Project 2 – Parser using the YACC Unix Tool

Due date: May 12, 2008 at midnight

Description: In this second part of the compiler project, you will be asked to develop a parser for the programming language miniC. The project (and the compiler) consists of three steps: lexical analysis, parsing and code generation will be developed throughout the semester. These three parts shall be developed in order and they have different deadlines. The compiler should be written in C using the flex and yacc Unix utility tools.

The project shall be developed individually. You are encouraged to discuss with other students, ask teachers, search for ideas on the Internet, etc. But, do not copy code. The solutions will be tested in the Linux-i386 environment. The evaluations of the solutions are based on these tests. The compiler can be developed in many environments. But, when the solutions are delivered, it must be possible to generate the lexical analyzer and the parser from their source codes, and compile the results, etc., in Linux-i386. See the instruction on how to turn-in your projects.

The language miniC was briefly described in the first part (Lexical Analyzer) of the project. Here some more details are given by means of a context free grammar (CFG).

Parsing. The parser shall be developed using the parser generator tool YACC.

In this second part of the project, there are mainly two things to do: First, based on the CFG (described here), develop a grammar for miniC in YACC. Second, make sure that the parser explicitly creates a parse tree when a program is being parsed.

The grammar for miniC described in this document is ambiguous. Moreover it is not complete: productions for the non-terminals expression and unary_expression are missing. It is your task to develop a non-ambiguous grammar in YACC, i.e. there must not be any warnings about conflicts when you generate the parser using YACC. Conflicts can be fixed by adding new non-terminals or by assigning precedence. But, it is important that you understand why there is a conflict before you try to fix it. By running YACC with the –v (verbose) option, YACC also generates a file y.output. This file contains a description of the state machine that YACC generates, and it also describes the conflicts.

To construct a parse tree, you shall use the semantical actions associated to the productions of the grammar. Consider the following example: Suppose there is a function called node that creates a new node.

```
E -> E + T      {$$ = node($2, $1, $3);}
E -> T          {SS = $1}
```

The identifiers $$, $1, $2, … are *synthesized attributes* and their values are computed bottom up. The identifier $$ refers to the left hand side of the production and $1, $2, … refer to the right hand side symbols ($2 refers to the character '+'). So when the first rule above is reduced, a new node in the parse tree is created. The second rule above has only symbol on the right hand side, and in many cases we do not need to create nodes for those productions.

The parse tree will be used in the third part of the project to emit three-address code.

Tokens

The tokens (IDENTIFIER, CONSTANT, SIZEOF, PTR_OP, LT_OP, GT_OP, LE_OP, GE_OP, ARRAY_OP, NE_OP, EQ_OP, AND_OP, OR_OP, TYPEDEF, INT, CHAR, VOID, STRUCT, IF, ELSE, WHILE and RETURN) that the parser expects to receive from the lexer is defined in the file yacc2.y that the parser is generated from. By running YACC with the option –d, YACC generates a header file y.tab.h with definitions of the tokens. The description of the lexer lex2.l can include this file.

Output of your project

The parser will repeatedly call the lexer, which returns the next token. So now you shall comment out the main function in the source code of the lexer lex2.l. Let the file yacc2.y contain the source code to the parser. In yacc2.y add the code

```
main(){
  do{yyparse();}
  while(!feof(yyin));
}
```

This will make the parser call the lexer for the next token until there are no more tokens. Now make sure that, if the input to the parser is grammatically correct then there is no output from the parser, and if it the input is not grammatically correct then the parser says on which line the first error is, and then terminate. In case there are errors, and the first error is on line number 11, then the output from your parser should be

```
syntax error, line 11
```

to stderr.

If there were no errors then the parser shall output some information about the parse tree. It shall how declarations there are on top level, and for each function (procedure) how many declarations, statements and binary operations (*, /, +, -, <, >, ==, >=, <=, !=, &&, ||) there are inside the function. For instance parsing the program

```
  int main(){
    struct node *l;
    int i;
    int sum;
    i=0;
    while(i<10){
      l=insert(i,l);
      i=i + 1;
    }
    sum=listsum_rec(l);
  print_int(sum);
}
```

Should result in the output

```
  Top level: 0 declarations.
  Main: 3 declarations, 6 statements, 2 binary operations.
```

How to Generate a Parser Executable:
Generate the lexer and parser using flex and YACC, respectively:

```
  flex lex2.l
  yacc -d -v yacc2.y
```

The results consist of the files lex.yy.c, y.tab.c and y.tab.h. Compile and link:

```
  gcc -c lex.yy.c
  gcc –c y.tab.c
  gcc lex.yy.o y.tab.o -ll
```

Some test cases with the correct results can be found at the class website. Test and make sure that everything works.

Turn-in Instructions: In your Unix account on the tlinux machine, in the folder xfer, place a file named proj2.tar.zip created using the zip and tar utilities.

Make sure you can unzipp and untar the file and that a makefile you have put inside your proj2.tar.zip file works properly, i.e., we will make your a.out executable using the "make" command and invoke the many tests using the command line:

```
  ./a.out < test.in
```

for all test1.in thought test20.in files.

Grammar for miniC:

```
program -> function_definition
         | declaration
         | program program

declaration -> type_specifier ';'
             | type_specifier declarator ';'

function_definition -> type_specifier declarator compound_statement

type_specifier -> VOID | INT | CHAR
                | STRUCT IDENTIFIER '{' struct_decl_list '}'
                | STRUCT IDENTIFIER
                 | TYPEDEF STRUCT '{' struct_decl_list '}' IDENTIFER
                 | TYPEDEF STRUCT IDENTIFIER '{' struct_decl_list '}' IDENTIFER

struct_decl_list -> type_specifier declarator ';'
                       | struct_decl_list type_specifier declarator ';'

declarator -> '*' direct_declarator
            | direct_declarator

direct_declarator -> IDENTIFIER
                   | direct_declarator '(' parameter_list ')'
                   | direct_declarator '(' ')'

parameter_list -> type_specifier declarator
                | parameter_list ',' parameter_list

statement -> IF '(' expression ')' statement
           | IF '(' expression ')' statement ELSE statement
           | compound_statement
           | ';'
           | unary_expression '=' expression ';'
            | WHILE '(' expression ')' statement
            | RETURN expression ';'

compound_statement -> '{' '}'
                    | '{' declaration_list '}'
                    | '{' statement_list '}'
                    | '{' declaration_list statement_list '}'

declaration_list -> declaration
                  | declaration_list declaration_list

statement_list -> statement
                | statement_list statement_list

expression -> …

unary_expression -> …
```