Modula^{-}: A Simple Modula Programming Language

Programming Assignment 2

Syntactic and Semantic Definitions

Due Date: 1:20PM,Tuesday, May 21, 2019

Your assignment is to write an LALR(1) parser for the *Modula*$^{-}$ language. You will have to write the grammar and create a parser using **yacc**. Furthermore, you will do some simple checking of semantic correctness. Code generation will be performed in the third phase of the project.

1 Assignment

You first need to write your symbol table, which should be able to perform the following tasks:

- Push a symbol table when entering a scope and pop it when exiting the scope.
- Insert entries for variables, constants, and procedure declarations.
- Lookup entries in the symbol table.

You then must create an LALR(1) grammar using **yacc**. You need to write the grammar following the syntactic and semantic definitions 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:

- revised version of your **lex** scanner
- a file describing what changes you have to make to your scanner
- your **yacc** parser
  Note: comments must be added to describe statements in your program
- Makefile
- test programs

1.2 Implementation Notes

Since **yyparse()** wants tokens to be returned back to it from the scanner. You should modify the definitions of **token**, **tokenInteger**, **tokenString**. For example, the definition of **token** should be revised to:

```
#define token(t) {LIST; printf("<\%s>\n","t"); return(t);}
```

2 Syntactic Definitions

2.1 Constant and Variable Declarations

There are two types of constants and variables in a program:

- global constants and variables
  declared inside the program
- local constants and variables
  declared inside functions

Data Types and Declarations

The predefined data types are **string**, **integer**, **boolean**, and **real**.

2.1.1 Constants

A constant declaration has the form:

> **const** *identifier* **=** *constant_exp; <...; identifier* **=** *constant_exp; >*

The type of the declared constant must be inferred based on the constant expression on the right-hand side. Note that constants cannot be reassigned or this code would cause an error.

For example,

```
const s = "Hey There";
      i = -25;
      f = 3.14;
      b = true;
```

2.1.2 Variables

A variable declaration has the form:

> **var** *identifier<, ... , identifier>*: *type; <...; identifier<, ... , identifier>*: *type;>*

where *type* is one of the predefined data types. For example,

```
var s : string;
    i : integer;
    d : real;
    b : boolean;
```

Arrays

Arrays declaration has the form:

identifier<, ... , *identifier*>: **array [***num, num* **] of** *type;*
...;
identifier<, ... , *identifier*>: **array [***num, num* **] of** *type;*

For example,

```
a: array [1, 10] of integer;     // an array of 10 integer elements
b: array [0, 5] of boolean;      // an array of 6 boolean elements
f: array [1, 100] of real;       // an array of 100 float-point elements
```

2.2 Program Units

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

2.2.1 Program

A program has the form:

module *identifier*
<zero or more variable and constant declarations>
<zero or more function declarations>
begin
<zero or more statements>
end *identifier*.

where the item in the < > pair is optional.

2.2.2 Procedures

Procedure declaration has the following form:

procedure *identifier* <(formal arguments)> <: *type* >
<zero or more constant and anvariable declarations>
begin
<one or more statements>
end *identifier*;

where **:** *type* is optional and *type* can be one of the predefined types. The formal arguments are declared in the following form:

identifier : *type* <, *identifier* : *type*, ... , *identifier* : *type*>

Parentheses are not required when no arguments are declared. No procedures may be declared inside a procedure. For example,

```
module example
// constants and variables
const a = 5;
var c : integer;
```

```
// procedure declaration
procedure add(a:integer, b:integer) : integer
begin
  return a+b;
end add;

// statements
begin
  c = add(a, 10);
  print c;
end example.
```

Note that procedures with no retuen type can not be used in expressions.

2.3 Statements

There are several distinct types of statements in *Modula*$^-$.

2.3.1 simple

The simple statement has the form:

identifier := *expression*;

or

identifier[*integer_expression*] := *expression*;

or

print *expression*; or **println** *expression*;

or

read *identifier*;

or

return; or **return** *expression*;

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

(1) − (unary)

(2) `*` `/`

(3) `+` `-`

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

(5) ∼

(6) `&&`

(7) `||`

Associativity is the left. Valid components of an expression include literal constants, variable names, function invocations, and array reference of the form

A [integer_expression]

function invocation
A function invocation has the following form:

identifier < (comma-separated expressions) >

2.3.2 conditional

The conditional statement may appear in two forms:

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

or

if (*boolean_expr*) **then**
<zero or more statements>
end;

2.3.3 loop

The loop statement has the form:

while (*boolean_expr*) **do**
<zero or more statements>
end ;

2.3.4 procedure invocation

A procedure has no return value. It has the following form:

identifier <(comma-separated expressions)>;

3 Semantic Definition

The semantics of the constructs are the same as the corresponding Pascal and C constructs, with the following exceptions and notes:

- The parameter passing mechanism for procedures in call-by-value.
- Scope rules are similar to C.
- The identifier after the **end** of program or procedure declaration must be the same identifiers as the name given at the beginning of the declaration.
- Types of the left-hand-side identifier and the right-hand-side expression of every assignment must be matched.
- The types of formal parameters must match the types of the actual parameters.

4 *yacc* Template (yacctemplate.y)

```
%{
#define Trace(t)        if (Opt_P) printf(t)
int Opt_P = 1;
%}

/* tokens */
%token SEMICOLON

%%
program:        identifier semi
                {
                Trace("Reducing to program\n");
                }
                ;

semi:           SEMICOLON
                {
                Trace("Reducing to semi\n");
                }
                ;
%%
#include "lex.yy.c"

yyerror(msg)
char *msg;
{
    fprintf(stderr, "%s\n", msg);
}

main()
{
    yyparse();
}
```