sT: A Simple *Turing* Programming Language

Programming Assignment 2

Syntactic and Semantic Definitions

Due Date: 1:20PM, Wednesday, May 25, 2023

Your assignment is to write an LALR(1) parser for the *sT* 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 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(s,t) {LIST; printf("<%s>\n",s); return(t);}
```

2 Syntactic Definitions

2.1 Data Types and Declarations

The predefined scalar data types are **bool**, **real**, **int**, and **string**. The only structured type is the *array*. There are two types of constants and variables in a program:

- global constants and variables
  declared outside functions and procedures
- local constants and variables
  declared inside functions, procedures, or blocks

Constants

A constant declaration has the form:

const *identifier* ⟨ :*type* ⟩ := *constant_exp*

where *constant_exp* is an expression of constant operands and ⟨ ⟩ denotes that type declaration is optional. Note that assignments to constants are not allowed after declaration. For example,

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

Variables

A variable declaration has the form:

var *identifier* ⟨ :*type* ⟩ := *constant_exp*

where the type declaration could be optional. Another form of variable declaration has the format:

var *identifier* :*type* ⟨ := *constant_exp* ⟩

where the value initialization could be optional. For example,

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

Arrays

Arrays can be declared by

var *identifier* : **array num .. num of** *type*

For example,

```
var a: array 1..10 of int       // an array of 10 integer elements
var b: array 0..5 of bool       // an array of 6 boolean elements
var f: array 1..100 of real     // an array of 100 float elements
```

2.2 Program Units

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

2.2.1 Program

A program has the form:

⟨ zero or more variable, constant, or function declarations ⟩
⟨ zero or more statements ⟩

where the item in the ⟨ ⟩ pair is optional.

2.2.2 Functions and Procedures

A function declaration has the following form:

function *identifier* ⟨ (zero or more formal arguments separated by comma) ⟩ **:** *type*
⟨ zero or more variable, or constant declarations and statements ⟩
end *identifier*

Parentheses are not required if no arguments are declared. No functions may be declared inside a function.

The formal arguments are declared in a formal argument section, which is a list of declaration separated by comma. Each declaration has the form

identifier **:***type*

Note that if arrays are to be passed as arguments, they must be fully declared. All arguments are passed by values.

Procedures are similar to functions, but return no value at all.

procedure *identifier* ⟨ (zero or more formal arguments separated by comma) ⟩
⟨ zero or more variable or constant declarations and statements ⟩
end *identifier*

For example, following are valid function or procedure declaration headers:

```
function func1(x :int, y :int, z :string) :bool
function func2(a :bool):int
procedure func3
```

The name of every function must be unique. For example, the following is a program:

```
% Example
% constant and variable declaration
const a :int := 5
var c :int

% function decclaration
function add (a :int, b :int) :int
  result a+b
end add
```

```
/% main statements
c := add(a, 10)
if (c > 10) then
  put -c
else
  put c
end if
put ("Hello World")
```

2.3 Statements

There are several distinct types of statements.

2.3.1 blocks

A block consists of a sequence of declarations and statements delimited by the keywords **begin** and **end**.

> **begin**
> ⟨zero or more variable or constant declarations and statements⟩
> **end**

Note that variables and constants that are declared inside a block are local to the statements in the block and no longer exist after the block is exited.

2.3.2 simple

The simple statement has the form:

> *identifier* := *expression*
> or
> **put** *expression*
> or
> **get** *identifier*
> or
> **result** *identifier* or **return**
> or
> **exit** ⟨ **when** *bool_expression* ⟩
> or
> **skip**

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

(1) − (unary)
(2) `* / mod`
(3) `+ -`
(4) `< <= = => > not=`
(5) `not`
(6) `and`
(7) `or`

Note that the − token can be either the binary subtraction operator, or the unary negation operator. All binary operators are left associative, except for the uniary operators in list (1). 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 ⟨ (expressions separated by zero or more comma) ⟩

2.3.3 conditional

The conditional statement may appear in two forms:

if *boolean_expr* **then**
⟨ zero or more variable or constant declarations and statements ⟩
else
⟨ zero or more variable or constant declarations and statements ⟩
end if

or

if *boolean_expr* **then**
⟨ zero or more variable or constant declarations and statements ⟩
end if

2.3.4 loop

The loop statement has the form:

loop
⟨ zero or more variable or constant declarations and statements ⟩
end loop

or

for ⟨ **decreasing** ⟩ *identifier* **: num .. num**
⟨ zero or more variable or constant declarations and statements ⟩
end for

2.3.5 procedure invocation

A procedure is a function that has no return value. A procedure call is then an invocation of such a function. It has the following form:

identifier ⟨ (zero or more expressions separated by comma) ⟩

where the parentheses should not be included when there are no actual arguments.

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.
- Two arrays are considered to be the same type if they have the same number of elements and the types of elements are the same.
- 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.
- Type declaration of a function must be matched with the type of its return value. Furthermore, the types of formal parameters must match the types of the actual parameters.

4 Example sT Program

```
{% Sigma.st
 %
 % Compute sum = 1 + 2 + ... + n
 %}

% constant and variable declarations
const n :int := 10
var sum :int
var index :int

% for loop
sum := 0
for index: 1 .. n
  sum := sum + index
end for
put "The result is "
put sum
skip

% loop
sum := 0
index := 1
loop
  sum := sum + index
  index := index + 1
  exit when index = n
end loop
put "The result is "
put sum
```

5 *yacc* Template

```
%{
#include <stdio.h>
#define Trace(t)        printf(t)
%}

/* tokens */
%token SEMICOLON

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

semi:           SEMICOLON
                {
                Trace("Reducing to semi\n");
                }
                ;
%%
#include "lex.yy.c"
FILE    *yyin;         /* file descriptor of source program */

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

main(int argc, char *argv[])
{
    /* open the source program file */
    if (argc != 2) {
        printf ("Usage: sc filename\n");
        exit(1);
    }
    yyin = fopen(argv[1], "r");         /* open input file */

    /* perform parsing */
    if (yyparse() == 1)                 /* parsing */
        yyerror("Parsing error !");     /* syntax error */
}
```