sD: A Simple D Programming Language

Programming Assignment 2

Syntactic and Semantic Definitions

Due Date: 1:20PM, Wednesday, May 13, 2025

Your assignment is to write an LALR(1) parser for the sD 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**, **float**, **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
- local constants and variables
  declared inside functions and blocks

Constants

A constant declaration has the form:

const *type identifier* = *constant_exp* ;

where *constant_exp* is an expression of constant operands. Note that assignments to constants are not allowed after declaration. For example,

```
const string s = "Hey There";
const int i = 25;
const float f = 3.14;
const bool b = true;
```

Variables

A variable declaration has the form:

type identifier_list ;

where *identifier_list* is a list of identifier declarations separated by comma:

identifier_decl, identifier_decl, . . ., identifier_decl

and *identifier_decl* is an identifier with an optional initialization

identifier ⟨ = *constant_exp* ⟩

For example,

```
string s;
int i = 10, j = 20;
float d;
bool b = false;
```

Arrays

Arrays can be declared by

type identifier[*integer_constant*] · · ·[*integer_constant*];

For example,

```
int a[10];      // an array of 10 integer elements
bool b[6];      // an array of 6 boolean elements
float f[100];   // an array of 100 real 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 >

where the item in the < > pair is optional. Every sD program must have at least one function, i.e. the `main` method.

2.2.2 Functions

A function declaration has the following form:

type identifier (⟨zero or more formal arguments separated by comma ⟩)
block

Parentheses are required even 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

type identifier

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

Functions may return one value or no value at all. Consequently, the return value declaration is either a simple type name or is `void`. A function that returns no value can be called a "procedure". For example, following are valid function declaration headers:

bool func1(int x, int y, string z)
int func2(bool a)
void func3( )

The name of every function must be unique. Furthermore, there must be one function name *main* with the `void` type. The program will start from the *main* function. For example, the following is a program:

```
const int a = 5;
int c;

int add (int a, int b) {
  return a+b;
}

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

2.3 Statements

There are six distinct types of statements.

2.3.1 blocks

A block consists of a sequence of declarations and statements delimited by the { and }. section:

{
⟨zero or more variable and constant declarations and statements⟩
}

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
print *expression* ; or **println** *expression* ;
or
read *identifier* ;
or
identifier++ ; or *identifier*−− ;
or
;

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

(1) ++ −− − (unary)
(2) * / %
(3) + −
(4) < <= == => > !=
(5) !
(6) &&
(7) ||

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] [. . .] [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*)
> one simple statement or a block
> **else**
> one simple statement or a block

or

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

2.3.4 loop

The loop statement has the form:

> **while** (*boolean_expr*)
> one simple statement or a block

or

> **for** (one simple statement; *boolean_expr*; one simple statement)
> one simple statement or a block

or

> **foreach** (*identifier* **:** `num .. num`)
> one simple statement or a block

Note that the semicolons of the simple statements in the **for** declaration will be omitted.

2.3.5 return

The return statement has the form:

> **return** *expression* ;

2.3.6 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 must be included even if 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.
- There must be a function name *main*. The program will start excution from the *main* function.

4 Example sD Program

```
/* Sigma.sd
 *
 * Compute sum = 1 + 2 + ... + n
 */

// constants and variables
const int n = 10;
int sum = 0;

// main function
void main()
{
    int index;

    // for loop
    for (index = 1; index <= n; index++)
    {
      sum = sum + index;
    }
    print "The sum is ";
    println sum;

    // foreach loop
    foreach (index : n .. 1)
    {
      sum = sum + index;
    }
    print "The total sum is ";
    println 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 */
}
```