 \leq e_i < E_i \text{ for } 1 \leq i \leq k$$

then id_0 `[` e_1 `]` $\cdots$ `[` e_k `]` is a valid reference to a location reserved for id_0. If the above condition on the indices of the reference to id_0 fails, a run-time error is reported and program execution is terminated.

5.5.3 References to Arrays

Suppose that $0 \leq j < k$, and that an array id_0 is declared as follows:

$type_0\ id_0$ `[` E_1 `]` $\cdots$ `[` E_j `]` `[` E_{j+1} `]` $\cdots$ `[` E_k `]` `;`

Let e be a sequence of j integer expressions such that

$0 \leq e_i < E_i$ for $1 \leq i \leq j$

Then id_0 `[` e_1 `]` $\cdots$ `[` e_j `]` refers to one of id_0's subarrays of $\prod_{i=j+1}^{k} E_i$ items of type $type_0$. This is legal only when id_0 `[` e_1 `]` $\cdots$ `[` e_j `]` appears as an actual parameter in a function call, and is matched to a formal parameter id_1 whose declaration is

`var` $type_0\ id_1$ `[` E'_1 `]` $\cdots$ `[` E'_n `]` `;`

where the following conditions hold:

- $k = j + n$
- $E'_1 \leq E_{j+1}$
- $E'_i = E_{j+i}$ for $2 \leq i \leq n$

5.6 Compile-time Error Conditions

Below are listed the semantic error conditions that elicit messages from **GPPL**. Most of the conditions are followed by example programs that illustrate the errors.

- Nonpositive dimension in array declaration.

```
{int a[0];
 float r,s[-5 * -4 * -1];
}
```

- Declared storage of type `void`.

```
{void a[10];
}
```

- Identifier declared more than once in the same block.

```
{int in;
 void in(var float r;) { input("%f\n",r); }
}
```

- Undeclared storage identifier.
- Undeclared function identifier.
- Binary operator type mismatch.

```
bool greater(int i; float r;)
  {return i > r;
  }
```

- Unary operator type mismatch.

```
{int i; bool p;
 p = !i;
}
```

- Assignment operator type mismatch.

```
{int i; bool p;
 p = i;
}
```

- Extra index in storage reference.

```
{int i[5][5];
 output("%d\n",i[3][4][1]);
}
```

- Missing index in array reference.

```
{int i[5][5];
 output("%d\n",i[3]);
}
```

- Extra actual parameter in function invocation.

- Missing actual parameter in function invocation.
- Type of actual parameter does not match type of formal parameter.
- Dimensions of actual parameter incompatible with dimensions of formal parameter (see section 5.5.3).
- Actual parameter is not a reference, but is matched to a reference formal parameter.

```
{int j;
 j = 5;
 square(j+1);
 void square(var int i;){i = i * i;}
}
```

- Array declared as value formal parameter.

```
 {void qsort(int a[1000];int left,right;){ ... }
 }
```

- Attempted input of boolean.

```
{bool p;
 input("%d\n",p);
}
```

- Non-boolean expression as condition in control statement.

```
{int i;
 i = 100;
 while (i) {output("%d\n",i); i = i - 1;}
}
```

- Call to `return` not in any function block.

```
{return;
}
```

- Missing object of `return` in non-`void` function.

```
{int fun(){return;}
}
```

- Mismatched type of object returned from function.

```
{int fun() {return 1.0;}
}
```

- Object returned by `void` function.

```
{void fun() {return 1.0;}
}
```

- Non-integer object of `exit` statement.

```
{bool condCheck() { ... }
 exit condCheck();
}
```

6 Implementation

6.1 Introduction

The reader should be familiar with the use of **Ox**, or should have access to the **Ox** user's manual [Bischoff].

This description of the implementation makes frequent references to **gc**'s source code and may be taken as a general guide to that code.

6.1.1 General Design Notes

Many aspects of the present implementation are influenced by a single design decision: **gc** generates a single stream of code during a single post-decoration left-to-right postorder traversal of the parse tree.

To improve time and space efficiency, some of the attributes are defined as pointers to lists of shared data items.

6.2 Overview of gc's Source Files

- `Makefile` is used to maintain consistent and up-to-date versions of the various component files of **gc**, and is input to the **Make** utility (see [Feldman]).

- `glob.h` contains declarations of types and macros that need be visible from all of **gc**'s `.c` files, including `y.tab.c` and `lex.yy.c`.
- `gppl.Y` is the Y-file and contains the definition of `main`.
- `gppl.L` is the L-file.
- `scan.c` consists of routines to support lexical analysis and evaluation of synthesized attributes of tokens.
- The routines called during tree decoration are contained in `sem.c`. In general, the semantic errors that interfere with normal attribute evaluation are detected and reported by these routines.
- `code.c` contains all of the target-language-dependent routines of **gc**. For expedience, these perform about half of the checks for semantic errors. Nearly all of these routines (namely those prefixed by `cg`) generate code and are executed during the post-decoration parse-tree traversal mentioned above.
- `util.c` holds utility routines for error handling and file handling.
- `scan.h`, `sem.h`, `code.h`, and `util.h` are header files, each of which holds `extern` declarations of routines declared in the corresponding `.c` file.

6.3 Symbols, Locales, and Environments

Conceptually, there are three kinds of *symbols* created by **gc**, one kind for each of **GPPL**'s ways of declaring identifiers. There are symbols for variables, for formal parameters, and for functions. `glob.h` has the declaration of `struct sym`, which is the type of **C** variable used to represent symbols of all three kinds. Some of the `struct sym` members are used for each kind of symbol, while some members are unused for some kinds of symbols. This facilitates use of a single set of routines for manipulating lists of each kind of symbol. The uses of the various members are described below. Source code that allocates the symbol space is located in `sem.c`. After allocation of that space, `symStock` points to the next available `struct sym` in the space, and is incremented each time one is allocated.

A *locale* is a list of symbols, all of the same kind. There are *variable locales*, *parameter locales*, and *function locales*.

Each function symbol has a parameter locale, which contains the symbols of the function's formal parameters.

There is a one-to-one correspondence between *environments* and blocks. Each environment that corresponds to a function block has a function symbol and a parameter locale (that of the function symbol). Environments of unnamed (non-function) blocks lack function symbols and parameter locales. Associated with each environment are a variable locale and a function locale, which are lists of symbols of variables and functions declared in the corresponding block. For each environment, except the one corresponding to the program block, there is an *outer* environment. Each environment has a *level*. Zero is the level of the program block's environment. Every other environment has a level greater by one than that of its outer environment. Locales and environments are searched during identifier resolution as indicated in section 5.1.

6.4 Construction of Symbols and Locales

The token `ID` has the inherited attribute `sym`. `ID` nodes that are parts of subtrees rooted at `funcDecl` or `stgDecl2` nodes are parts of symbol declarations. In each of those contexts, `sym` is evaluated by constructing a new symbol (cf. section 6.7). Such construction involves allocating a `struct sym` and assigning the members that pertain to the symbol's kind. The **C** function `newFuncSym` is defined in `sem.c` and is used for constructing function symbols. Parameter symbols and variable symbols are referred to generically as *storage symbols*, and are constructed by calling `newStgSym`, also in `sem.c`. All symbols use members of their `struct sym` to store:

- The lexeme of the `ID` token.
- The line number of the symbol's declaration.
- a *type*, corresponding to one of: `int`, `float`, `bool`, `void`.
- The level of the environment in which the declaration appears.

Locales are represented as linked lists of `struct sym` variables. When a symbol appears in a locale, the `next` member of the symbol's `struct sym` is `lambdaSym` if the symbol is the last in the list. Otherwise `next` points to the next symbol in the locale.

6.4.1 Construction of Storage Symbols

Dimension Lists A variable symbol or parameter symbol may be that of an array, so for these kinds of symbols it is necessary to keep an ordered list of the symbol's declared dimensions. `dimList` is a synthesized attribute of `dimSeq` nodes. Childless `dimSeq` nodes have empty `dimLists`. Every other `dimSeq` node has a `dimList` constructed by appending the value of a bracketed constant integer expression onto the head of the `dimList` of the node's `dimSeq` child. Thus a `dimSeq` node whose parent is a `stgDecl2` node has a reverse-order list of the dimensions of the symbol.

`struct sym` **Members Specific to Storage Symbols** In addition to the above-mentioned members common to all symbols, storage symbols have:

- a *storage class*. This distinguishes symbols of variables, value parameters, and reference parameters.
- a *scaled dimension list*, i.e., a `dimList` whose every member has, in addition to its dimension, a scale factor used in calculating addresses at run-time.
- the *size* of the symbol, i.e., the amount of storage to be reserved for it.

6.4.2 Construction of Storage Locales

Variable Locales These are synthesized in subtrees whose roots are labeled with the nonterminal `block`, the result of the synthesis being `block`'s `varLoc` attribute. Examination of the Y-file's rules involving `blockElemList` reveals that it also has the `varLoc` attribute. The synthesis is incomplete for `blockElemList` nodes whose parents are not `block` nodes.

Parameter Locales The `paramDeclList` nonterminal has an attribute `loc` used to synthesize parameter locales. Complete synthesis entails that the parent of the `paramDeclList` node is labeled `funcDecl`. At such a `paramDeclList` node, `loc` is a reverse-order list of the symbols of a formal parameter list. That reversal of the order is important to parameter list matching (cf. section 6.8.1) and is easily implemented using the **C** function `concatLocs` (in `sem.c`).

6.4.3 Construction of Function Symbols

Besides a lexeme, a line number, and a level, each function symbol has:

- An integer value that determines a label for the entry point of the function.
- The amount of stack space used to store the actual parameters during calls of the function.
- A parameter locale. This differs slightly from the parameter locale synthesized at the nearby `paramDeclList` node (section 6.4.2): Before `newFuncSym` "attaches" the locale to the incipient function symbol, the locale is traversed and an address is assigned to each of its symbols.

6.4.4 Construction of Function Locales

This is analogous to the synthesis of variable locales (see section 6.4.2). Fragments of locales are concatenated and carried rootward in a subtree whose "main limb" is a chain of `blockElemList` nodes.

6.5 Allocation of Registers for Expression Evaluation

6.5.1 Need for Storage of Intermediate Results

In addition to declared storage (variables and formal parameters), the **gc**-generated program needs undeclared storage where intermediate results of expressions can be stored. For instance, the expression

```
(b + c) * (d + e)
```

requires that the results of `(b + c)` and `(d + e)` be available at the same time, so that they can be multiplied without changing the contents of any declared storage location.

6.5.2 Indication of a Scheme for Register Allocation

The practice of generating all code in one continuous stream, during a post-decoration left-to-right postorder traversal, is consistent with a certain simple way of allocating registers for expression evaluation. Suppose that each register is identified by a non-negative integer serial number, and consider an expression of the form

$$expr_1 \;\; op_0 \;\; expr_2.$$

In **gc**'s output, the code that evaluates $expr_1$ precedes immediately the code that evaluates $expr_2$. Thus any register needed to evaluate $expr_1$, except the one in which its result is stored, may be reused for the evaluation of $expr_2$. If evaluation of $expr_1$ requires no register numbered less than n, then the evaluation of $expr_2$ can use registers numbered $n+1$ and greater, without interfering with the evaluation of $expr_1$. Further, the results in registers n and $n+1$ can be combined according to op_0, the result being stored in register n.

The result of a unary operation can be stored in the same register as its operand.

The indicated scheme affords efficient register allocation for left-recursive expressions. It is wasteful for expressions that are deeply right-recursive.

6.5.3 Attributes for Register Allocation

The `regNum` Attribute Except for the `typeSpec` nonterminal, the nonterminals derivable from the `expr` nonterminal inherit the attribute `regNum`. A node's `regNum` is the serial number of the virtual register which is to store any intermediate result corresponding to the node.

The Y-file has the rules:

```
actParamList : expr ;
```

and

```
actParamList : actParamList ',' expr ;
```

For each `expr` in the list, `cgPush` generates code to push the result of the corresponding expression onto the activation stack. After that code is executed, `expr`'s register can be reused. It follows that each of the `expr` nodes can have the same `regNum` value. `actParamList`'s `regNum` attribute is essentially a dummy that carries a register number leafward so that it can be copied into each of the `expr` nodes.

Consider rules other than the `actParamList` rules mentioned above. For those other rules having the form:

$$lhsSym \ \texttt{:} \ \cdots \ rnSym_0 \ \cdots \ rnSym_k \ \cdots \ \texttt{;}$$

where, for $0 \leq i \leq k$, the $rnSym_i$ are exactly the nonterminals of the right hand side having the `regNum` attribute:

- if $lhsSym$ does not have the `regNum` attribute, $rnSym_i$'s `regNum` attribute is i.
- if $lhsSym$ has the `regNum` attribute, $rnSym_i$'s `regNum` attribute is i plus the `regNum` attribute of $lhsSym$.

The `maxRegNum` Attribute Registers are allocated by the **gc**-generated program at block-entry time. **gc** determines for each block how many registers are needed for activations of the given block. The synthesized attribute `maxRegNum` of the nonterminal `statement` is the largest serial number of all the registers used in executing the statement. For `block` nodes, `maxRegNum` is the largest serial number of all the registers used in executing statements of the block.

6.6 The `env` Attribute

6.6.1 Construction

In `glob.h` is the declaration of `struct env`, which gives the form of **gc**'s representations of environments. Each node labeled `block` has an inherited attribute `env`, which is a pointer to a `struct env`. An `env` is built from attributes described above and passed as arguments to the **C** function `newEnv` (see `sem.c`). The arguments are as follows:

- `outerEnv`, a pointer to the outer environment of the incipient environment.

- `funcSym`. If the new environment is that of a function block, this points to the block's function symbol. Otherwise this is the null pointer `lambdaSym`. For a function block, the environment's parameter locale is accessed indirectly, as the `paramLoc` member of the `funcSym` member of the `struct env`.

- `varLoc` and `funcLoc`, the lists of variables and functions declared in the corresponding block.

- `maxRegNum`. See section 6.5.3.

