

# *Ada*<sup>-</sup>: A Simple Ada Programming Language

## Programming Assignment 2

### Syntactic and Semantic Definitions

**Due Date: 1:20PM, Tuesday, May 25, 2021**

Your assignment is to write an LALR(1) parser for the *Ada*<sup>-</sup> 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 procedures and blocks

#### Data Types and Declarations

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

##### 2.1.1 Constants

A constant declaration has the form:

*identifier* : **constant** <: *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,

```
s: constant := "Hey There";
i: constant := -25;
f: constant : float := 3.14;
b: constant := true;
```

##### 2.1.2 Variables

A variable declaration has the form:

*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,

```
s: string;
i := 10;
d: float;
b: boolean = false;
```

#### Arrays

Arrays declaration has the form:

*identifier* : *type* [ *num* ] ;

For example,

```
a: integer [10];           // an array of 10 integer elements
b: boolean [5];            // an array of 6 boolean elements
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:

```
program identifier
<declare
zero or more variable and constant declarations>
<zero or more procedure declarations>
begin
<zero or more statements>
end
end identifier
```

where the item in the < > pair is optional.

### 2.2.2 Procedures

Procedure declaration has the following form:

```
procedure identifier < ( formal arguments ) > < return type >
block
end identifier ;
```

where *block* is a block statement (see Section 2.3.2), **return** *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,

```
program Example
-- constant and variable declaration
declare
  a: integer := 5;
  c: integer;
-- function declaration
procedure add (a: integer; b: integer) return integer
begin
  return a+b;
end;
end add;
-- main block
begin
  c := add(a, 10);
  println c;
end
end Example
```

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

## 2.3 Statements

There are several distinct types of statements in *Ada*<sup>-</sup>.

### 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) not
- (6) and
- (7) or

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 **begin** and **end** with an optional **declare** section. The simple statement has the form:

```
< declare
  zero or more variable and constant declarations>
begin
  <one or more statements>
end ;
```

### 2.3.3 conditional

The conditional statement may appear in two forms:

```
if boolean_expr then
  a block or simple statement
else
  a block or simple statement
end if ;
```

or

```
if boolean_expr then
  a block or simple statement
end if ;
```

### 2.3.4 loop

The loop statement has two forms:

```
while boolean_expr loop
  a block or simple statement
end loop ;
```

or

```
for ( identifier in num . . num )
  a block or simple statement
end loop ;
```

### 2.3.5 procedure invocation

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

```
identifier < ( semicomma-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 is call-by-value.
- Scope rules are similar to C.
- The identifier after the **end** of program or procedure declaration must be the same identifier 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();
}
```