CSE 310
ASSIGNMENT 3

Construction of a Parser and an Intermediate Code Generator For a subset of Pascal

November 29, 2011

1 Introduction

In our last assignment, we constructed a lexical analyzer which is able to generate token stream for parser. Now is the time to build our own compiler. First, we will build a parser with the help of Lex (Flex) and Yacc (Bison) for the same subset of the language "Pascal" used in the previous assignments. Our task will be to write a yacc source file and to modify our previous lex source file which will be used by Lex and Yacc to generate our intended parser. Next, we will build an intermediate code generator. Since we are not going to design a back-end, we choose Assembly Language as our intended intermediate representation because we can use assemblers to assemble and run the generated assembly codes. So this will effectively be the last step towards our compiler.

2 Input

The input to your program will be a text file containing a program written in a subset language of Pascal we are dealing with. See the sample input file to get an idea.

The names of the input and output files will be input by the user from command line. Use command line arguments for that.

3 Output

Your program should produce an output file – containing all the grammar rules matched by the parser in the exact order they are matched by the parser. For every identifier and number inserted in the symbol table, print the corresponding lexeme in the output file. Also whenever your parser encounters an assignment statement, print the symbol table to indicate that, the correct value is stored in the symbol table.

Your program should produce another output file – containing the semantically equivalent 8086 assembly program generated by your intermediate code generator.

See the attached sample output files for a clear idea about how the output file should look like.

4 Implementation Goals

Step 1: Building a Parser

You have to accomplish the following tasks:

- Modify your code for Symbol Table so that three information of a symbol can be stored in it. That is, now your tokens will look like - `<TYPE, NAME, VALUE> e.g., <NUM, 5, 5.000000>, <ID, myId, 4.000000> etc.`

- Write the following grammar rules in your yacc source file (i.e., .y file) in appropriate syntax with corresponding actions.

```
                program → PROGRAM ID '(' identifier_list ')' SEMICOLON
                          declarations subprogram_declarations
                          compound_statement

          identifier_list → ID | identifier_list COMMA ID

              declarations → declarations VAR identifier_list COLON type
                          SEMICOLON | ε

                   type → standard_type
                          | ARRAY '[' NUM DOTDOT NUM ']' OF standard_type

            standard_type → INTEGER | REAL

  subprogram_declarations → subprogram_declarations subprogram_declaration
                          SEMICOLON | ε

  subprogram_declaration → subprogram_head declarations compound_statement

         subprogram_head → FUNCTION ID arguments COLON standard_type
                          SEMICOLON
                          | PROCEDURE ID arguments SEMICOLON

               arguments → '(' parameter_list ')'| ε

          parameter_list → identifier_list COLON type
                          | parameter_list SEMICOLON identifier_list COLON
                          type

       compound_statement → BEGIN optional_statements END

      optional_statements → statement_list | ε

           statement_list → statement
                          | statement_list SEMICOLON statement

                statement → variable ASSIGNOP expression
                          | procedure_statement
                          | compound_statement
                          | IF expression THEN statement
                          | IF expression THEN statement ELSE statement
```

```
                        | WHILE expression DO statement

                        | write '(' ID ')'

          variable  →   ID | ID '[' expression ']'

procedure_statement  →   ID | ID '(' expression_list ')'

    expression_list  →   expression | expression_list COMMA expression

         expression →    simple_expression
                         | simple_expression RELOP simple_expression

   simple_expression →   term | sign term | simple_expression ADDOP term

              term →     factor | term MULOP factor

            factor  →    ID
                         | ID '(' expression_list ')'
                         | NUM
                         |'(' expression ')'
                         | NOT factor

              sign →     '+' | '-'
```