By three calls of `symDupCheck`, `newEnv` verifies that the three locales of the environment are disjoint from one another. `assignLevel` is called to traverse each locale and write the environment's level into the `struct sym` of each symbol of the locale. The variable locale is traversed by `assignAddresses`, which determines for each variable a frame-pointer-relative address, and returns the size of the storage required by the block's variables.

6.6.2 Inheritance

As seen above, environments are built up principally from information synthesized in subtrees that correspond to **gc**'s declarative constructs. The information contained in a given environment is carried leafward via the `env` attribute:

- for identifier resolution and code generation, and

- to be passed as the `outerEnv` argument when calling `newEnv` to build environments of the next higher level.

6.7 Resolution of Identifier References

`ID` nodes that are parts of subtrees rooted at `stgRef` or `funcInvoc` nodes appear in the context of identifier references (storage references or function invocations). In those contexts, `ID`'s inherited attribute `sym` is evaluated by searching the appropriate locales of a chain of environments for a symbol having the same lexeme (cf. section 6.4).

6.7.1 Storage References

These are resolved by calling `stgRefLookup`, which is passed a string (the lexeme of the `ID` of the reference) and an `env` (see section 6.6.1), which indicates the environment in which the reference occurs. The mentioned lookup function follows the algorithm indicated in section 5.1 and is easily understood by examining its text in `sem.c`. If the search is successful, `stgRefLookup` returns a pointer to the symbol found. Otherwise the error is reported and the null pointer `lambdaSym` is returned.

6.7.2 Function Invocations

The behavior of `funcInvocLookup` is quite like that of `stgRefLookup`. Instead of searching the variable locales and parameter locales in the chain of environments, `funcInvocLookup` searches the function locales.

6.8 Parameter List Matching

This section considers matching of non-empty parameter lists of non-built-in functions.

6.8.1 Checking List Length

The semantics of **GPPL** require that the numbers of actual and formal parameters be the same. Recall from section 6.4.3 that construction of a function symbol entails synthesis of a reverse-order list of the function's parameters, and installation of that list in the symbol's `struct sym`. The `actParamList` nonterminal has an inherited attribute `formParamList`. For `actParamList` nodes whose parent nodes are labeled `funcDecl`, this is the `paramList` member of the `struct sym` of the `ID` associated with the function invocation. The parents of other `actParamList` nodes are themselves `actParamList` nodes, and in each such case, the child inherits the `formParamList` of the parent, less the list's head. The decapitation is performed by `cdrParamList`, which, if passed a list of length zero or one, reports that there were too many actual parameters. If the `actParamList` node corresponding to the `expr` node of the leftmost actual parameter receives a `formParamList` of more than one symbol, the **C** function `paramMatchCheck` reports a missing actual parameter.

6.8.2 Other Checks

Each actual/formal pair must agree in type (sections 3.2 and 5.3), and this is checked by `paramMatchCheck`.

Reference formal parameters can be matched only to storage references. The `isStgRef` synthesized attribute of the `expr` nonterminal is a predicate that is `TRUE` exactly when the node to which it belongs corresponds to a storage reference. Passing a `FALSE` instance of `isStgRef` to `paramMatchCheck` along with the symbol of a reference parameter results in an error report.

6.9 Index Matching in Storage References

The dimensions of a storage reference must be consistent (sections 5.5.2 and 5.5.3) with those of the storage declaration to which it is matched by the scope rules (section 6.7.1). This is checked using list-matching methods similar to those for parameter lists (section 6.8.1). Recall from section 6.4.1 that the `dimList` of the `ID` node of the reference is a scaled, reverse-order list of the symbol's storage dimensions.

There are two cases:

- The storage reference is matched to a formal parameter. In this case the `formalMatch` attribute of the corresponding `stgRef` node is the symbol of the matched parameter, and **gc** must test the condition of section 5.5.3. The **C** function `adjustDimList` (in `sem.c`) matches the less significant dimensions of the dimension list of its `actualSym` argument one-by-one with the dimensions of the dimension list of its `formalSym` argument. It returns the dimension list of its `actualSym` argument truncated by the dimensions matched.

- The storage reference is not matched to a formal parameter, and the `formalMatch` attribute of the `stgRef` node is `lambdaSym`. In this case the condition of section 5.5.2 is tested. `adjustDimList` is called, but the main body of its code is not executed, and it returns the whole dimension list of its `actualSym` argument.

In each case the list returned by `adjustDimList` becomes the value of the `dimList` attribute of the `indexSeq` node whose parent is the `stgRef` node corresponding to the storage reference. The `dimList` attribute of an `indexSeq` node whose parent is itself an `indexSeq` node is inherited as the `dimList` of the parent, less its least significant dimension.

6.10 Code Generation

Chapter 6, *Implementation of Block-Structured Languages*, of [MacLennan] provides good background for understanding **gc**'s code generation and the behavior of the generated programs.

The reader should refer to `code.c` and some sample generated code while studying this section.

6.10.1 Target Language

For the present implementation, the target language is **C**. This is done for the following reasons:

1. The generated programs can be run on any system having a **C** compiler.
2. The **C** library functions `malloc`, `printf`, `scanf`, and `exit`, and **C**'s floating point operations can be used, code generation for the corresponding **GPPL** constructs being of peripheral interest.

Except as mentioned in point 2 above, the generated **C** code is very low level, and is directly comparable to assembly language. The only uses of **C** control constructs constitute:

- *conditional branch* (implemented using an `if` together with a `goto`).
- *unconditional branch* (a `goto`).
- *return from function call.* **C** labels are not stored in variables, so return addresses are represented on the stack as integers. **gc** writes a single `switch` statement whose `case`s are labeled with those integer representations. The code fragment executed for a given `case` is a `goto` whose target is the label of the code following the code for the function call.

Only two variables (those for the stack pointer and the frame pointer) and only one function (`main`) are declared in the generated program.

6.11 Structure and Behavior of the Generated C Program

6.11.1 The Activation Stack

The stack is allocated by a `malloc` call, then the stack pointer is made ready for the first push operation. The stack "grows" from high memory to low memory. Thus each pop gives the stack pointer a higher address, and each push is a subtraction from the stack pointer. Casts are obviated by declaring as unions the items on the stack. In the following, *top* is used in the sense of *stack-as-an-abstract-data-type*, and without reference to the implementation.

An *activation record* is a segment of the stack corresponding to execution of a block and containing the following items pushed in the following order:

1. The *actual parameters*. These are present only for activations of function blocks, being the results of the expressions in the actual parameter list of the invocation. They are pushed leftmost-first, so the rightmost parameter is nearest the top.

2. The *return address*. This is present only for activations of function blocks. It is an integer and is equivalent to the label in the caller to which to return control upon deactivation.

3. The *static link*. This points to an activation record of the enclosing block, the present activation having eventually resulted from that activation of the enclosing block.

4. The *dynamic link*. This points to the record of the activation which immediately preceded the present activation.

5. The *return-value register* (RVR). This is provided to receive, just prior to deactivation, a value to be returned to the previous activation.

6. The *expression registers*. These are used for storing intermediate results of expressions, as discussed in section 6.5.3.

7. *Variables*. This is storage corresponding to variable declarations (see sections 3.1 and 5.5.1).

6.11.2 Function Calls

Preparatory to a function call, the expressions constituting the actual parameters are executed, and their results are pushed on the stack one at a time. This is followed by pushing the return address. Then the static link is computed and pushed, and there is executed an unconditional branch to the beginning of the function's block.

6.11.3 Block Entry

Function blocks are entered as the targets of `goto` statements in the code generated for the caller. Thus the parameters, the return address, and the static link are present on top of the stack at block-entry time.

Unnamed blocks, which are like compound statements, are entered by virtue of their position in the program, rather than pursuant to a function call. Activation records of unnamed blocks have no parameters or return address. The first instruction in the code generated for an unnamed block pushes the static link (in this case, the same as the dynamic link).

An activation record for either kind of block has a dynamic link, registers, and storage for variables. The block-entry code concerned with these items is the same for either kind of block.

6.11.4 Block Exit

Following the code generated for the statements of a block is the *block-exit code*. Such code restores the stack pointer and frame pointer to the values they contained immediately prior to activation of the block being exited, those values having been stored on the stack previously. The block-exit code:

- for either kind of block is executed when control "falls through" the code of the currently active block, i.e., when all of its statements have been executed.

- for a function block is executed to carry out a `return` from the function. In this case the block-exit code is the target of a `goto` in the code generated for the `return` statement

6.11.5 Function Return

A `return` statement in a function block may be nested in unnamed blocks. Thus at the time of its execution, there may be several activation records on top of that of the given function block. Each of the activation records must be popped, and any returned value must be left in RVR of the function block's now-otherwise-useless activation record, so that it can be accessed by the caller.

Here is the sequence of events effecting a `return`:

1. If the function is non-`void`, the value to be returned is copied into RVR of the topmost activation record.
2. For each unnamed block in which the `return` is nested:
   (a) the top activation record is popped.
   (b) If the function is non-`void`, the value in RVR of the just-popped activation record is copied into RVR of the activation record now on top of the stack.
3. The function block's activation record is popped. This is done by transferring control to the function block's block-exit code.
4. The return address, which is now on top of the stack, is read and popped, and control is returned to the statement following the function call.
5. The space for the function's actual parameters is popped.

6.11.6 Storage References and Address Computations

For each storage reference, there is designated a register to hold the address of the storage to be accessed. The *static distance* between a symbol reference and its declaration is the level of the environment of the reference minus the level of the environment of its declaration.

Addressing the Correct Activation Record The static link of an activation record is the head of a linked list of activation records. The static distance is the number of links of that list that must be traversed for calculation of the correct address for the given storage reference. The result of this traversal is that the register designated to hold the address contains the frame pointer of an activation record corresponding to the block of the symbol's declaration.

Offset within the Activation Record The symbol's frame-pointer-relative address (whose arithmetic sign depends on whether the reference is to a parameter or a variable) is added into the register.

Offset within the Symbol's Storage Area After the register is made to contain (as described above) the address of the beginning of the symbol's storage area, there is a possibly-empty sequence of indexing operations, one for each array index in the storage reference. The following steps are done for each such operation:

1. The index expression is evaluated and checked against the appropriate dimension for out-of-bound condition.

2. The value of the index expression is multiplied by the appropriate scale factor and the result is added into the register designated for the reference.

Conditional Dereferencing The address is dereferenced exactly when it is not matched to a reference formal parameter in a function call or the storage reference of an `input` statement.

6.12 C Functions for Code Generation

The following are roughcast descriptions of some of **gc**'s code generation routines. The function definitions are to be found in `code.c`.

6.12.1 `newLabelNum`

With each call, this returns the next in an increasing sequence of non-negative integers. Used for generic generation of labels.

6.12.2 `newReturnLabelNum`

Similar to `newLabelNum`, but used only for generating labels for returns from functions (cf. `cgTrailer`).

6.12.3 `cgHeader`

Writes whatever code needs to be up front, in particular that for:

- the declaration of the type of the stack items
- declaration of stack pointer and frame pointer
- function header for `main`
- stack allocation
- stack initialization

6.12.4 `cgTrailer`

Writes all of the code that must be at the end of the program:

- a `switch` statement that converts an integer on top of the stack into the target label of a `goto`. The switch is executed after each function execution. (cf. `newReturnLabelNum`).
- index out-of-bounds trap.
- closing curly brace of `main`.

6.12.5 `stgSize`

Calculates the amount of storage needed for a given number of locations of a given type. This is an abstraction stub for future implementations with different target languages.

6.12.6 `cgFuncAddress`

Due to the order of code generation, code for a function declaration may immediately follow code for an executable element. When control leaves the code for such an executable element, control must jump over the code for the function in order to avoid execution of the code when the function hasn't been called. Target labels are needed:

- for the jump-over.
- for calls of the function.

These labels are built by giving different alphabetic prefixes to the same integer, namely the `label` member of the `struct sym` passed to `cgFuncAddress`.

6.12.7 `cgFuncCall`

Generates code that:

- pushes the return address
- computes and pushes the static link (one dereference for each unit of static distance)
- transfers control to the code generated for the function
- provides a label as the target of a `goto` executed after the callee is done
- stores the result of the function call in the register designated for that purpose (unless calling a `void` function)
- pops the actual parameters

6.12.8 `cgFuncTrailer`

Generates:

- the `goto` that causes the return of control to the caller.
- the target label for the function jump-over described in section 6.12.6.

6.12.9 `cgBlockEnter`

Code generated by this is as indicated in section 6.11.3.

6.12.10 `cgBlockExit`

See section 6.11.4.

References

[Bischoff] Kurt M. Bischoff, *User Manual for Ox: An Attribute-Grammar Compiling System based on Yacc, Lex and C*, TR#92-30, Department of Computer Science, Iowa State University, Ames, Iowa. December, 1992.

[Feldman] S. I. Feldman, *Make–A Program for Maintaining Computer Programs*, AT&T Bell Laboratories, Murray Hill, New Jersey. Revised and reprinted as PS1:12 in *UNIX Programmer's Manual*, Usenix Association, 1986.

[MacLennan] Bruce J. MacLennan, *Principles of Programming Languages*, 2nd. Ed., Holt, Rinehart and Winston, 1987.

Appendix A: Sample **GPPL** Programs

This appendix shows some examples of programs written in **GPPL**.

