CSCE 531, Spring 2018, Matthews
Project 4: Functions
Assigned: April 10, Due: April 16, 11:59PM

April 13, 2018

1 Overview

This assignment is an extension to Project 3, 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

For previous projects we have modified the grammar. While this list may not be complete, in particular, we have

1. the declarations were redone to be left-recursive
2. reworked the expression productions to use prededences 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 types.
6. $\vdots$

In addition in this next project we will add

1. Some additional operators: unary-minus, exponentiation, pointer dereference, address-of operator, functions, array references, etc.

2. Add long, and char to base types.
3. Add pointers to base types, both declarations and references.
4. Add functions, including function-definitions and invocations.
5. Add arrays of base types and perhaps pointers to base types, including declarations and references
6. changes to correct typos in the grammar for project 3 that listed just ¡statement¿ in the if-then-else etc. instead of <statement-list>.
7. Semantically you should check for undefined variables, redefined variables etc.
8. ⋮

2.1 More on Modifications for Project 4

For this project the grammatical changes that are necessary include:

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

3 Function Definitions

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

$$
\begin{array}{lcl}
\textit{program} & \rightarrow & \text{PROGRAM} \\
 & & \quad \textit{declaration}\ldots \\
 & & \text{BEGINTOK} \\
 & & \quad \textit{statement}\ldots \\
 & & \text{END ;}
\end{array}
$$

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:

FuncDef → 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.

Also, function prototypes should be handled

FuncDef → Type ID '(' typelist ')' ';

4 Pointer and Array references and declarations

4.1 Pointers

Pointers and the operations '*' and '&' (address of) are going to be allowed. So that expressions need to be extended to allow $E \rightarrow' *' ID$ and $E \rightarrow' \&' ID$. There also needs to modifications for declarations of the form $D \rightarrow Basetype\ '*'\ ID$. In core2 we are going to restrict pointers to only point to one of the base types, namely int, long or char. As an example of something you should be able to handle:

```
long  *p, x, *z;
```

4.2 Arrays

One dimensional arrays of the base types or pointers to the basetypes should be allowed. You need to modify the grammar again for declarations and for expressions. For declarations $D \rightarrow ID'[' INT\_CONSTANT']'$

```
long  *p[10], x, z[4];
```

Also, expressions need to be modified to allow for array references, such as

```
x:= 23 * x[i+1]*x[i]; \\
a[i+3] := z * q + r + a[i]\\
```

4.3 Types and Widths

For ints and pointers assume they are four bytes long. For longs assume 8 and for chars 1.

5 Symbol Table Modifications

Now the symbol table code that you were supplied includes:

```
/* /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/Examples/Table/kr.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. 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/Exampl 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);*”.

6 Opcodes

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

Opcode	Left	Right	Result	BranchTarget	Comment
CALL	null	null	null	FunctionName	
RET	null	null	null	null	
PROLOGUE	null	null	null	null	
EPILOGUE	null	null	null	null	

7 Output

7.1 Dumping the “main” Symbol Table

7.2 Dumping the Symbol Table for each Function

7.3 Dumping the Code

8 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) mycore - my core compiler (non-debugging version.)
   (b) clean - remove all executables and generated files such as lex.yy.c.
   (c) test run mycore on your test files
4. test1, $\cdots$ testn - test files for your compiler
5. out1, $\cdots$ outn - the output from testing your compiler