- Use Symbol Table for all communications between your lexical analyzer and parser. Hints:
  - Redefine the type of `yylval`
    ```
    # ifndef YYSTYPE
    # define YYSTYPE SymbolInfo*
    # endif
    ```
  - Associate `yylval` with this newly defined type.
    ```
    extern YYSTYPE yylval;
    ```
  - In lex source file you can typecast the return value as follows:
    ```
    yylval = (YYSTYPE)symbolPointer;
    /* Here symbolPointer is of type SymbolInfo* which points
    to the symbol table entry for corresponding token. */
    ```
- Define appropriate precedence and associativity of operators.
- Handle **`if-else`** ambiguity as described in Niemann's tutorial.
- Evaluate each expression and in assignment rule, update the value of the variable in symbol table and print the final result.
- For every number and identifier stored in the symbol table, print their lexemes in the output from yacc source file.
- Print error messages with line numbers for all syntax and semantic errors encountered in the input file. Hint: redefine **`void yyerror(char*)`**
- At the end of parsing, print the complete symbol table.

Step 2: Generating 8086 Assembly Code

Enhance your parser from Step 1 to produce assembly code for the following:

- Expressions with ADDOP, MULOP and RELOP
- Statements with **`while/if-else`**
- Define a function **`newTemp()`** that returns a new temporary whenever it is called
- Define a function **`newLabel()`** that returns a new label on each call
- Add a new variable ***`code`*** to your class **`SymbolInfo`**
- Write an assembly method for printing the value of an **ID** in the console which you may need to use if the input PASCAL program contains a write statement.
- There might be some other parts that your assembly code must contain (e.g., stack initialization). Print those directly into the output file from the main function in yacc

5 Hints for Step 2

Following hints may be useful:

- Recall your expertise on Assembly Language and practice writing some small programs with **`while/if-else,`** ADDOP, MULOP and RELOP

  Example:

  Source Program

```
program example(input, output);

var x, y : integer;
var f : real;
var c : array [3 .. 10] of integer;

begin
      x := 3;
      y := 10;

      while x < y do
      begin
            c[x] := x;
            x := x + 1
      end
end
```

  Assembly Equivalent: Do it yourself

- Consider, intermediate code generation from a parse/syntax tree.
  - Each non-terminal symbol must have two things associated with it
    - An address (i.e., a temporary variable) to hold its value
    - A variable to hold the code to obtain this value from its child expression applying the operators
  - You can modify the SymbolInfo class as follows:

    class SymbolInfo{
    char * key; // holds lexeme value as well as temporary variable name
    char * type;
    char * code; // holds the generated code
    };

  - For each non-terminal that produce an operation as the following rule:

    `simple_expression → simple_expression ADDOP term`

    create a new temporary variable to hold the value of it and assign the **`key`** attribute with it. Then place the code that produces this operation in assembly in the **`code`** attribute. Your yacc code may look like this:

```
simple_expression : simple_expression ADDOP term  {
  SymbolInfo temp;
  temp.key = newTemp();
  temp.code = concatenate(temp.code,$1.code,$3.code);
  temp.code = concatenate(temp.code,AssemblyCodeOfADDOP);
  $$ = temp;
}
```

  - For production that has no operation, needs no temporary variable to be created. For example: **`term → factor`**
  - Code for **`while/if-else`** is a bit tricky; you will need some jumps and some labels.
  - At statement level, no address information is required, hence, no **`key`** attribute is used, and only **`code`** attribute is used.

6 Instruction for Compilation

Execute the following commands sequentially:

- **yacc –d parser.y**
- **g++ -c –o parser.o y.tab.c**
- **lex –t lexicalAnalyzer.l > lexicalAnalyzer.c**
- **g++ -c –o lexicalAnalyzer.o lexicalAnalyzer.c**
- **g++ -o myParser lexicalAnalyzer.o parser.o –ll -ly**
- **./myParser <input_file> <output_file_1> <output_file_2>**

7 Submission Deadline

You must submit your assignment on the next lab class. No late submissions will be accepted.

8 Important Issue

The names of the students who will copy and help others in copying will be published on the notice board with student numbers and they will be given severe penalty. We hope we will find no one like that. Try yourself to complete the assignment. If you cannot understand anything regarding the assignment, ask for clarifications to the subject teachers. We will appreciate it if you show us your own work in the lab.