Dec 14 11:24 1992 fib.g Page 1

```
/* GPPL program for Fibonacci function */

{
int i;

i = 0;
while (i<24)
  {output("%d  ",i);
   output("%d\n",fib(i));
   i = i + 1;
  }


int fib(int i;)
  {if (i == 0)
      then return 0;
   if (i == 1)
      then return 1;
   return (fib(i-2) + fib(i-1));
  }
}
```

Dec 13 18:57 1992 fact.g Page 1

```
/* GPPL program for factorial function */

/* stores in an array the factorials of the first ten integers,
     then prints them.
 */

{
int i,f[10];

i = 0;
while (i<10)
  {f[i] = fact(i);
   i = i + 1;
  }


int fact(int i;)
  {if (i == 0)
      then return 1;
      else return i*fact(i-1);
  }


i = 0;
while (i<10)
  {output("%d  ",i);
   output("%d\n",f[i]);
   i = i + 1;
  }


}
```

```
/* GPPL selection sort program */

/* generates some data, prints it, sorts it, prints it */

{
int b[1000];

genData(b,100);
printData(b,100);
selSort (100,b) ;
printData(b,100);



int mod (int i,m;)
  {int dum;
   return i-((i/m)*m);
  }



void genData(var int a[1000];int size;)
  {int i;
   a[0] = 29;
   i = 1;
   while (i < size)
     {a[i] = mod((i*i + a[i-1]),size);
      i = i + 1;
     }
  }



void printData(var int a[1000];int size;)
  {int i;
   i = 0;
   while (i < size)
     {output("%d   ",i);
      output("%d\n",a[i]);
      i = i + 1;
     }
  }
```



```
void selSort(int size; var int a[1000];)
  {int i,j,temp,min;

   i = 0;
   while (i < size)
     {
      min = i;
      j = i + 1;
      while (j < size)
        {
         if (a[j] < a[min]) then min = j;
         j = j + 1;
        }
      temp = a[i];
      a[i] = a[min];
      a[min] = temp;
      i = i + 1;
     }
  }
}
```

```
/* quicksort--based on K&R's example in "The C Programming Language" */

/* reads some integers from standard input, quicksorts them and
     prints them to standard output
 */

{
int b[10000];
int inSize;

qsort (b,0,(inSize = getData(b)) - 1);
printData(b,inSize);


int getData(var int a[10000];)
  {int i,inStatus;

   i = 0;
   inStatus = input("%d\n",a[i]);
   while (inStatus != EOF && inStatus != 0)
     {i = i + 1;
      inStatus = input("%d\n",a[i]);
     }
   return i;
  }


void printData(var int a[10000]; int size;)
  {int i;

   i = 0;
   while (i < size)
     {output("%d\n",a[i]);
      i = i + 1;
     }
  }
```



```
void qsort(var int v[10000]; int left,right;)
  {int i,last;

   if (left >= right)
      then
      return;
   swap(v,left,(left+right)/2);
   last = left;
   i = left + 1;
   while (i <=  right)
     {if (v[i] < v[left])
         then
         {last = last + 1;
          swap (v,last,i);
         }
      i = i + 1;
     }
   swap(v,left,last);
   qsort(v,left,last-1);
   qsort(v,last+1,right);
  }


void swap(var int v[10000]; int i,j;)
  {int temp;

   temp = v[i];
   v[i] = v[j];
   v[j] = temp;
  }


}
```

```
/* GPPL program to find prime numbers */

/* reads a number from standard input, prints all primes less than that number.
 */

{int n;
 bool a[100001];

 input("%d\n",n);
 if (n >= 100001) then
    {output("\n  too many\n");
     exit(-1);
    }
    else if (n < 0) then
    {output("\n  must be positive\n");
     exit(-1);
    }
 calcPrimes(n,a);
 printPrimes(n,a);


 void calcPrimes(int size; var bool a[100001];)
   {int i;

    a[0] = FALSE;
    a[1] = FALSE;
    i = 2; while (i < size) {a[i] = TRUE; i = i + 1;}

    i = 2;
    while (i < size)
      {int j;

       if a[i] then
          {j = 2 * i;
           while (j < size)
             {a[j] = FALSE;
              j = j + i;
             }
          }
       i = i + 1;
      }
   }



 void printPrimes(int size; var bool a[100001];)
   {int i;

    i = 2;
    while (i < size)
      {if a[i] then output("%d\n",i);
       i = i + 1;
      }
   }
}
```

