1 Overview

This assignment is an extension to Project 4, that provides semantic actions to generate quadruples for functions. This include both definitions and invocations. This of course includes processing parameter lists in the function definitions and argument lists in the invocations. It also includes teh respective attributes for the new nonterminals to handle parameter_lists and argument_lists.

2 Grammatical Changes

The original grammar for core, which was used for projects 1 and 2, is included below. For previous projects we have modified the grammar. In particular, we have

1. eliminated the input-statement and the output-statement.
2. reworked the expression productions to use precedences to eliminate shift-reduce errors and to eliminate extra nonterminals, operand and factor.
3. Added some additional operators: unary-minus, exponentiation, pointer dereference, address-of operator, etc.
4. Added the middle break loop.
5. Added int, float, and pointers.
6. :
program → PROGRAM declaration . . .
           BEGINTOK statement . . .
           END ;

declaration → DECLARE ID [ , ID ] . . . ;

statement → assignment-statement
             — if-statement
             — loop-statement
             — input-statement
             — output-statement

assignment-statement → ID := expression
if-statement → IF comparison THEN
               statement . . .
               [ ELSE
               statement . . . ]
               ENDIF ;
loop-statement → WHILE comparison LOOP
               statement . . .
               ENDLOOP ;
input-statement → INPUT ID [ , ID ] . . . ;
output-statement → OUTPUT ID [ , ID ] . . . ;
comparison → '(' operand compOperator operand ')
expression → [ expression '+' ] term
             — [ expression '-' ] term
term → [ term '*' ] operand
operand → INTEGER
         — ID
         — '(' expression ')' compOperator → '<' | '=' | '==' | '!=' | '>'

Figure 1: Mini-language Core
2.1 Modifications for Project 5

For this project the grammatical changes that are necessary include:

- The grammar needs to be extend to include function definitions.
- Expressions need to be extended to handle function invocations.
- Symbol table changes include: multiple symbol table changes, symbol table entries need to reflect that the entry is a function, a list of parameters and types.
- ...

3 Function Definitions

To place the function definitions before any declarations of variables how do we modify the grammar section below.

\[ program \rightarrow \text{PROGRAM} \\
\hspace{1em} \text{declaration} \ldots \\
\hspace{2em} \text{BEGIN TOK} \\
\hspace{3em} \text{statement} \ldots \\
\hspace{4em} \text{END;} \]

If we introduce a new nonterminal for a list of function definitions, and place this new “function definitions” nonterminal before “declaration” then this would ensure that all the function definitions occur before any variable in the main program is declared.

The grammar for a function definition is given below:

\[ \text{FuncDef} \rightarrow \text{Type ID ‘(‘ parmlist ‘)’ BEGINTOK declaration L END ;} \]

When the production is reduced we need to

- Insert the return type information into the symbol table entry for the function (ID.place).
- Attach the parmlist to the symbol table (global symbol table) entry for the function (ID.place). Note the parameters should be inserted into the symbol table for the function.
- The declarations for the local variables of the function need to be inserted into the symbol table for this function.
- Note as L is parsed and we generate code, we should be generating offsets off the base pointer of the form “offset(%ebp) for local variables and “-offset(%ebp) for arguments.
• We need to insert a couple of markers, to control which symbol table we are referencing. Note with the nesting that we are using only one symbol table is active at any time. So we set `struct nlist **CurrentTable` initially to the global table (hashtab I think) and then switch it when we start parsing the `parmlist` of a function definition to the table for this function. The Markers needed are:

1. One to switch symbol tables from the global table to the newly allocated for this function.
2. One near the end to switch from the function symbol table back to the global table.

### 4 Symbol Table Modifications

Now the symbol table code that you were supplied includes:

```c
/* /class/csce531-001/Examples/Symt/symtab.h */
#define ENDSTR 0
#define MAXSTR 100
#include <stdio.h>

struct nlist { /* basic table entry */
  char *name;
  struct nlist *next; /*next entry in chain */
  int val;
};

#define HASHSIZE 53

struct nlist *lookup(char *);
struct nlist *install(char *);

/* /class/csce531-001/Examples/Symt/symtab.c */
static struct nlist *hashtab[HASHSIZE]; /* the table */
```

After this in the lexical analyzer when we recognize an ID we find it into the hashtable with either insert/find. You may have noticed that I changed the HASHSIZE Value from 100 to 53. This is because it is generally a good idea to have the HASHSIZE to be prime (reference See Knuth "Art of Computer Programming" or URL http://planetmath.org/encyclopedia/GoodHashTablePrimes.html).

Then there are a number of necessary changes to the files symtab.c and symtab.h from the directory /class/csce531-001/Examples/Symt. Note you may have moved this code into your core.y routines-section and then some of the changes will need to be modified. The necessary changes are summarized in the following list:

1. We need to add a global hash table pointer `CurrentTable` of type “struct nlist **”. It should be defined in symtab.c and then an “`extern struct nlist **CurrentTable;`” should appear in symtab.h.
2. The “static” on the declaration of hashtab needs to be removed and a similar extern definition should be added to symtab.h.

3. We need to add a new function newTable which uses malloc to allocate space for a hashtable.

4. To switch from the global table (hashtab) we use the code “CurrentTable = newTable();”.

5. To switch back in a semantic action we use “CurrentTable = hashtab;”.

6. Install and find should be modified to take a second argument, the table. So the reference then in the flex file would be “yylval.place = install(yytext, CurrentTable);”.

5 Opcodes

To handle functions we need to add a few opcodes to the list in our intermediate language.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Left</th>
<th>Right</th>
<th>Result</th>
<th>BranchTarget</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>FunctionName</td>
</tr>
<tr>
<td>RET</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>PROLOGUE</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>EPILOGUE</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td></td>
</tr>
</tbody>
</table>

6 Output

6.1 Dumping the “main” Symbol Table

6.2 Dumping the Symbol Table for each Function

6.3 Dumping the Code

7 Summary of Deliverables

1. functions.y - Bison specification file for core-plus
2. core.l - Flex specification file for core
3. Makefile with at least the targets
   (a) mycc - my core compiler
   (b) clean - remove all executables
   (c) test run mycc on your test files
4. test1, · · · testn - test files for your compiler
5. out1, · · · outn - the output from testing your compiler