# *Go*$^-$: A Simple Programming Language

## Programming Assignment 2

### Syntactic and Semantic Definitions

**Due Date: 10:20AM,Thursday, May 25, 2017**

Your assignment is to write an LALR(1) parser for the *Go*$^-$ 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 Program Units

The program units of $Go^{-}$ are the *program* and *functions*.

### 2.1.1 Program

A program has the form:

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

where the item in each $<$ $>$ pair is optional. A non-empty program needs to have a method named *main*, where the program starts.

### 2.1.2 Functions

Function declaration has the following form:

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

where *type* can be one of the predefined types or **void**, and formal arguments are declared in the form:

> *identifier type* $<$, *identifier type*, ... , *identifier type* $>$

Parentheses are required even when no arguments are declared. For example,

```
// global variables
var c int
var a int = 5

// function declaration
func int add (a int, b int)
  return a+b
end

// main statements
func void main ( ) {
  c = add(a, 10)
  print c
}
```

Note that functions with the **void** retuen type are usually called as procedures and can not be used in expressions.

## 2.2 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 **boolean**, **integer**, **real**, and **string**.

### 2.2.1 Constants

A constant declaration has the form:

**const** *identifier* **=** *constant_exp*

where the data type of the constant identifier is determined by the data type of the right-hand-side constant. Note that constants are immutable. In other words, constants cannot be reassigned or this code would cause an error.

For example,

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

### 2.2.2 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 = "Hey There"
var i int
var d real
var b bool = true
```

### Arrays

Arrays declaration has the form:

**var** *identifier* [ *constant_exp* ] *type*

For example,

```
var a [10]int          // an array of 10 integer elements
var b [5]bool          // an array of 5 boolean elements
var f [100]real        // an array of 100 float-point elements
```

## 2.3 Statements

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

### 2.3.1 Simple Statements

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) &
(8) |

Note that the − token can be either the binary subtraction operator, or the unary negation operator. Associativity is 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 of the form

A [ integer_expression ]

**Function Invocation**
A function invocation has the following form:

*identifier* ( <comma-separated expressions> )

### 2.3.2 Compound

A compound statement consists of a block of declarations and statements that are enclosed by the delimiters { and }:

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

### 2.3.3 Conditional

The conditional statement may appear in two forms:

> **if** ( *boolean_expr* )
> <simple or compound statement>
> **else**
> <simple or compound statement>

or

> **if** ( *boolean_expr* )
> <simple or compound statement>

### 2.3.4 Loop

The loop statement has the form:

> **for** ( <zero or one statement ;> *boolean_expr* <; zero or one statement> )
> <simple or compound statement>

### 2.3.5 Procedure Invocation

A procedure is a function that has no return value. It has the following form:

> **go** *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. Furthermore, the types of formal parameters must match the types of the actual parameters.
- 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.

# 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();
}
```