Appendix B: **gc** Source Code

This appendix contains the **Make**, **Ox**, and **C** source code for **gc**.

```
# $Header: Makefile,v 1.3 92/12/14 11:31:46 bischoff Exp $

SHELL= /bin/sh
CFLAGS =
CC=cc

.IGNORE:

gc: y.tab.o lex.yy.o scan.o sem.o code.o util.o
	$(CC) $(CFLAGS) -o gc y.tab.o lex.yy.o scan.o \
	                      sem.o code.o util.o -ly -ll

ox.out.y ox.out.l: gppl.Y gppl.L
	ox -YY -Yn500000 -Yc60000 -Yr30000 gppl.Y gppl.L

strip: gppl.Y gppl.L
	ox -S gppl.Y gppl.L

lex.yy.c : ox.out.l
	lex ox.out.l

y.tab.c y.tab.h: ox.out.y
	yacc -dv ox.out.y
	diff y.tab.h y.tab.h.bak > y.tab.h.diff 2>&1
	if test -s y.tab.h.diff; then cp y.tab.h y.tab.h.bak; fi
	rm y.tab.h.diff

scan.o: glob.h util.h scan.c
	$(CC) $(CFLAGS) -c scan.c

sem.o: glob.h y.tab.h.bak util.h sem.c
	$(CC) $(CFLAGS) -c sem.c

code.o: code.c y.tab.h.bak glob.h util.h
	$(CC) $(CFLAGS) -c code.c

util.o: util.c
	$(CC) $(CFLAGS) -c util.c

y.tab.o: y.tab.c y.tab.h.bak glob.h util.h scan.h sem.h code.h
	$(CC) $(CFLAGS) -c y.tab.c

lex.yy.o: lex.yy.c y.tab.h.bak glob.h util.h scan.h
	$(CC) $(CFLAGS) -c lex.yy.c

clean:
	rm *.o ox.out.? lex.yy.c y.tab.c y.tab.h y.output gc

LC:
	lint lex.yy.c y.tab.c sem.c scan.c code.c util.c
```



```
/* $Header: glob.h,v 1.3 92/12/14 11:31:56 bischoff Exp $ */

#include <stdio.h>

#define DONTCARE 0

#define ceiling(num,inc) (((inc) * ((num)/(inc))) + (((num)%(inc))?(inc):0))

#define max(t1,t2) (((t1)>(t2))?(t1):(t2))

#define spaceSetup(string,type,name,size,minSize,after,stock) \
if ((size) < (minSize)) {fprintf(stderr,\
"allocation request for %s space too small.\n",string);\
exit(-1);}\
if (((name) = (type *) malloc((size) * sizeof(type)))  \
    == \
    ((type *) NULL) \
   )   \
{fprintf(stderr,"malloc error in %s space allocation.\n",string);exit(-1);}\
(stock) = (name) + 1; \
(after) = ((name) + (size));

typedef unsigned short lineNumType;
typedef unsigned short labelType;
typedef unsigned short levelType;
typedef short regNumType;

enum stgClass {VARIABLEC,VALPARAMC,REFPARAMC};
enum symType  {INTT,FLOATT,BOOLT,VOIDT,ERRT};
enum mapType  {ARITHM,LOGM};
enum pred     {FALSE,TRUE};

extern enum pred doCodeGen;

extern long stgSize();
extern void handleSemError();
```

```
union num {int i;
           float f;
           int b;
          };

struct dim {long size;
            long scale;
            struct dim *next;
           };

struct sym {
            /* members used for all kinds of symbols */
            char *id;
            lineNumType line;
            enum symType type;
            levelType level;
            struct sym *next;

            /* members used only for storage symbols */
            enum stgClass class;
            long addr;
            long size;
            struct dim *dimList;

            /* members used only for function symbols */
            long paramsSize;
            labelType label;
            struct sym *paramLoc;
           };

struct env {struct sym *funcSym;
            struct sym *varLoc;
            struct sym *funcLoc;
            levelType level;
            short nRegs;
            labelType exitLabel;
            long varsSize;
            struct env *outerEnv;
           };

extern struct dim *dimListSpace;
#define lambdaDim dimListSpace

extern struct sym *symSpace;
#define lambdaSym symSpace

extern struct env *envSpace;
#define lambdaEnv envSpace
```



```
/* $Header: scan.h,v 1.3 92/12/14 11:32:01 bischoff Exp $ */

extern void scanSetup();
extern char *stringLookup();
extern char *copyLexemeOfSTRING();
extern void eatComment();
```

```
/* $Header: sem.h,v 1.3 92/12/14 11:32:04 bischoff Exp $ */

#define FIRSTREGNUM 0

extern struct sym *newStgSym();
extern struct sym *newFuncSym();
extern struct sym *concatLocs();
extern struct sym *cdrParamList();
extern struct sym *stgRefLookup();
extern struct sym *funcInvocLookup();
extern struct env *newEnv();
extern struct dim *adjustDimList();
extern struct dim *cdrDimList();
extern struct dim *consDimList();
extern enum symType unrOpTypeCheck();
extern enum symType binOpTypeCheck();
extern enum symType assignmentTypeCheck();
extern int whileStmtCheck();
extern void semSetup();
extern void paramMatchCheck();
```



```
/* $Header: code.h,v 1.3 92/12/14 11:32:08 bischoff Exp $ */

#include <stdio.h>
#include "y.tab.h"

extern FILE *tempOut;
#define tempOutID "g.c"

extern labelType newLabelNum();
extern labelType newReturnLabelNum();
extern long stgSize();
extern void cgHeader();
extern void cgTrailer();
extern void cgFuncAddress();
extern void cgFuncCall();
extern void cgFuncTrailer();
extern void cgBlockEnter();
extern void cgBlockExit();
extern void cgBinArithExpr();
extern void cgRelExpr();
extern void cgBinLogExpr();
extern void cgLogNegExpr();
extern void cgUnrArithExpr();
extern void cgEmptyIndexSeq();
extern void cgConstSimpleExpr();
extern void cgIndexSeq();
extern void cgAssign();
extern void cgPush();
extern void cgCondDeref();
extern void cgOutputNoArg();
extern void cgOutputOneArg();
extern void cgCast();
extern void cgInput();
extern void cgIfThen();
extern void cgIfThenElse();
extern void cgThen();
extern void cgElse();
extern void cgWhileStatement();
extern void cgWdecision();
extern void cgWhile();
extern void cgReturnStmt();
extern void cgExitStmt();
extern void cgParamMatchCheck();
```

```
/* $Header: util.h,v 1.3 92/12/14 11:32:11 bischoff Exp $ */

extern void yyerror();
extern void fatal();
extern void internal();
extern void handleOverflow();
extern FILE *safeOpen();
extern void printStringArray();
```



```
%{
/* $Header: gppl.L,v 1.3 92/12/14 11:32:15 bischoff Exp $ */

#include "glob.h"
#include "scan.h"
#include "y.tab.h"

%}

IDENT ([a-zA-Z_][a-zA-Z0-9_]*)

%%
int             return(INT);        @{ @INT.line@     = yylineno; @}
float           return(FLOAT);      @{ @FLOAT.line@   = yylineno; @}
bool            return(BOOL);       @{ @BOOL.line@    = yylineno; @}
void            return(VOID);       @{ @VOID.line@    = yylineno; @}
var             return(VAR);
if              return(IF);         @{ @IF.line@      = yylineno; @}
then            return(THEN);
else            return(ELSE);
while           return(WHILE);      @{ @WHILE.line@   = yylineno; @}
return          return(RETURN);     @{ @RETURN.line@  = yylineno; @}
exit            return(EXIT);       @{ @EXIT.line@    = yylineno; @}
input           return(INPUT);
output          return(OUTPUT);
\"              {char *strDum;
                 strDum = copyLexemeOfSTRING();
                 return (STRING);
                                    @{ @STRING.str@   = strDum;   @}
                }
\/\*            eatComment();
[ \n\t\f]*      ;
[0-9]+\.[0-9]+ return(FCONST);     @{ sscanf(yytext,"%f",
                                              &(@FCONST.val@.f)
                                             );
                                    @}
[0-9]+          return(ICONST);     @{ sscanf(yytext,"%d",
                                              &(@ICONST.val@.i)
                                             );
                                    @}
EOF             return(EOFCONST);
TRUE            return(BCONST);     @{ @BCONST.val@.b = 1;        @}
FALSE           return(BCONST);     @{ @BCONST.val@.b = 0;        @}
\(              return('(');        @{ @'('.line@     = yylineno; @}
\[              return('[');        @{ @'['.line@     = yylineno; @}
[,]             return(',');        @{ @','.line@     = yylineno; @}
[{]             return('{');
[}]             return('}');
[)]             return(')');
[;]             return(';');
[\]]            return(']');
&&              return(AND);        @{ @AND.line@     = yylineno; @}
\|\|            return(OR);         @{ @OR.line@      = yylineno; @}
!               return(NOT);        @{ @NOT.line@     = yylineno; @}
[*]             return('*');        @{ @'*'.line@     = yylineno; @}
[+]             return('+');        @{ @'+'.line@     = yylineno; @}
```

```
[/]             return('/');        @{ @'/'.line@     = yylineno; @}
[\-]            return('-');        @{ @'-'.line@     = yylineno; @}
=               return(GETS);       @{ @GETS.line@    = yylineno; @}
==              return(EQ);         @{ @EQ.line@      = yylineno; @}
!=              return(NEQ);        @{ @NEQ.line@     = yylineno; @}
\<              return(LT);         @{ @LT.line@      = yylineno; @}
\>              return(GT);         @{ @GT.line@      = yylineno; @}
\<=             return(LEQ);        @{ @LEQ.line@     = yylineno; @}
\>=             return(GEQ);        @{ @GEQ.line@     = yylineno; @}
{IDENT}         {
                 return(ID);
                 @{ @ID.str@ = stringLookup(yytext,yyleng,1);
                    @ID.line@ = yylineno;
                 @}
                }
.               {fprintf(stderr,"line %d: illegal character\n",yylineno);
                 handleSemError();
                }
%%
```



```
 /* $Header: gppl.Y,v 1.3 92/12/14 11:32:17 bischoff Exp $ */

/* a compiler for a general purpose programming language */

%{
#include "glob.h"
#include "y.tab.h"
#include "scan.h"
#include "sem.h"
#include "code.h"
#include "util.h"
%}


%start prog

%token ID ICONST FCONST BCONST EOFCONST
%token INT FLOAT BOOL VOID
%token VAR
%token IF THEN ELSE WHILE
%token EQ LT GT GEQ LEQ NEQ
%token AND OR NOT
%token INPUT OUTPUT STRING
%token RETURN EXIT

%right GETS
%left OR
%left AND
%nonassoc EQ NEQ
%nonassoc LT LEQ GT GEQ
%left '+' '-'
%left '*' '/'
%right NOT
```

```
@traversal t

@attributes {struct env *env;
             regNumType maxRegNum;
            }
            execElem statement
            returnStatement exitStatement
            ifStatement whileStatement

@attributes {struct env *env;
             regNumType regNum;
             regNumType maxRegNum;
             struct sym *formParamList;
             struct sym *func;
             lineNumType line;
            }
            actParamList

@attributes {struct env *env;
             struct sym *varLoc,*funcLoc;
             regNumType maxRegNum;
            }
            block blockElemList

@attributes {struct sym *loc;
             enum stgClass class;
            }
            stgDecl

@attributes {struct sym *loc;
             struct sym *funcSym;
            }
            paramDeclList

@attributes {struct sym *loc;
            }
            paramDecl

@attributes {struct env *env;
             struct sym *loc;
            }
            funcDecl

@attributes {enum symType type;
             lineNumType line;
            }
            typeSpec

@attributes {enum symType type;
             struct sym *loc;
             enum stgClass class;
            }
            stgDecl2 stgDecl2List
```



```
@attributes {struct dim *dimList;
            }
            dimSeq

@attributes {struct env *env;
             enum symType type;
             regNumType regNum;
             regNumType maxRegNum;
            }
            funcInvoc assignExpr

@attributes {struct env *env;
             enum symType type;
             regNumType regNum;
             regNumType maxRegNum;
             struct sym *formalMatch;
             enum pred isStgRef;
            }
            expr simpleExpr

@attributes {struct env *env;
             regNumType regNum;
             regNumType maxRegNum;
             struct dim *dimList;
             struct sym *sym;
             lineNumType line;
            }
            indexSeq

@attributes {struct env *env;
             enum symType type;
             regNumType regNum;
             regNumType maxRegNum;
             struct sym *formalMatch;
            }
            stgRef

@attributes {char *str;
             lineNumType line;
             struct sym *sym;
            }
            ID

@attributes {char *str;
            }
            STRING

@attributes {lineNumType line;
            }
            AND OR NOT
            GETS
            GT LT EQ NEQ GEQ LEQ
            '*' '+' '/' '-' '(' ',' '['
            FOR IF WHILE DO
            INT FLOAT BOOL VOID
            RETURN EXIT
```

```
@attributes {labelType label;
             regNumType decisionRegNum;
            }
            then wDecision

@attributes {labelType label;
            }
            else while

@attributes {union num val;
            }
            ICONST FCONST BCONST

@attributes {long val;
            }
            iConstExpr
```



```
@macro constSimpleExpr(SYM,TYPE,)
   @i @simpleExpr.type@ = TYPE;
   @i @simpleExpr.maxRegNum@ = @simpleExpr.regNum@;
   @i @simpleExpr.isStgRef@ = FALSE;
   @t cgConstSimpleExpr(@simpleExpr.regNum@,TYPE,@SYM.val@);
@end


@macro unrOpExprdefs(op,)
  @i @expr.type@ = unrOpTypeCheck(@expr.1.type@,op,@op.line@);
  @i @expr.1.env@ = @expr.env@;
  @i @expr.1.regNum@ = @expr.regNum@;
  @i @expr.maxRegNum@ = @expr.1.maxRegNum@;
  @i @expr.1.formalMatch@ = lambdaSym;
  @i @expr.isStgRef@ = FALSE;
@end


@macro binOpExprdefs(op,)
  @i @expr.type@ = binOpTypeCheck(@expr.1.type@,@expr.2.type@,op,@op.line@);
  @i @expr.1.env@ = @expr.env@;
  @i @expr.2.env@ = @expr.env@;
  @i @expr.1.regNum@ = @expr.regNum@;
  @i @expr.2.regNum@ = @expr.regNum@ + 1;
  @i @expr.maxRegNum@ = max(@expr.1.maxRegNum@,@expr.2.maxRegNum@);
  @i @expr.1.formalMatch@ = lambdaSym;
  @i @expr.2.formalMatch@ = lambdaSym;
  @i @expr.isStgRef@ = FALSE;
@end

@macro binArithExpr(op,)
   @t cgBinArithExpr(@expr.1.regNum@,op,@expr.type@);
@end

@macro relExpr(op,)
  @t cgRelExpr(@expr.1.regNum@,op,@expr.1.type@);
@end

@macro binLogExpr(op,)
  @t cgBinLogExpr(@expr.1.regNum@,op);
@end
```

```
%%
prog
        :       block
                @{ @i @block.env@ = newEnv(lambdaEnv,
                                           lambdaSym,
                                           @block.varLoc@,
                                           @block.funcLoc@,
                                           @block.maxRegNum@
                                          );
                @}
        ;

block
        :       '{'     blockElemList   '}'
                @{ @i @blockElemList.env@ = @block.env@;
                   @i @block.maxRegNum@ = @blockElemList.maxRegNum@;
                   @i @block.varLoc@ = @blockElemList.varLoc@;
                   @i @block.funcLoc@ = @blockElemList.funcLoc@;
                   @t cgBlockExit(@block.env@);
                @}
        ;

blockElemList
        :
                @{ @i @blockElemList.varLoc@ = lambdaSym;
                   @i @blockElemList.funcLoc@ = lambdaSym;
                   @i @blockElemList.maxRegNum@ = FIRSTREGNUM - 1;
                   @t cgBlockEnter(@blockElemList.env@);
                @}
        |       blockElemList   execElem
                @{ @i @blockElemList.1.env@ = @blockElemList.env@;
