Scala$^-$: A Simple Scala Programming Language

Programming Assignment 2

Syntactic and Semantic Definitions

Due Date: 1:20PM,Tuesday, June 9, 2020

Your assignment is to write an LALR(1) parser for the *Scala*$^-$ 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 object
- local constants and variables
  declared inside methods

Data Types and Declarations

The predefined data types are **char**, **string**, **int**, **boolean**, and **float**.

2.1.1 Constants

A constant declaration has the form:

val *identifier* <: *type* > **=** *constant_exp*

where the item in the < > pair is optional, and then 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,

```
val s = "Hey There"
val i = -25
val f = 3.14
val b:boolean = true
```

2.1.2 Variables

A variable declaration has the form:

var *identifier* <: *type;*>< **=** *constant_exp* >

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

```
var s : string
var i = 10
var d : real
var b: boolean = false
```

Arrays

Arrays declaration has the form:

var *identifier* **:** *type* **[** *num* **]**

For example,

```
var a: int [10]           // an array of 10 integer elements
var b: boolean [5]        // an array of 6 boolean elements
var f: float [100]        // an array of 100 float elements
```

2.2 Program Units

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

2.2.1 Program

A program has the form:

> **object** *identifier* {
> <zero or more variable and constant declarations>
> one or more method declarations
> }

where the item in the < > pair is optional. Every *Scala*$^{-}$ program has at least one method, i.e. the `main` method: `def main() { }`.

2.2.2 Methods

Method declaration has the following form:

> **def** *identifier* (<formal arguments >) <: *type* >
> {
> <zero or more constant and variable declarations>
> <zero or more statements>
> }

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 methods may be declared inside a method. For example,

```
object example {
  // constants and variables
  val a = 5
  var c : int
  // procedure declaration
  def add(a:int, b:int) : int
  {
    return a+b
  }

  // main statements
  def main ( )
  {
    c = add(a, 10)
    println (c)
  }
}
```

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

2.3 Statements

There are several distinct types of statements in *Scala*$^{-}$.

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 block

A block is a collection of statements enclosed by { and }. The simple statement has the form:

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

2.3.3 conditional

The conditional statement may appear in two forms:

> **if** (*boolean_expr*)
> a block or simple statement
> **else**
> a block or simple statement

or

> **if** (*boolean_expr*)
> a block or simple statement

2.3.4 loop

The loop statement has two forms:

> **while** (*boolean_expr*)
> a block or simple statements

or

> **for** (*identifier* <**-** *num* **to** *num*)
> a block or simple statement

2.3.5 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();
}
```