Introduction to Compiler Design by Prof. Yi-Ping You

Project Assignment 2

Syntactic Definitions

Due Date: 11:59PM, November 14, 2018

Your assignment is to write an LALR(1) parser for the C−(C minus) language. You will have to write the grammar and create a parser using **yacc**. In the third assignment, you will do some simple checking of semantic correctness. Code generation will be performed in the last phase of the project.

1 Assignment

You must create an LALR(1) grammar using **yacc**. You need to write the grammar following the syntactic in the following sections. Once the LALR(1) grammar is defined you can then execute **yacc** to produce a C program called "**y.tab.c**", which contains the parsing function **yyparse()**. You must supply a main function to invoke **yyparse()**. The parsing function **yyparse()** calls **yylex()**. You will have to revise your scanner function **yylex()**.

1.1 What to Submit

You should submit the following items:

- a revised version of your lex scanner
- your yacc parser
- a report: a file describing what changes you have to make to your scanner since the previous version you turned in, the abilities of your parser, the platform to run your parser, and how to run your parser.
- a Makefile in which the name of the output executable file must be named '**parser**' (**Please make sure it works well. TAs will rebuild your parser with this makefile. No further grading will be made if the** *make* **process fails or the executable '***parser***' is not found.**)

1.2 Implementation Notes

Since **yyparse()** wants tokens to be returned back to it from the scanner, you should modify the scanner to pass token information to **yyparse()**. For example, when the scanner recognizes an identifier, the action should be revised as

```
([A-Za-z])([A-Za-z0-9])* { tokenString("id", yytext); return ID; }
   /* Note that the symbol `ID` is defined by the yacc parser */
```

2 Syntactic Definitions

2.1 Program Units

The two program units are the *program* and the *functions*.

Program

A program has the form:

<zero or more variable, constant, and function declarations and function definitions>

Note that there should be at least one function definition existed in the program, and the function which is firstly defined is treated as the entry point of the program execution. There are two types of variables in a program:

- global variables
  declared outside the function definitions, and
- local variables
  declared inside the function definitions.

Function

A function declaration has the following form:

type identifier (<zero or more formal arguments>) ;

A function definition has the following form:

type identifier (<zero or more formal arguments>)
<one compound statement>

Parentheses are required even if no arguments are declared. No functions may be declared inside a function.

The formal arguments are declared in a formal argument section, which is a list of declarations separated by commas. Each declaration has the form:

type identifier

That is, exactly one identifier appears after the type specification. Note that if arrays are to be passed as arguments, the formal parameter must include the declared sizes of all dimensions. All arguments are passed by values. Functions may return one value or no value at all. Consequently, the return value declaration is a simple type name. If no return value is declared, the type name must be **void** because void is not a scolar data type in C- -. A function that returns no value can be called a "procedure". For example, following are valid function declarations:

```
bool func1(int x, int y[2][8], string z);
string func2(bool a);
void func3();         // procedure
void func4(double b); // procedure
int func5();
```

2.2 Data Types and Declarations

The five predefined scalar data types are **int**, **double**, **float**, **string**, and **bool**. The only structured type is the array. A variable declaration has the form:

scalar_type identifier_list;

where *identifier_list* is a list of identifiers separated by commas:

identifier, *identifier*, ..., *identifier*

Note that an identifier can be initialized with the value of an expression:

identifier = *expression*

In addition, an identifier can also appear as the form

identifier [*integer_constant*] [*integer_constant*] [...]

integer_constant is **an** integer literal.

to represent an array declaration.

Note that an array can also be initialized in the following form:

identifier [*integer_constant*] [*integer_constant*] [...] = *initial_array*

where *initial_array* has the form:

{*zero or more expressions separated by comma* }

A constant declaration has the form:

const *scalar_type const_list*;

const_list is a list of identifiers with initialized values separated by commas:

identifier = *literal_constant*, *identifier* = *literal_constant*, ..., *identifier* = *literal_constant*

where *literal_constant* is a constant of the proper type (e.g. an integer constant, string constant, etc.). "Note that assignments to constants are not allowed and constants can not be declared in terms of other named constants." The quoted note can be ignored in this assignment and later processed by the semantic analyzer.

Note that identifiers in *const_list* do not have array form.

2.3 Statements

There are seven distinct types of statements: compound, simple, conditional, while, for, jump, and procedure call.