                   @i @blockElemList.maxRegNum@ =
                         max(@blockElemList.1.maxRegNum@,
                             @execElem.maxRegNum@
                            );
                   @i @blockElemList.varLoc@ = @blockElemList.1.varLoc@;
                   @i @blockElemList.funcLoc@ = @blockElemList.1.funcLoc@;
                   @i @execElem.env@ = @blockElemList.env@;
                @}
        |       blockElemList   stgDecl
                @{ @i @blockElemList.1.env@ = @blockElemList.env@;
                   @i @blockElemList.maxRegNum@ = @blockElemList.1.maxRegNum@;
                   @i @blockElemList.varLoc@ =
                        concatLocs(@blockElemList.1.varLoc@,@stgDecl.loc@);
                   @i @blockElemList.funcLoc@ = @blockElemList.1.funcLoc@;
                   @i @stgDecl.class@ = VARIABLEC;
                @}
```



```
        |       blockElemList   funcDecl
                @{ @i @blockElemList.1.env@ = @blockElemList.env@;
                   @i @blockElemList.maxRegNum@ = @blockElemList.1.maxRegNum@;
                   @i @blockElemList.varLoc@ = @blockElemList.1.varLoc@;
                   @i @blockElemList.funcLoc@ =
                        concatLocs(@blockElemList.1.funcLoc@,@funcDecl.loc@);
                   @i @funcDecl.env@ = @blockElemList.env@;
                @}
        ;

execElem
        :       statement
                @{ @i @statement.env@ = @execElem.env@;
                   @i @execElem.maxRegNum@ = @statement.maxRegNum@;
                @}
        |       block
                @{ @i @block.env@ = newEnv(@execElem.env@,
                                           lambdaSym,
                                           @block.varLoc@,
                                           @block.funcLoc@,
                                           @block.maxRegNum@
                                          );
                   @i @execElem.maxRegNum@ = FIRSTREGNUM - 1;
                @}
        ;

statement
        :       ';'
                @{ @i @statement.maxRegNum@ = FIRSTREGNUM - 1;
                @}
        |       expr    ';'
                @{ @i @expr.env@ = @statement.env@;
                   @i @expr.regNum@ = FIRSTREGNUM;
                   @i @statement.maxRegNum@ = @expr.maxRegNum@;
                   @i @expr.formalMatch@ = lambdaSym;
                @}
        |       ifStatement
                @{ @i @ifStatement.env@ = @statement.env@;
                   @i @statement.maxRegNum@ = @ifStatement.maxRegNum@;
                @}
        |       whileStatement
                @{ @i @whileStatement.env@ = @statement.env@;
                   @i @statement.maxRegNum@ = @whileStatement.maxRegNum@;
                @}
        |       returnStatement
                @{ @i @returnStatement.env@ = @statement.env@;
                   @i @statement.maxRegNum@ = @returnStatement.maxRegNum@;
                @}
        |       exitStatement
                @{ @i @exitStatement.env@ = @statement.env@;
                   @i @statement.maxRegNum@ = @exitStatement.maxRegNum@;
                @}
        ;
```

```
expr
        :       assignExpr
                @{ @i @expr.type@ = @assignExpr.type@;
                   @i @assignExpr.env@ = @expr.env@;
                   @i @assignExpr.regNum@ = @expr.regNum@;
                   @i @expr.maxRegNum@ = @assignExpr.maxRegNum@;
                   @i @expr.isStgRef@ = FALSE;
                @}
        |       simpleExpr
                @{ @i @expr.type@ = @simpleExpr.type@;
                   @i @simpleExpr.env@ = @expr.env@;
                   @i @simpleExpr.regNum@ = @expr.regNum@;
                   @i @expr.maxRegNum@ = @simpleExpr.maxRegNum@;
                   @i @simpleExpr.formalMatch@ = @expr.formalMatch@;
                   @i @expr.isStgRef@ = @simpleExpr.isStgRef@;
                @}
        |       '('     expr    ')'
                @{ @i @expr.type@ = @expr.1.type@;
                   @i @expr.1.env@ = @expr.env@;
                   @i @expr.1.regNum@ = @expr.regNum@;
                   @i @expr.maxRegNum@ = @expr.1.maxRegNum@;
                   @i @expr.1.formalMatch@ = @expr.formalMatch@;
                   @i @expr.isStgRef@ = @expr.1.isStgRef@;
                @}
        |       '-'     expr
                @{ unrOpExprdefs('-',)
                   @t cgUnrArithExpr(@expr.regNum@,'-',@expr.type@);
                @}
        |       '+'     expr
                @{ unrOpExprdefs('+',)
                   @t cgUnrArithExpr(@expr.regNum@,'+',@expr.type@);
                @}
        |       NOT     expr
                @{ unrOpExprdefs(NOT,)
                   @t cgLogNegExpr(@expr.regNum@);
                @}
        |       expr    '*'     expr
                @{ binOpExprdefs('*',)
                   binArithExpr('*',)
                @}
        |       expr    '/'     expr
                @{ binOpExprdefs('/',)
                   binArithExpr('/',)
                @}
        |       expr    '+'     expr
                @{ binOpExprdefs('+',)
                   binArithExpr('+',)
                @}
        |       expr    '-'     expr
                @{ binOpExprdefs('-',)
                   binArithExpr('-',)
                @}
```



```
        |       expr    EQ      expr
                @{ binOpExprdefs(EQ,)
                   relExpr(EQ,)
                @}
        |       expr    NEQ     expr
                @{ binOpExprdefs(NEQ,)
                   relExpr(NEQ,)
                @}
        |       expr    LT      expr
                @{ binOpExprdefs(LT,)
                   relExpr(LT,)
                @}
        |       expr    GT      expr
                @{ binOpExprdefs(GT,)
                   relExpr(GT,)
                @}
        |       expr    LEQ     expr
                @{ binOpExprdefs(LEQ,)
                   relExpr(LEQ,)
                @}
        |       expr    GEQ     expr
                @{ binOpExprdefs(GEQ,)
                   relExpr(GEQ,)
                @}
        |       expr    AND     expr
                @{ binOpExprdefs(AND,)
                   binLogExpr(AND,)
                @}
        |       expr    OR      expr
                @{ binOpExprdefs(OR,)
                   binLogExpr(OR,)
                @}
        ;

assignExpr
        :       stgRef  GETS    expr
                @{ @i @assignExpr.type@ = assignmentTypeCheck(@stgRef.type@,
                                                              @expr.type@,
                                                              @GETS.line@
                                                             );
                   @i @stgRef.env@ = @assignExpr.env@;
                   @i @expr.env@ = @assignExpr.env@;
                   @i @expr.regNum@ = @assignExpr.regNum@ + 1;
                   @i @stgRef.regNum@ = @assignExpr.regNum@;
                   @i @assignExpr.maxRegNum@ =
                         max(@stgRef.maxRegNum@,@expr.maxRegNum@);
                   @i @stgRef.formalMatch@ = lambdaSym;
                   @i @expr.formalMatch@ = lambdaSym;
                   @t cgAssign(@stgRef.regNum@,@assignExpr.type@);
                @}
        ;
```

```
simpleExpr
        :       stgRef
                @{ @i @simpleExpr.type@ = @stgRef.type@;
                   @i @stgRef.env@ = @simpleExpr.env@;
                   @i @stgRef.regNum@ = @simpleExpr.regNum@;
                   @i @simpleExpr.maxRegNum@ = @stgRef.maxRegNum@;
                   @i @stgRef.formalMatch@ = @simpleExpr.formalMatch@;
                   @i @simpleExpr.isStgRef@ = TRUE;
                   @t cgCondDeref(@simpleExpr.formalMatch@,@simpleExpr.regNum@,
                                  @simpleExpr.type@
                                 );
                @}
        |       funcInvoc
                @{ @i @simpleExpr.type@ = @funcInvoc.type@;
                   @i @funcInvoc.env@ = @simpleExpr.env@;
                   @i @funcInvoc.regNum@ = @simpleExpr.regNum@;
                   @i @simpleExpr.maxRegNum@ = @funcInvoc.maxRegNum@;
                   @i @simpleExpr.isStgRef@ = FALSE;
                @}
        |       ICONST
                @{ constSimpleExpr(ICONST,INTT,)   @}
        |       FCONST
                @{ constSimpleExpr(FCONST,FLOATT,) @}
        |       BCONST
                @{ constSimpleExpr(BCONST,BOOLT,)  @}
        |       EOFCONST
                @{ @i @simpleExpr.type@ = INTT;
                   @i @simpleExpr.maxRegNum@ = @simpleExpr.regNum@;
                   @i @simpleExpr.isStgRef@ = FALSE;
                   @t cgEOF(@simpleExpr.regNum@);
                @}
        ;

iConstExpr
        :       ICONST
                @{ @i @iConstExpr.val@ = @ICONST.val@.i;
                @}
        |       '-'     iConstExpr
                @{ @i @iConstExpr.val@ = - @iConstExpr.1.val@ ;
                @}
        |       '+'     iConstExpr
                @{ @i @iConstExpr.val@ =   @iConstExpr.1.val@ ;
                @}
        |       '('     iConstExpr      ')'
                @{ @i @iConstExpr.val@ =   @iConstExpr.1.val@ ;
                @}
        |       iConstExpr      '*'     iConstExpr
                @{ @i @iConstExpr.val@ = @iConstExpr.1.val@ *
                                           @iConstExpr.2.val@ ;
                @}
        |       iConstExpr      '+'     iConstExpr
                @{ @i @iConstExpr.val@ = @iConstExpr.1.val@ +
                                           @iConstExpr.2.val@ ;
                @}
```



```
        |       iConstExpr      '/'     iConstExpr
                @{ @i @iConstExpr.val@ = @iConstExpr.1.val@ /
                                           @iConstExpr.2.val@ ;
                @}
        |       iConstExpr      '-'     iConstExpr
                @{ @i @iConstExpr.val@ = @iConstExpr.1.val@ -
                                           @iConstExpr.2.val@ ;
                @}
        ;

stgRef
        :       ID      indexSeq
                @{ @i @ID.sym@ =
                        stgRefLookup(@stgRef.env@,@ID.str@,@ID.line@);
                   @i @indexSeq.env@ = @stgRef.env@;
                   @i @indexSeq.regNum@ = @stgRef.regNum@;
                   @i @stgRef.type@ = @ID.sym@->type;
                   @i @indexSeq.dimList@ =
                        adjustDimList(@stgRef.formalMatch@,@ID.sym@,@ID.line@);
                   @i @stgRef.maxRegNum@ = @indexSeq.maxRegNum@;
                   @i @indexSeq.sym@ = @ID.sym@;
                   @i @indexSeq.line@ = @ID.line@;
                @}
        ;

indexSeq
        :
                @{ @i @indexSeq.maxRegNum@ = @indexSeq.regNum@;
                   @t cgEmptyIndexSeq(@indexSeq.env@,@indexSeq.sym@,
                         @indexSeq.line@,@indexSeq.regNum@,@indexSeq.dimList@
                                     );
                @}
        |       indexSeq        '['     expr    ']'
                @{ @i @indexSeq.1.env@ = @indexSeq.env@;
                   @i @expr.env@ = @indexSeq.env@;
                   @i @indexSeq.1.regNum@ = @indexSeq.regNum@;
                   @i @expr.regNum@ = @indexSeq.regNum@ + 1;
                   @i @indexSeq.maxRegNum@ = max(@indexSeq.1.maxRegNum@,
                                                 @expr.maxRegNum@
                                                );
                   @i @indexSeq.1.dimList@ = cdrDimList(@indexSeq.dimList@,
                                                        @indexSeq.sym@,
                                                        @indexSeq.line@
                                                       );
                   @i @indexSeq.1.sym@ = @indexSeq.sym@;
                   @i @indexSeq.1.line@ = @indexSeq.line@;
                   @i @expr.formalMatch@ = lambdaSym;
                   @t cgIndexSeq(@indexSeq.sym@,@indexSeq.regNum@,
                                 @indexSeq.dimList@
                                );
                @}
        ;
```

```
funcInvoc
        :       ID      '('     ')'
                @{ @i @funcInvoc.type@ = @ID.sym@->type;
                   @i @ID.sym@ =
                        funcInvocLookup(@funcInvoc.env@,@ID.str@,@ID.line@);
                   @i @funcInvoc.maxRegNum@ = @funcInvoc.regNum@;
                   @t cgFuncCall(@funcInvoc.env@,@ID.sym@,
                                 @funcInvoc.regNum@,TRUE,@ID.line@
                                );
                @}
        |       ID      '('     actParamList    ')'
                @{ @i @actParamList.env@ = @funcInvoc.env@;
                   @i @funcInvoc.type@ = @ID.sym@->type;
                   @i @ID.sym@ =
                        funcInvocLookup(@funcInvoc.env@,@ID.str@,@ID.line@);
                   @i @actParamList.regNum@ = @funcInvoc.regNum@;
                   @i @funcInvoc.maxRegNum@ = @actParamList.maxRegNum@;
                   @i @actParamList.formParamList@ = @ID.sym@->paramLoc;
                   @i @actParamList.func@ = @ID.sym@;
                   @i @actParamList.line@ = @ID.line@;
                   @t cgFuncCall(@funcInvoc.env@,@ID.sym@,
                                 @funcInvoc.regNum@,FALSE,@ID.line@
                                );
                @}
        |       typeSpec        '('     expr    ')'
                @{ @i @expr.env@ = @funcInvoc.env@;
                   @i @funcInvoc.type@ = @typeSpec.type@;
                   @i @expr.regNum@ = @funcInvoc.regNum@;
                   @i @funcInvoc.maxRegNum@ = @expr.maxRegNum@;
                   @i @expr.formalMatch@ = lambdaSym;
                   @t cgCast(@expr.regNum@,@typeSpec.type@,@expr.type@);
                @}
        |       INPUT   '('     STRING  ','     stgRef  ')'
                @{ @i @stgRef.env@ = @funcInvoc.env@;
                   @i @funcInvoc.type@ = INTT;
                   @i @stgRef.regNum@ = @funcInvoc.regNum@;
                   @i @funcInvoc.maxRegNum@ = @stgRef.maxRegNum@;
                   @i @stgRef.formalMatch@ = lambdaSym;
                   @t cgInput(@STRING.str@,@stgRef.regNum@,
                              @stgRef.type@,@'('.line@
                             );
                @}
        |       OUTPUT  '('     STRING  ')'
                @{ @i @funcInvoc.type@ = INTT;
                   @i @funcInvoc.maxRegNum@ = @funcInvoc.regNum@;
                   @t cgOutputNoArg(@STRING.str@,@funcInvoc.regNum@);
                @}
```



```
        |       OUTPUT  '('     STRING  ','     expr    ')'
                @{ @i @expr.env@ = @funcInvoc.env@;
                   @i @funcInvoc.type@ = INTT;
                   @i @expr.regNum@ = @funcInvoc.regNum@;
                   @i @funcInvoc.maxRegNum@ = @expr.maxRegNum@;
                   @i @expr.formalMatch@ = lambdaSym;
