Kotlin$^-$: A Simple Kotlin Programming Language

Programming Assignment 2

Syntactic and Semantic Definitions

Due Date: 1:20PM, Tuesday, May 17, 2022

Your assignment is to write an LALR(1) parser for the *Kotlin*$^-$ 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 and blocks

Data Types and Declarations

The predefined data types are **int**, **string**, **bool**, 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 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:string = "Hey There"
val i = -25
val f = 3.14
val b:bool = 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. When both the type attribute declaration, i.e : *type* and initialization are omitted from variable declarations, the default data type is **int**. For example,

```
var s: string
var i = 10
var d: float
var b: bool = false
```

Arrays

Arrays declaration has the form:

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

For example,

```
var a: integer [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 *procedures*.

2.2.1 Program

A program has the form:

> **class** *identifier* {
> < variable and constant declarations or function declarations >
> }

where the item in the < > pair is optional. Every *Kotlin*$^{-}$ program must have at least one method, i.e. the `main` method.

2.2.2 Functions

Function declaration has the following form:

> **fun** *identifier* (<formal arguments >) < **:** *type* >
> *block*

where *block* is a block statement (see Section 2.3.2), **:** *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 required even when no arguments are declared. No functions may be declared inside a function. For example,

```
class example {
  val a = 5
  var c : int

  fun add (a: int, b: int) : int {
    return a+b
  }

  fun main() {
    c = add(a, 10)
    if (c > 10)
      print -c
    else
      print c
    println ("Hello World")
  }
}
```

Note that functions with no retuen type are generally called procedures and can not be used in expressions, whereas functions with retuen values can be used in expressions.

2.3 Statements

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

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 block statement has the form:

> **{**
> < variable and constant declarations or 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 statement

or

> **for** (*identifier* **in** *num* **. .** *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.
- 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();
}
```