2.3.1 compound

A compound statement consists of a block of statements delimited by a pair of braces, and an optional variable and constant declaration section:

{
<zero or more variable and constant declarations>
<zero or more statements>
}

there is no strict order between variable/constant declaration and statememts.

Note that declarations inside a compound statement are local to the statements in the block and no longer exist after the block is exited.

2.3.2 simple

The simple statement has the form:

variable_reference = *expression*;

or

print *variable_reference*; or **print** *expression*;

or

read *variable_reference*;

A *variable_reference* can be simply an *identifier* or an *array_reference* in the form of

identifier [*expression*] [*expression*] [...]

expressions
Expressions are written in infix notation, using the following operators with the precedence:

(1) `-` (unary)
(2) `* / %`
(3) `+ -`
(4) `< <= == >= > !=`
(5) `!`
(6) `&&`
(7) `||`

Note that:

1. All operator are left-associative except `-` (unary) and `!`.
2. The token '`-`' can be either the binary subtraction operator, or the unary negation operator. Parentheses may be used to group subexpressions to dictate a different precedence. Valid components of an expression include literal constants, variable names, function invocations, and array reference in the form of

   identifier [*expression*] [*expression*] [...]

3. The part of semantic checking will be handled in the next assignment. In this assignment, you don't need to check semantic errors like '`a = 3 + true;`'. Just take care of syntactic errors!

function invocation
A function invocation has the following form:

identifier (<zero or more expressions separated by commas>);

2.3.3 conditional

The conditional statement may appear in two forms:

if (*boolean_expression*) {
<zero or more variable and constant declarations>
<zero or more statements>

} **else** {
<zero or more variable and constant declarations>
<zero or more statements>
}

there is no strict order between variable/constant declarations and statements.

or

if (*boolean_expression*) { <zero or more statements and variable/constant declarations> }

2.3.4 while

The while statement has the form:

while (*boolean_expression*) {
<zero or more variable and constant declarations>
<zero or more statements>
}

or

do {
<zero or more variable and constant declarations>
<zero or more statements>
} **while** (*boolean_expression*) ;

there is no strict order between variable/constant declarations and statements.

2.3.5 for

The for statement has the form:

for (*initial_expression* ; *control_expression* ; *increment_expression*) {
<zero or more variable and constant declarations>
<zero or more statements>
}

there is no strict order between variable/constant declarations and statememts.

Grammatically, the three components of a **for** loop are **expressions**. Most commonly *initial_expression* and *increment_expression* are assignments or function calls and *control_expression* is a relational expression. Any of the three parts can be omitted, although the semicolons must remain.

2.3.6 jump

The jump statements have three types: **return**, **break**, and **continue**. They have the forms:

return *expression* ;

or

break;

or

continue;

2.3.7 function invocation

A procedure is a function that has no return value. A procedure call is then an invocation of such a function. It has the following form:

identifier (<zero or more expressions separated by commas>) ;

3 Online Parser

If you have any problems about what the parser should output, check out the following online parser, which reads your input and provides you a standard output.

```
http://sslab.cs.nctu.edu.tw/compiler/online_parser
User name:Compiler-f18
Password:81f-relipmoC
```

4 What Should Your Parser Do?

The parser should list information according to **Opt_Source** and **Opt_Token** options (the same as Project 1). If the input file is syntactically correct, print

```
|------------------------------|
| There is no syntactic error! |
|------------------------------|
```

Once the parser encounters a syntactic error, print an error message in the form of

```
|--------------------------------------------------------------------------
| Error found in Line #[line number where the error occurs]:  [source code of that line]
|
| Unmatched token:  [the token that is not recognized]
|--------------------------------------------------------------------------
```

and stop parsing immediately (error recovery is not required).

Sample(s) will be later announced at the course forum.

5 *yacc* Template

This template (`yacctemplate.y`) will be later announced at the course forum as well. Note that you should modify your lex scanner and extend this template to generate your parser.

6 How to Submit the Assignment?

Create a directory, named "YourID" and store all files of the assignment under the directory. Zip the directory as a single archive, name the archive as "YourID.zip". Upload the zipped file to the **e-Campus (E3)** system.

Note that the penalty for late homework is **15% per day** (weekends count as 1 day). Late homework will not be accepted after sample codes have been posted. In addition, homework assignments must be individual work. If I detect what I consider to be intentional plagiarism in any assignment, the assignment will receive **zero credit**.