                   @t cgOutputOneArg(@STRING.str@,@expr.regNum@,@expr.type@);
                @}
        ;

actParamList
        :       expr
                @{ @i @expr.regNum@ = @actParamList.regNum@;
                   @i @actParamList.maxRegNum@ = @expr.maxRegNum@;
                   @i @expr.env@ = @actParamList.env@;
                   @i @expr.formalMatch@ = @actParamList.formParamList@;
                   @t paramMatchCheck(@actParamList.formParamList@,
                                      @expr.type@,
                                      @expr.isStgRef@,TRUE,
                                      @actParamList.func@,@actParamList.line@
                                     );
                      cgPush(@actParamList.regNum@);
                @}
        |       actParamList    ','     expr
                @{ @i @actParamList.1.env@ = @actParamList.env@;
                   @i @expr.env@ = @actParamList.env@;
                   @i @actParamList.1.regNum@ = @actParamList.regNum@;
                   @i @expr.regNum@ = @actParamList.regNum@;
                   @i @actParamList.maxRegNum@ = max(@actParamList.1.maxRegNum@,
                                                     @expr.maxRegNum@
                                                    );
                   @i @actParamList.1.formParamList@ =
                        cdrParamList(@actParamList.formParamList@,
                                     @actParamList.func@,
                                     @actParamList.line@
                                    );
                   @i @actParamList.1.func@ = @actParamList.func@;
                   @i @actParamList.1.line@ = @actParamList.line@;
                   @i @expr.formalMatch@ = @actParamList.formParamList@;
                   @t paramMatchCheck(@actParamList.formParamList@,
                                      @expr.type@,
                                      @expr.isStgRef@,FALSE,
                                      @actParamList.func@,@actParamList.line@
                                     );
                      cgPush(@actParamList.regNum@);
                @}
        ;
```

```
funcDecl
        :       typeSpec        ID      '('     paramDeclList   ')'     block
                @{ @i @block.env@ = newEnv(@funcDecl.env@,
                                           @ID.sym@,
                                           @block.varLoc@,
                                           @block.funcLoc@,
                                           @block.maxRegNum@
                                          );
                   @i @funcDecl.loc@ = @ID.sym@;
                   @i @ID.sym@ = newFuncSym(@ID.str@,@ID.line@,
                                            @typeSpec.type@,
                                            newLabelNum(),
                                            @paramDeclList.loc@
                                           );
                   @i @paramDeclList.funcSym@ = @ID.sym@;
                   @t cgFuncTrailer(@ID.sym@);
                @}
        ;

paramDeclList
        :
                @{ @i @paramDeclList.loc@ = lambdaSym;
                   @t cgFuncAddress(@paramDeclList.funcSym@);
                @}
        |       paramDeclList   paramDecl
                @{ @i @paramDeclList.loc@ =
                        concatLocs(@paramDecl.loc@,@paramDeclList.1.loc@);
                   @i @paramDeclList.1.funcSym@ = @paramDeclList.funcSym@;
                @}
        ;

paramDecl
        :               stgDecl
                @{ @i @paramDecl.loc@ = @stgDecl.loc@;
                   @i @stgDecl.class@ = VALPARAMC;
                @}
        |       VAR     stgDecl
                @{ @i @paramDecl.loc@ = @stgDecl.loc@;
                   @i @stgDecl.class@ = REFPARAMC;
                @}
        ;

stgDecl
        :               typeSpec        stgDecl2List    ';'
                @{ @i @stgDecl.loc@ = @stgDecl2List.loc@;
                   @i @stgDecl2List.type@ = @typeSpec.type@;
                   @i @stgDecl2List.class@ = @stgDecl.class@;
                @}
        ;
```



```
typeSpec
        :       INT
                @{ @i @typeSpec.type@ = INTT;
                   @i @typeSpec.line@ = @INT.line@;
                @}
        |       FLOAT
                @{ @i @typeSpec.type@ = FLOATT;
                   @i @typeSpec.line@ = @FLOAT.line@;
                @}
        |       BOOL
                @{ @i @typeSpec.type@ = BOOLT;
                   @i @typeSpec.line@ = @BOOL.line@;
                @}
        |       VOID
                @{ @i @typeSpec.type@ = VOIDT;
                   @i @typeSpec.line@ = @VOID.line@;
                @}
        ;

stgDecl2List
        :       stgDecl2
                @{ @i @stgDecl2List.loc@ = @stgDecl2.loc@;
                   @i @stgDecl2.type@ = @stgDecl2List.type@;
                   @i @stgDecl2.class@ = @stgDecl2List.class@;
                @}
        |       stgDecl2List    ','     stgDecl2
                @{ @i @stgDecl2List.loc@ = concatLocs(@stgDecl2.loc@,
                                                      @stgDecl2List.1.loc@
                                                     );
                   @i @stgDecl2List.1.type@ = @stgDecl2List.type@;
                   @i @stgDecl2List.1.class@ = @stgDecl2List.class@;
                   @i @stgDecl2.type@ = @stgDecl2List.type@;
                   @i @stgDecl2.class@ = @stgDecl2List.class@;
                @}
        ;

stgDecl2
        :       ID      dimSeq
                @{ @i @stgDecl2.loc@ = @ID.sym@;
                   @i @ID.sym@ = newStgSym(@ID.str@,@ID.line@,
                                           @stgDecl2.type@,
                                           @stgDecl2.class@,
                                           @dimSeq.dimList@
                                          );
                @}
        ;
```

```
dimSeq  :
                @{ @i @dimSeq.dimList@ = lambdaDim;
                @}
        |       dimSeq  '['     iConstExpr      ']'
                @{ @i @dimSeq.dimList@ =
                         consDimList(@iConstExpr.val@,@dimSeq.1.dimList@,
                                    @'['.line@
                                   );
                @}
        ;

ifStatement
        :       IF      expr    then    execElem
                @{ @i @execElem.env@ = @ifStatement.env@;
                   @i @expr.env@ = @ifStatement.env@;
                   @i @expr.regNum@ = FIRSTREGNUM;
                   @i @ifStatement.maxRegNum@ = max(@expr.maxRegNum@,
                                                    @execElem.maxRegNum@
                                                   );
                   @i @expr.formalMatch@ = lambdaSym;
                   @i @then.label@ = newLabelNum();
                   @i @then.decisionRegNum@ = @expr.regNum@;
                   @t cgIfThen(@expr.type@,@IF.line@,@then.label@);
                @}

        |       IF      expr    then    execElem        else    execElem
                @{ @i @expr.env@ = @ifStatement.env@;
                   @i @execElem.env@ = @ifStatement.env@;
                   @i @execElem.1.env@ = @ifStatement.env@;
                   @i @expr.regNum@ = FIRSTREGNUM;
                   @i @ifStatement.maxRegNum@ =
                         max(@expr.maxRegNum@,
                             max(@execElem.maxRegNum@,
                                 @execElem.1.maxRegNum@
                                )
                            );
                   @i @expr.formalMatch@ = lambdaSym;
                   @i @then.label@ = newLabelNum();
                   @i @then.decisionRegNum@ = @expr.regNum@;
                   @i @else.label@ = @then.label@;
                   @t cgIfThenElse(@expr.type@,@IF.line@,@then.label@);
                @}
        ;

then
        :       THEN
                @{ @t cgThen(@then.decisionRegNum@,@then.label@);
                @}
        ;
```



```
else
        :       ELSE
                @{ @t cgElse(@else.label@);
                @}
        ;

whileStatement
        :       while   '('     expr    ')'     wDecision       execElem
                @{ @i @expr.env@ = @whileStatement.env@;
                   @i @execElem.env@ = @whileStatement.env@;
                   @i @expr.regNum@ = FIRSTREGNUM;
                   @i @whileStatement.maxRegNum@ = max(@expr.maxRegNum@,
                                                       @execElem.maxRegNum@
                                                      );
                   @i @expr.formalMatch@ = lambdaSym;
                   @i @while.label@ = newLabelNum();
                   @i @wDecision.label@ = @while.label@;
                   @i @wDecision.decisionRegNum@ = @expr.regNum@;
                   @t cgWhileStatement(@expr.type@,@'('.line@,@while.label@);
                @}
        ;

while
        :       WHILE
                @{ @t cgWhile(@while.label@);
                @}
        ;

wDecision
        :
                @{ @t cgWdecision(@wDecision.decisionRegNum@,@wDecision.label@);
                @}
        ;

returnStatement
        :       RETURN  expr    ';'
                @{ @i @expr.env@ = @returnStatement.env@;
                   @i @expr.regNum@ = FIRSTREGNUM;
                   @i @returnStatement.maxRegNum@ = @expr.maxRegNum@;
                   @i @expr.formalMatch@ = lambdaSym;
                   @t cgReturnStmt(@returnStatement.env@,@expr.regNum@,
                                   @expr.type@,@RETURN.line@
                                  );
                @}
        |       RETURN  ';'
                @{ @i @returnStatement.maxRegNum@ = FIRSTREGNUM - 1;
                   @t cgReturnStmt(@returnStatement.env@,DONTCARE,
                                   VOIDT,@RETURN.line@
                                  );
                @}
```

```
exitStatement
        :       EXIT    expr    ';'
                @{ @i @expr.env@ = @exitStatement.env@;
                   @i @expr.regNum@ = FIRSTREGNUM;
                   @i @exitStatement.maxRegNum@ = @expr.maxRegNum@;
                   @i @expr.formalMatch@ = lambdaSym;
                   @t cgExitStmt(@expr.regNum@,@expr.type@,@EXIT.line@);
                @}
        ;
```



```
%%
#include <stdio.h>

void yyyCheckForResizes();


main(argc,argv)
  int argc;
  char *argv[];
  {int parseStatus;

   yyyCheckForResizes(argc,argv);
   scanSetup();
   semSetup();
   tempOut = safeOpen(tempOutID,"w");
   cgHeader();
   parseStatus = yyparse();
   cgTrailer();
   fclose(tempOut);
   return(parseStatus || (doCodeGen == FALSE));
  }
```

```
/* $Header: util.c,v 1.3 92/12/14 11:32:23 bischoff Exp $ */

#include "glob.h"

extern int yylineno;

void yyerror(cp)
  char *cp;
  {fprintf(stderr,"Syntax error near line %d: %s\n",yylineno,cp);
   yyyerror();
  }



void fatal(cp)
  char *cp;
  {fprintf(stderr,"Fatal error: %s\n",cp);
   exit(-1);
  }



void internal(s)
  char *s;
  {fprintf(stderr,"gppl internal error: %s\n",s);
  }



void handleOverflow(cp)
  char *cp;
  {fatal(cp);
  }



FILE *safeOpen(name,mode)
  char *name,*mode;
  {FILE *dum;
   if ((dum = fopen(name,mode))
        ==
        ((FILE *)NULL)
       )
       {fprintf(stderr,"fatal error opening %s.\n",name);
        exit(-1);
       }
       else
       return dum;
  }
```



```
void printStringArray(arr,sizeofArr,fp)
  char **arr;
  int sizeofArr;
  FILE *fp;
  {long i;
   for (i=0;i<(sizeofArr/sizeof(char *));i++)
       fprintf(fp,arr[i]);
  }
```

```
/* $Header: scan.c,v 1.3 92/12/14 11:32:27 bischoff Exp $ */

#include <stdio.h>
#include <limits.h>
#include "glob.h"
#include "util.h"

extern int yylineno;
#define DFAACCEPT 0



void eatComment()
  {char c;
   int state = 1;
   while (state != DFAACCEPT)
     {c = yyinput();
      switch (state)
        {case 1: switch(c)
                   {case '*' : state = 2; break;
                    default : ;
                   }
                 break;
         case 2: switch(c)
                   {case '*' : break;
                    case '/' : state = DFAACCEPT;
                               break;
                    default  : state = 1;
                   }
                 break;
         default : ;
        }
     }  /* while */
  }
```



```
#define strBufSize 512

char *copyLexemeOfSTRING()
  {char c;
   char *cp,*afterBuf,*temp;
   char buf[strBufSize];
   int state = 1;

   cp = buf;
   *cp++ = '\"';
   afterBuf = buf + strBufSize;
   while (state != DFAACCEPT)
     {*cp++ = c = yyinput();
      if (cp == afterBuf) fatal("string too long\n");
      switch (state)
        {case 1: switch(c)
                   {case '\\' : state = 2; break;
                    case '\"' : state = DFAACCEPT; break;
                    default : ;
                   }
                 break;
         case 2: state = 1;
                 break;
         default : ;
        }
     }  /* while */
   if ((temp = (char *)malloc(1 + cp - buf))
       ==
       ((char *)NULL)
      )
      fatal("malloc error in string allocation\n");
   *cp = '\0';
   strcpy(temp,buf);
   return temp;
  }



/* The size of the array of pointers to permanent null-terminated copies
 * of lexemes of identifiers.
 */
int maxNstrings = 509;


/* stringTable[i] is either (char *)NULL or a
 * permanent null-terminated copy of the lexeme of an identifier.
 */
static char **stringTable;
```

```
/*  generates and returns a pointer to a permanent null-terminated copy
 *  of the first len characters of the string cp.
 */
static char *allocString(cp,len)
  char *cp;
  int len;
  {char *ncp, *str_save;
   if ((ncp = ((char *) malloc(len+1)))
       ==
       ((char *) NULL)
      )
      fatal("internal malloc error in allocString");
   str_save = ncp;
   while (len--)
         *ncp++ = *cp++;
   *ncp = '\0';
   return(str_save);
  }


/*  compute a hash value for a string */
static long stringHash(cp,len)
  char *cp;
  int len;
  {int hashval;
   switch (hashval = len)
     {default: hashval += cp[len - 2] * 512;
      case 3:  hashval += cp[len - 1] * 64;
      case 2:  hashval += cp[1] * 8;
      case 1:  hashval += cp[0]; break;
      case 0:  fatal("internal error: can't seek the empty string.\n");
     }
   return(hashval);
  }
```



```
/* Search the string table for a string matching the first len characters of
 * cp.  If it is absent and insertion is requested, try to insert it, returning
 * a pointer to a permanent null-terminated copy if successful, null pointer
 * if unsuccessful (due to a full table).  If absent and insertion not
 * requested, return null pointer.  If present, return pointer to the copy.
 */
char *stringLookup(cp,len,insertIfAbsent)
  char *cp;
  int len;
  int insertIfAbsent;
  {int i,try;
   char **dum;
   try = stringHash(cp,len) % maxNstrings;
