Introduction to Compiler Design by Prof. Yi-Ping You

Project Assignment 2

Syntactic Definitions

Due Date: 11:59PM, December 1, 2015

Your assignment is to write an LALR(1) parser for the $\mathcal{P}$ 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:

> *identifier*;
> <zero or more variable and constant declaration>
> <zero or more function declaration>
> <one compound statement>
> **end** *identifier*

A program has no arguments, and hence no parentheses are present in the header. There are two types of variables in a program:

- global variables
  declared after the identifier but before the compound statement
- local variables
  declared inside the compound statement and functions

Function

A function declaration has the following form:

> *identifier* (<zero or more formal arguments>): *type*;
> <one compound statement>
> **end** *identifier*

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 declaration separated by semicolons. Each declaration has the form

> *identifier_list: type*

where *identifier_list* is a list of identifier separated by comma:

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

At least one identifier must appear before each colon, which is followed by exactly one type specification. Note that if arrays are to be passed as arguments, they must be fully declared. All arguments are passed by values.

Functions may return one value or no value at all. Consequently, the return value declaration is either a simple type name or is empty. If no return value is declared, there is no colon before the terminating semicolon. A function that returns no value can be called a "procedure". For example, following are valid function declaration headers:

```
func1(x, y: integer; z: string): boolean;
func2(a: boolean): string;
func3();        // procedure
func4(b: real); // procedure
func5(): integer;
```

2.2 Data Types and Declarations

The four predefined scalar data types are **integer**, **real**, **string**, and **boolean**. The only structured type is the **array**. A variable declaration has the form:

> **var** *identifier_list*: *scalar_type*;

or

> **var** *identifier_list*: **array** *integer_constant* **to** *integer_constant* **of** *type*;

where *integer_constant* must be a positive integer constant.
A constant declaration has the form:

> **var** *identifier_list*: *literal_constant*;

where *literal_constant* is a constant of the proper type (e.g. an integer literal, string literal, true, or false). Note that assignments to constants are not allowed and constants can not be declared in terms of other named constants. Variables may not be initialized in declarations.

2.3 Statements

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

2.3.1 compound

A compound statement consists of a block of statements delimited by the keywords **begin** and **end**, and an optional variable and constant declaration section:

> **begin**
> <zero or more variable and constant declaration>
> <zero or more statements>
> **end**

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* [*integer_expression*] [*integer_expression*] [...]

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

(1) `-` (unary)

(2) `* / mod`

(3) `+ -`

(4) `< <= = >= > <>`

(5) `not`

(6) `and`

(7) `or`

Note that:

1. The token '`-`' can be either the binary subtraction operator, or the unary negation operator. Associativity is the left. 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 [*integer_expression*] [*integer_expression*] [...]

2. 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 (<expressions separated by zero or more comma>)

2.3.3 conditional

The conditional statement may appear in two forms:

if *boolean_expr* **then**
<zero or more statements>
else
<zero or more statements>
end if

or

if *boolean_expr* **then** <zero or more statements> **end if**

Note that an integer expression is not a boolean expression.

2.3.4 while

The while statement has the form:

while *boolean_expr* **do**
<zero or more statements>
end do

2.3.5 for

The for statement has the form:

> **for** *identifier* := *integer_constant* **to** *integer_constant* **do**
> <zero or more statements>
> **end do**

2.3.6 return

The return statement has the form:

> **return** *expression* ;

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* (<expressions separated by zero or more comma>) ;

3 What Should Your Parser Do?

The parser should list information according to **Opt_S** and **Opt_T** 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).

NOTES:

- Sample(s) will be later announced at the course forum.
- Incorrect Makefile will result in **zero credit** this time!
  - Makefile must be named as "Makefile" and do not have suffix names; that is, Makefile.txt or Makefile.l is not accepted.
  - Makefile is used to compile programs. No execution (e.g.. ./parser err.p) should be included in your Makefile.

4 *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.

5 How to Submit the Assignment?

You should create a directory, named "YourID" and store all files of the assignment under the directory. Once you are ready to submit your program, zip the directory as a single archive, name the archive as "YourID.zip", and upload it 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 reduced or, usually, **zero credit**.