   for (i=0; i<maxNstrings; i++)
       {dum = stringTable + try;
        try = (try + 1) % maxNstrings;
        if (*dum == (char *)NULL)
           {if (insertIfAbsent) *dum = allocString(cp,len);
            return(*dum);
           }
        if (strlen(*dum) != len) continue; /* don't match proper prefixes */
        if (!strncmp(*dum,cp,len)) return(*dum);
       }
   handleOverflow("out of room in string table");
  }


void scanSetup()
  {int i;

   if ((stringTable = (char **) malloc(maxNstrings * sizeof(char *)))
       !=
       ((char **) NULL)
      )
      for (i=0; i<maxNstrings; i++) stringTable[i] = (char *)NULL;
      else
      fatal("malloc error in string table allocation\n");
  }
```

```
/* $Header: sem.c,v 1.3 92/12/14 11:32:32 bischoff Exp $ */

#include "glob.h"
#include "y.tab.h"
#include "util.h"

static int symSpaceSize = 1000;
       struct sym *symSpace;
static struct sym *symStock,*afterSymSpace;


static int envSpaceSize = 1000;
       struct env *envSpace;
static struct env *envStock,*afterEnvSpace;

static int dimListSpaceSize = 1000;
       struct dim *dimListSpace;
static struct dim *dimListStock,*afterDimListSpace;



void handleSemError()
  {doCodeGen = FALSE;
  }



void semSetup()
  {spaceSetup("symbol",struct sym,symSpace,
              symSpaceSize,3,afterSymSpace,symStock
             )

   spaceSetup("environment",struct env,envSpace,
              envSpaceSize,2,afterEnvSpace,envStock
             )

   spaceSetup("dimension list",struct dim,dimListSpace,
              dimListSpaceSize,2,afterDimListSpace,dimListStock
             )

   lambdaSym->next = lambdaSym;
   lambdaSym->dimList = lambdaDim;
   lambdaSym->paramLoc = lambdaSym;

   lambdaEnv->level = -1;
   lambdaEnv->varLoc = lambdaSym;
   lambdaEnv->funcLoc = lambdaSym;
   lambdaEnv->outerEnv = lambdaEnv;

   lambdaDim->next = lambdaDim;
  }
```



```
struct dim *consDimList(size,dimList,line)
  long size;
  struct dim *dimList;
  lineNumType line;
  {struct dim *dum;
   if (size < 1)
      {fprintf(stderr,
         "nonpositive dimension in array declaration near line %d.\n",line
              );
       handleSemError();
      }
   if ((dum = dimListStock++) == afterDimListSpace)
      handleOverflow("out of space for dimension lists\n");
   dum->size = size;
   dum->next = dimList;
   return dum;
  }



static enum pred nameConflict(sym1,sym2)
  struct sym *sym1,*sym2;
  {if (sym1->id == sym2->id)
      {fprintf(stderr,
               "symbol %s multiply defined: lines %d and %d.\n",
               sym1->id,sym1->line,sym2->line
              );
       handleSemError();
       return TRUE;
      }
      else
      return FALSE;
  }



static void *symDupCheck(loc1,loc2)
  struct sym *loc1,*loc2;
  {struct sym *dum;
   while (loc1 != lambdaSym)
     {dum = loc2;
      while (dum != lambdaSym)
        {if (nameConflict(loc1,dum)) break;
         dum = dum->next;
        }
      loc1 = loc1->next;
     }
  }
```

```
/* returns the concatenation of two sequences of symbols,
 * issuing an error message when there is a symbol E1 in
 * one sequence and a symbol E2 in the other sequence
 * such that E1 and E2 have the same identifier.
 * Tacks rightLoc onto the end of leftLoc and returns leftLoc.
 */
struct sym *concatLocs(leftLoc,rightLoc)
  struct sym *leftLoc,*rightLoc;
  {struct sym **prevLeftP;

   symDupCheck(rightLoc,leftLoc);
   prevLeftP = &leftLoc;
   while (*prevLeftP != lambdaSym) prevLeftP = &((*prevLeftP)->next);
   *prevLeftP = rightLoc;
   return leftLoc;
  }



struct dim *cdrDimList(dimList,sym,line)
  struct dim *dimList;
  struct sym *sym;
  lineNumType line;
  {if (dimList == lambdaDim)
      {if (sym != lambdaSym)
          fprintf(stderr,"extra index in line %d\'s reference to %s.\n",
                  line,sym->id
                 );
       handleSemError();
       return lambdaDim;
      }
      else
      return dimList->next;
  }
```



```
struct dim *adjustDimList(formalSym,actualSym,line)
  struct sym *formalSym,*actualSym;
  lineNumType line;
  {struct dim *formDL,*actDL;
   char *msg;

   if (formalSym == lambdaSym) return actualSym->dimList;
   if (actualSym == lambdaSym) return lambdaDim;
   if (formalSym->type != actualSym->type)
      {msg = "type mismatch.\n"; goto errD;}
   formDL = formalSym->dimList;
   actDL = actualSym->dimList;
   while (formDL != lambdaDim)
     {if ((actDL == lambdaDim) ||
          (formDL->scale != actDL->scale) ||
          (formDL->size > actDL->size)
         )
         {msg = "incompatible dimensions.\n"; goto errD;}
      formDL = formDL->next;
      actDL = actDL->next;
     }
   return actDL;
errD:
   fprintf(stderr,
      "actual parameter %s matched to formal parameter %s\n  at line %d: %s",
           actualSym->id,formalSym->id,line,msg
          );
   handleSemError();
   while (formDL != lambdaDim) {formDL = formDL->next; actDL = actDL->next;}
   return actDL;
  }



struct sym *cdrParamList(loc,func,line)
  struct sym *loc;
  struct sym *func;
  lineNumType line;
  {if ((loc == lambdaSym) || (loc->next == lambdaSym))
      {if (func != lambdaSym)
          fprintf(stderr,"extra actual parameter in call to %s near line %d.\n",
                  func->id,line
                 );
       handleSemError();
      }
  if (loc == lambdaSym)
     return lambdaSym;
     else
     return loc->next;
  }
```

```
void paramMatchCheck(formParamList,exprType,isStgRef,isFirstExpr,func,line)
  struct sym *formParamList;
  enum symType exprType;
  enum pred isStgRef,isFirstExpr;
  struct sym *func;
  lineNumType line;

  {enum pred mismatch = FALSE;
   struct sym *symDum;

   if ((func == lambdaSym) || (formParamList == lambdaSym)) return;
   if (isFirstExpr)
      {symDum = formParamList;
       symDum = symDum->next;
       while (symDum != lambdaSym)
         {fprintf(stderr,"missing actual parameter in line %d's call to %s.\n",
                         line,func->id
                 );
          handleSemError();
          mismatch = TRUE;
          symDum = symDum->next;
         }
      }
   if (mismatch) return;
   if ((formParamList->class == REFPARAMC) && !isStgRef)
      {fprintf(stderr,
       "actual parameter in call to %s near line %d is not a reference,\n\
   but matched to formal reference parameter %s.\n",
       func->id,line,formParamList->id
              );
       handleSemError();
      }
   if (formParamList->type != exprType)
      {fprintf(stderr,
       "parameter type mismatch in call to %s near line %d: \n\
    formal parameter %s.\n",
       func->id,line,formParamList->id
              );
       handleSemError();
      }
  }
```



```
static long scaleCalc(dimList,type)
  struct dim *dimList;
  enum symType type;
  {long prevScale,prevSize,sizeOfEach;

   sizeOfEach = stgSize(type,1L);
   prevScale = 1;
   prevSize = 1;
   while (dimList != lambdaDim)
     {dimList->scale = prevScale * prevSize;
      prevScale = dimList->scale;
      prevSize = dimList->size;
      dimList->scale *= sizeOfEach;
      dimList = dimList->next;
     }
   return (prevScale * prevSize * sizeOfEach);
  }



struct sym *newStgSym(lexeme,line,type,class,dimList)
  char *lexeme;
  lineNumType line;
  enum symType type;
  enum stgClass class;
  struct dim *dimList;

  {struct sym *dum;

   if (type == VOIDT)
      {fprintf(stderr,"line %d: declared storage of type void for %s.\n",
               line,lexeme
              );
       handleSemError();
       type = INTT;
      }

   if ((dum = symStock++) == afterSymSpace)
      handleOverflow("out of space in symbol table\n");
   dum->id = lexeme;
   dum->line = line;
   dum->type = type;
   dum->class = class;
   dum->size = scaleCalc(dimList,dum->type);
   if ((class == VALPARAMC) && (dimList != lambdaDim))
      {fprintf(stderr,"array %s declared as value parameter at line %d.\n",
               lexeme,line
              );
       handleSemError();
      }
   if (class == REFPARAMC) dum->size = stgSize(type,1L);
   dum->dimList = dimList;
   dum->next = lambdaSym;
   return dum;
  }
```

```
static long assignAddresses(loc)
  struct sym *loc;
  {long size = 0;

   while (loc != lambdaSym)
     {loc->addr = size;
      size += loc->size,
      loc = loc->next;
     }
   return size;
  }



static void *assignLevel(loc,level)
  struct sym *loc;
  levelType level;
  {while (loc != lambdaSym)
     {loc->level = level;
      loc = loc->next;
     }
  }



struct sym *newFuncSym(lexeme,line,type,label,paramLoc)
  char *lexeme;
  lineNumType line;
  enum symType type;
  labelType label;
  struct sym *paramLoc;
  {struct sym *dum;

   if ((dum = symStock++) == afterSymSpace)
      handleOverflow("out of space in symbol table\n");
   dum->id = lexeme;
   dum->line = line;
   dum->type = type;
   dum->paramsSize = assignAddresses(paramLoc);
   dum->label = label;
   dum->paramLoc = paramLoc;
   dum->next = lambdaSym;
   return dum;
  }
```



```
struct env *newEnv(outerEnv,funcSym,varLoc,funcLoc,maxRegNum)
  struct env *outerEnv;
  struct sym *funcSym;
  struct sym *varLoc,*funcLoc;
  regNumType maxRegNum;
  {struct env *innerEnv;

   if ((innerEnv = envStock++) == afterEnvSpace)
      handleOverflow("out of space in environment table\n");
   symDupCheck(funcSym->paramLoc,varLoc);
   symDupCheck(varLoc,funcLoc);
   symDupCheck(funcSym->paramLoc,funcLoc);
   innerEnv->funcSym = funcSym;
   innerEnv->level = outerEnv->level + 1;
   assignLevel(funcSym->paramLoc,innerEnv->level);
   assignLevel(varLoc,innerEnv->level);
   assignLevel(funcLoc,innerEnv->level);
   innerEnv->funcSym->paramLoc = funcSym->paramLoc;
   innerEnv->varLoc =   varLoc;
   innerEnv->funcLoc =  funcLoc;
   innerEnv->nRegs = maxRegNum + 1;
   innerEnv->exitLabel = newLabelNum();
   innerEnv->varsSize = assignAddresses(varLoc);
   innerEnv->outerEnv = outerEnv;
   return innerEnv;
  }



struct sym *stgRefLookup(env,str,line)
  struct env *env;
  char *str;
  lineNumType line;
  {struct sym *dumSym;

   while (env != lambdaEnv)
     {dumSym = env->funcSym->paramLoc;
      while ((dumSym != lambdaSym) && (dumSym->id != str))
        dumSym = dumSym->next;
      if (dumSym != lambdaSym) return dumSym;

      dumSym = env->varLoc;
      while ((dumSym != lambdaSym) && (dumSym->id != str))
        dumSym = dumSym->next;
      if (dumSym != lambdaSym) return dumSym;

      env = env->outerEnv;
     }
   fprintf(stderr,"undeclared storage identifier %s at line %d.\n",
                  str,line
          );
   handleSemError();
   return lambdaSym;
  }
```

```
struct sym *funcInvocLookup(env,str,line)
  struct env *env;
  char *str;
  lineNumType line;
  {struct sym *dumSym;

   while (env != lambdaEnv)
     {dumSym = env->funcLoc;
      while ((dumSym != lambdaSym) && (dumSym->id != str))
        dumSym = dumSym->next;
      if (dumSym != lambdaSym) return dumSym;

      env = env->outerEnv;
     }
   fprintf(stderr,"undeclared function identifier %s at line %d.\n",
                  str,line
          );
   handleSemError();
   return lambdaSym;
  }


static enum mapType inTypeOf(op)
  int op;
  {switch (op)
     {
      case '*':
      case '+':
      case '-':
      case '/':
      case LT:
      case GT:
      case EQ:
      case LEQ:
      case GEQ:
      case NEQ: return ARITHM;
      case NOT:
      case AND:
      case OR: return LOGM;
      default: fprintf(stderr,"unknown unary operator\n"); exit(-1);
     }
  }
```



```
static enum mapType outTypeOf(op)
  int op;
  {switch (op)
     {
      case '*':
      case '+':
      case '-':
      case '/': return ARITHM;
      case LT:
      case GT:
      case EQ:
      case LEQ:
      case GEQ:
      case NEQ:
      case AND:
      case OR:
      case NOT: return LOGM;
      default: fprintf(stderr,"unknown binary operator\n"); exit(-1);
     }
  }


enum symType binOpTypeCheck(type1,type2,op,line)
  enum symType type1,type2;
  int op;
  lineNumType line;
  {if ((type1 == ERRT) || (type2 == ERRT)) return ERRT;
   if (
       (type1 != type2)
       ||
       (type1 == VOIDT)
       ||
       ((inTypeOf(op) == ARITHM) != ((type1 == INTT) || (type1 == FLOATT)))
      )
      {fprintf(stderr,"binary operator type mismatch at line %d.\n",line);
       handleSemError();
       return ERRT;
      }
   return (outTypeOf(op) == ARITHM)?type1:BOOLT;
  }
```

```
enum symType unrOpTypeCheck(type,op,line)
  enum symType type;
  int op;
  lineNumType line;
  {if (type == ERRT) return ERRT;
   if (
       (type == VOIDT)
       ||
       ((inTypeOf(op) == ARITHM) != ((type == INTT) || (type == FLOATT)))
      )
      {fprintf(stderr,"unary operator type mismatch at line %d.\n",line);
       handleSemError();
       return ERRT;
      }
   return (outTypeOf(op) == ARITHM)?type:BOOLT;
  }



enum symType assignmentTypeCheck(type1,type2,line)
  enum symType type1,type2;
  lineNumType line;
  {if ((type1 == ERRT) || (type2 == ERRT)) return ERRT;
   if (type1 != type2)
      {fprintf(stderr,
               "type mismatch in assignment at line %d.\n",line
              );
       handleSemError();
       return ERRT;
      }
   return type1;
  }
```



```
/* $Header: code.c,v 1.3 92/12/14 11:32:35 bischoff Exp $ */

#include "glob.h"
#include "y.tab.h"
#include "util.h"

#define ALIGNSIZE   4
#define STACKSIZE1 1000000L
#define STACKSIZE  (ceiling(STACKSIZE1,ALIGNSIZE))

#define nFREGS 3
#define RETREGADDR1   -2    /* relative to the callee's fp */
#define RETREGADDR2   -4    /* relative to sp upon return  */

/* RA, VA, PA are for calculations of frame-pointer-relative addresses */
#define RA(regNum)       (-(nFREGS + (regNum)))
#define VA(variableAddr) (-(nFREGS + dumEnv->nRegs + (variableAddr)))
#define PA(paramAddr)    ((paramAddr) + 1 + 1)


enum pred doCodeGen = TRUE;

FILE *tempOut;



long stgSize(type,howMany)
  enum symType type;
  long howMany;
  {return(howMany);
  }



static labelType labelSeq = 0;
static labelType returnLabelSeq = 0;



labelType newLabelNum()
  {return (labelSeq++);
  }



labelType newReturnLabelNum()
  {return (returnLabelSeq++);
  }
```

```
char *cHeaderCode[] =
{
"\n",
"#include <stdio.h>\n",
"\n",
"union stackItem {\n",
"   int i; \n",
"   float f;\n",
"   int b; \n",
"   int *ip;\n",
"   float *fp;\n",
"   int *bp; \n",
"   union stackItem *up;\n",
"  };\n",
"\n",
"\n",
"main()\n",
"{\n",
" register union stackItem *sp,*fp;\n",
"\n",
" if ((sp = (union stackItem *)malloc(STACKSIZE * sizeof(union stackItem)))\n",
"      ==\n",
"      ((union stackItem *)NULL)\n",
"     )\n",
"     fprintf(stderr,\"malloc error in stack allocation\\n\");\n",
" sp = sp + STACKSIZE;\n",
"\n",
};



void cgHeader()
  {if (doCodeGen == FALSE) return;
   fprintf(tempOut,"#define STACKSIZE %d\n",STACKSIZE);
   printStringArray(cHeaderCode,sizeof(cHeaderCode),tempOut);
  }



void cgTrailer()
  {long i;

   if (doCodeGen == FALSE) return;
   fprintf(tempOut,"\n\texit(0);\n");
   fprintf(tempOut,"rts:\n");
   fprintf(tempOut,"  switch ((*(sp++)).i)  \n    {\n");
   for (i=0; i<returnLabelSeq; i++)
       fprintf(tempOut,"     case %4d: goto r%d;\n",i,i);
   fprintf(tempOut,"    } /* rts switch */ \n");
   fprintf(tempOut,"IOBtrap:\n");
   fprintf(tempOut,"\tfprintf(stderr,\"array index out of bounds.\\n\");\n");
   fprintf(tempOut,"\texit(-1);\n");
   fprintf(tempOut,"} /* main */ \n");
  }
```



```
void cgFuncCall(env,funcSym,regNum,isEmptyActList,line)
  struct env *env;
  struct sym *funcSym;
  regNumType regNum;
  enum pred isEmptyActList;
  lineNumType line;
  {int retLabel;
   int staticDist,i;
   struct sym *dum;

   if ((doCodeGen == FALSE) || (funcSym == lambdaSym)) return;
   if (isEmptyActList)
      {dum = funcSym->paramLoc;
       while (dum != lambdaSym)
         {fprintf(stderr,"missing actual parameter in line %d's call to %s.\n",
                         line,funcSym->id
                 );
          handleSemError();
          dum = dum->next;
         }
      }
   staticDist = env->level - funcSym->level;
   retLabel = newReturnLabelNum();
   fprintf(tempOut,"\t(*--sp).i = %d;\n",retLabel);
   fprintf(tempOut,"\t(*--sp).up = fp;\n");
   for (i=0; i<staticDist; i++)
       fprintf(tempOut,"\t*sp = *(*sp).up;\n");
   fprintf(tempOut,"\tgoto f%d;\n",funcSym->label);
   fprintf(tempOut,"r%d:\n",retLabel);
   if (funcSym->type != VOIDT)
      fprintf(tempOut,"\tfp[%d] = sp[%d];\n",RA(regNum),RETREGADDR2);
   fprintf(tempOut,"\tsp += %d;\n",funcSym->paramsSize);
  }



void cgBlockEnter(env)
  struct env *env;
  {if (doCodeGen == FALSE) return;
   if (env != lambdaEnv)
      {if (env->funcSym == lambdaSym)
          fprintf(tempOut,"\t(*--sp).up = fp;\n");
       fprintf(tempOut,"\t(*--sp).up = fp;\n");
       fprintf(tempOut,"\tfp = sp + 1;\n");
       fprintf(tempOut,"\tsp -= %d;\n",
               1 + env->nRegs + env->varsSize
              );
      }
  }
```

```
void cgBlockExit(env)
  struct env *env;
  {if (doCodeGen == FALSE) return;
   if (env == lambdaEnv) return;
   fprintf(tempOut,"BE%d:\tsp = fp;\n",env->exitLabel);
   fprintf(tempOut,"\tfp = (*(sp++ - 1)).up;\n");
  }



void cgFuncAddress(sym)
  struct sym *sym;
  {if (doCodeGen == FALSE) return;
   if (sym != lambdaSym)
      {fprintf(tempOut,"\tgoto j%d;\n",sym->label);
       fprintf(tempOut,"f%d:\t\t\t\t\t\t/* %s */\n",sym->label,sym->id);
      }
  }



void cgFuncTrailer(sym)
  struct sym *sym;
  {if (doCodeGen == FALSE) return;
   fprintf(tempOut,"\tgoto rts;\n",sym->label);
   fprintf(tempOut,"j%d:\n",sym->label);
  }



void cgBinArithExpr(destReg,op,type)
  int destReg;
  int op;
  enum symType type;
  {char *typeExt;

   if (doCodeGen == FALSE) return;
   switch(type)
     {case INTT:    typeExt = "i";  break;
      case FLOATT:  typeExt = "f";  break;
     }
   fprintf(tempOut,"\tfp[%d].%s %c= fp[%d].%s;\n",
           RA(destReg),
           typeExt,
           op,
           RA(destReg + 1),
           typeExt
          );
  }
```



```
void cgRelExpr(destReg,op,type)
  int destReg;
  int op;
  enum symType type;
  {char *typeExt,*opStr;
   labelType label;

   if (doCodeGen == FALSE) return;
   switch(type)
     {case INTT:    typeExt = "i";  break;
      case FLOATT:  typeExt = "f";  break;
     }
   switch (op)
     {case EQ:  opStr = "=="; break;
      case LEQ: opStr = "<="; break;
      case GEQ: opStr = ">="; break;
      case NEQ: opStr = "!="; break;
      case LT:  opStr = "<"; break;
      case GT:  opStr = ">"; break;
     }
   label = newLabelNum();
   fprintf(tempOut,"\tif (fp[%d].%s %s fp[%d].%s) goto b%d;\n",
           RA(destReg),typeExt,opStr,
           RA(destReg + 1),typeExt,label
          );
   fprintf(tempOut,"\tfp[%d].i = 0;\n",RA(destReg));
   fprintf(tempOut,"\tgoto B%d;\n",label);
   fprintf(tempOut,"b%d:\tfp[%d].i = 1;\n",label,RA(destReg));
   fprintf(tempOut,"B%d:\n",label);
  }
```

```
void cgBinLogExpr(destReg,op)
  int destReg;
  int op;
  {labelType label;
   char unrOp,rslt1,rslt2;

   if (doCodeGen == FALSE) return;
   label = newLabelNum();
   switch(op)
     {case AND: unrOp = '!'; rslt1 = '1'; rslt2 = '0'; break;
      case OR : unrOp = ' '; rslt1 = '0'; rslt2 = '1'; break;
     }
   fprintf(tempOut,"\tif (%c(fp[%d].i)) goto b%d;\n",
           unrOp,RA(destReg),label
          );
   fprintf(tempOut,"\tif (%c(fp[%d].i)) goto b%d;\n",
           unrOp,RA(destReg + 1),label
          );
   fprintf(tempOut,"\tfp[%d].i = %c;\n",RA(destReg),rslt1);
   fprintf(tempOut,"\tgoto B%d;\n",label);
   fprintf(tempOut,"b%d:\tfp[%d].i = %c;\n",label,RA(destReg),rslt2);
   fprintf(tempOut,"B%d:\n",label);
  }



void cgUnrArithExpr(destReg,op,type)
  int destReg;
  int op;
  enum symType type;
  {char *typeExt;

   if (doCodeGen == FALSE) return;
   if (op == '+') return;
   switch(type)
     {case INTT:    typeExt = "i";  break;
      case FLOATT:  typeExt = "f";  break;
     }
   fprintf(tempOut,"\tfp[%d].%s = -fp[%d].%s;\n",
           RA(destReg),typeExt,RA(destReg),typeExt
          );
  }
```



```
void cgLogNegExpr(regNum)
  int regNum;
  {labelType label;

   if (doCodeGen == FALSE) return;
   label = newLabelNum();
   fprintf(tempOut,"\tif ((fp[%d].i) == 0) goto b%d;\n",
           RA(regNum),label
          );
   fprintf(tempOut,"\tfp[%d].i = 0;\n",RA(regNum));
   fprintf(tempOut,"\tgoto B%d;\n",label);
   fprintf(tempOut,"b%d:\tfp[%d].i = 1;\n",label,RA(regNum));
   fprintf(tempOut,"B%d:\n",label);
  }



void cgEmptyIndexSeq(env,sym,line,regNum,dimList)
  struct env *env;
  struct sym *sym;
  lineNumType line;
  int regNum;
  struct dim *dimList;
  {long addr;
   int staticDist,i;
   struct env *dumEnv;

   if (doCodeGen == FALSE) return;
   if (sym == lambdaSym) return;
   staticDist = env->level - sym->level;
   while (dimList != lambdaDim)
      {fprintf(stderr,
               "missing index for line %d\'s reference to array %s.\n",
               line,sym->id
              );
       dimList = dimList->next;
       handleSemError();
      }
   fprintf(tempOut,"\tfp[%d].up = fp;\n",RA(regNum));
   dumEnv = env;
   for (i=0; i<staticDist; i++)
       {dumEnv = dumEnv->outerEnv;
        fprintf(tempOut,"\tfp[%d] = *(fp[%d].up);\n",RA(regNum),RA(regNum));
       }
   addr = (sym->class == VARIABLEC)?(VA(sym->addr)):(PA(sym->addr));
   fprintf(tempOut,"\tfp[%d].up += %d;\n",RA(regNum),addr);
   if (sym->class == REFPARAMC)
      fprintf(tempOut,"\tfp[%d] = *(fp[%d].up);\n",
                      RA(regNum),RA(regNum)
             );
  }
```

```
void cgIndexSeq(sym,regNum,dimList)
  struct sym *sym;
  int regNum;
  struct dim *dimList;
  {if (doCodeGen == FALSE) return;
   if (sym == lambdaSym) return;
   fprintf(tempOut,"\tif (fp[%d].i < 0) goto IOBtrap;\n",
           RA(regNum + 1)
          );
   fprintf(tempOut,"\tif (fp[%d].i > %d) goto IOBtrap;\n",
           RA(regNum + 1),
           dimList->size - 1
          );
   fprintf(tempOut,"\tfp[%d].i *= %d;\n",RA(regNum + 1),dimList->scale);
   fprintf(tempOut,"\tfp[%d].up -= fp[%d].i;\n",
           RA(regNum),RA(regNum + 1)
          );
  }



void cgConstSimpleExpr(regNum,type,value)
  int regNum;
  enum symType type;
  union num value;
  {if (doCodeGen == FALSE) return;
   switch(type)
     {case INTT:
        fprintf(tempOut,"\tfp[%d].i = %d;\n",RA(regNum),value.i);
        break;
      case FLOATT:
        fprintf(tempOut,"\tfp[%d].f = %f;\n",RA(regNum),value.f);
        break;
      case BOOLT:
        fprintf(tempOut,"\tfp[%d].b = %d;\n",RA(regNum),value.b);
        break;
     }
  }



void cgEOF(regNum)
  int regNum;
  {if (doCodeGen == FALSE) return;
   fprintf(tempOut,"\tfp[%d].i = EOF;\n",RA(regNum));
  }
```



```
void cgAssign(regNum,type)
  int regNum;
  enum symType type;
  {char *typeExt;

   if (doCodeGen == FALSE) return;
   switch(type)
     {case INTT:    typeExt = "i";  break;
      case FLOATT:  typeExt = "f";  break;
      case BOOLT:   typeExt = "b";  break;
     }
   fprintf(tempOut,"\t*(fp[%d].%sp) = fp[%d].%s;\n",
           RA(regNum),typeExt,RA(regNum + 1),typeExt
          );
   fprintf(tempOut,"\tfp[%d].%s = fp[%d].%s;\n",
           RA(regNum),typeExt,RA(regNum + 1),typeExt
          );
  }



void cgPush(regNum)
  int regNum;
  {if (doCodeGen == FALSE) return;
   fprintf(tempOut,"\t*--sp = fp[%d];\n",RA(regNum));
  }



void cgCondDeref(formal,regNum,type)
  struct sym *formal;
  int regNum;
  enum symType type;
  {char *typeExt;

   if (doCodeGen == FALSE) return;
   if ((formal != lambdaSym) && (formal->class == REFPARAMC)) return;
   switch(type)
     {case INTT:    typeExt = "i";  break;
      case FLOATT:  typeExt = "f";  break;
      case BOOLT:   typeExt = "b";  break;
     }
   fprintf(tempOut,"\tfp[%d].%s = *(fp[%d].%sp);\n",
           RA(regNum),typeExt,
           RA(regNum),typeExt
          );
  }
```

```
void cgCast(regNum,newType,oldType)
  int regNum;
  enum symType newType,oldType;
  {char *ext1,*ext2,*cCast;

   if (doCodeGen == FALSE) return;
   switch(newType)
     {case INTT:    ext1 = "i"; cCast = "(int)"; break;
      case FLOATT:  ext1 = "f"; cCast = "(float)"; break;
      case BOOLT:   ext1 = "b"; cCast = "(int)"; break;
     }
   switch(oldType)
     {case INTT:    ext2 = "i";  break;
      case FLOATT:  ext2 = "f";  break;
      case BOOLT:   ext2 = "b";  break;
     }
   fprintf(tempOut,"\tfp[%d].%s = %sfp[%d].%s;\n",
           RA(regNum),ext1,cCast,RA(regNum),ext2
          );
  }



void cgOutputNoArg(str,regNum)
  char *str;
  regNumType regNum;
  {if (doCodeGen == FALSE) return;
   fprintf(tempOut,"\tfp[%d].i = printf(%s);\n",RA(regNum),str);
    /*  fprintf(tempOut,"\tfflush(stdout);\n"); */
  }
```



```
void cgOutputOneArg(str,regNum,type)
  char *str;
  int regNum;
  enum symType type;
  {if (doCodeGen == FALSE) return;
   switch(type)
     {case INTT:
        fprintf(tempOut,"\tfp[%d].i = printf(%s,fp[%d].i);\n",
                RA(regNum),str,RA(regNum));
        break;
      case FLOATT:
        fprintf(tempOut,
                "\tfp[%d].i = printf(%s,fp[%d].f);\n",
                RA(regNum),str,RA(regNum)
               );
        break;
      case BOOLT:
        fprintf(tempOut,
          "\tfp[%d].i = printf(%s,(*(fp[%d].b)?\"true\":\"false\"));\n",
                RA(regNum),str,RA(regNum)
               );
        break;
     }
   /*  fprintf(tempOut,"\tfflush(stdout);\n"); */
  }



void cgInput(str,regNum,type,line)
  char *str;
  int regNum;
  enum symType type;
  lineNumType line;
  {if (doCodeGen == FALSE) return;
   fprintf(tempOut,"\tfp[%d].i = ",RA(regNum));
   switch(type)
     {case INTT:
        fprintf(tempOut,"scanf(%s,fp[%d].ip);\n",
                str,RA(regNum));
        break;
      case FLOATT:
        fprintf(tempOut,"scanf(%s,fp[%d].fp);\n",
                str,RA(regNum));
        break;
      case BOOLT:
        fprintf(stderr,"attempted input of boolean near line %d.\n",line);
        handleSemError();
        break;
     }
  }
```

```
static void condCheck(type,line)
  enum symType type;
  lineNumType line;
  {if (type != BOOLT)
      {fprintf(stderr,
   "non-boolean expression as condition in control statement near line %d.\n",
               line
              );
       handleSemError();
      }
  }


void cgIfThen(type,line,label)
  enum symType type;
  lineNumType line;
  labelType label;
  {condCheck(type,line);
   if (doCodeGen == FALSE) return;
   fprintf(tempOut,"e%d:\n",label);
  }


void cgIfThenElse(type,line,label)
  enum symType type;
  lineNumType line;
  labelType label;
  {condCheck(type,line);
   if (doCodeGen == FALSE) return;
   fprintf(tempOut,"i%d:\n",label);
  }


void cgThen(regNum,label)
  int regNum;
  labelType label;
  {if (doCodeGen == FALSE) return;
   fprintf(tempOut,"\tif (!(fp[%d].i)) goto e%d;\n",RA(regNum),label);
  }
```



```
void cgElse(label)
  labelType label;
  {if (doCodeGen == FALSE) return;
        fprintf(tempOut,"kl%d:\tgoto i%d;\n",label,label);
        fprintf(tempOut,"e%d:\n",label);
   /* labels prefixed by kl are kludges to silence the C compiler's
    *   "statement not reached" warnings when the then clause ends
    *   with a return statement.
    */
  }


void cgWdecision(regNum,label)
  int regNum;
  labelType label;
  {if (doCodeGen == FALSE) return;
   fprintf(tempOut,"\tif (!(fp[%d].i)) goto W%d;\n",RA(regNum),label);
  }


void cgWhileStatement(type,line,label)
  enum symType type;
  lineNumType line;
  labelType label;
  {condCheck(type,line);
   if (doCodeGen == FALSE) return;
   fprintf(tempOut,"\tgoto w%d;\n",label);
   fprintf(tempOut,"W%d:\n",label);
  }


void cgWhile(label)
  labelType label;
  {if (doCodeGen == FALSE) return;
   fprintf(tempOut,"w%d:\n",label);
  }
```

```
void cgReturnStmt(env,regNum,type,line)
  struct env *env;
  regNumType regNum;
  enum symType type;
  lineNumType line;

  {if (doCodeGen == FALSE) goto err;
   if (type != VOIDT)
      fprintf(tempOut,"\tfp[%d] = fp[%d];\n",RETREGADDR1,RA(regNum));
   while (env->funcSym == lambdaSym)
     {fprintf(tempOut,"\tsp = fp;\n");
      fprintf(tempOut,"\tfp = (*fp).up;\n");
      if (type != VOIDT)
         fprintf(tempOut,"\tfp[%d] = sp[%d];\n",RETREGADDR1,RETREGADDR1);
      if ((env = env->outerEnv) == lambdaEnv)
         {fprintf(stderr,"line %d: return from no function.\n",line);
          goto err;
         }
     }
   if (env->funcSym->type != VOIDT)
      {if (type == VOIDT)
          {fprintf(stderr,
           "line %d: missing object of return in non-void function %s.\n",
                   line,env->funcSym->id
                  );
           goto err;
          }
       if (env->funcSym->type != type)
          {fprintf(stderr,
           "line %d: mismatched type of object returned from function %s.\n",
                  line,env->funcSym->id
                 );
           goto err;
          }
      }
      else
      {if (type != VOIDT)
          {fprintf(stderr,"line %d: object returned by void function %s.\n",
                   line,env->funcSym->id
                  );
           goto err;
          }
      }
   fprintf(tempOut,"\tgoto BE%d;\n",env->exitLabel);
   return;
err:
   handleSemError();
   return;
  }
```



```
void cgExitStmt(regNum,type,line)
  regNumType regNum;
  enum symType type;
  lineNumType line;
  {if (doCodeGen == FALSE) return;
   if (type != INTT)
      {fprintf(stderr,"line %d: non-integer object of exit statement.\n",
               line
              );
       handleSemError();
       return;
      }
   fprintf(tempOut,"\texit(fp[%d].i);\n",RA(regNum));
  }
```