BITS Pilani, Hyderabad Campus
Department of Computer Sc. and Information Systems
Second Semester, 2024-25
CS F363 Compiler Construction
Project

1 Language Specifications

The toy language is made up of the following alphabet.

1.1 Alphabet

- **Lowercase alphabets:**

```
a, b, c, d, ..., z
```

- **Digits** :

```
0, 1, 2, ..., 9
```

- **Special symbols:**

```
+ - % / * < > = _ ( ) ; , (comma) : { }
```

1.2 Constants

Constants in a language represent fixed values that do not change during the execution of a program. Types of constants included in this language are as given below.

- `Integer Constants`:
  - Decimal constants: made of the digits `0, 1, 2, ..., 9`.
  - Binary constants: made of the digits `0` and `1`.
  - Octal constants: made of the digits `0, 1, 2, ..., 7`.

  Every integer will be represented as a pair `(value, base)` where the value can be a decimal constant, binary constant, or octal constant, and the base is from the set $\{2, 8, 10\}$.

- `Character constants`: A character constant in C is a single character enclosed in single quotes (‘ ’). It represents an integer value corresponding to the ASCII or Unicode encoding of the character.

  Examples:

```
‘a’  // Represents the character ‘a’
‘Z’  // Represents the character ‘Z’
```

```
'1'  // Represents the character '1'
'\n' // Represents the newline character
'\t' // Represents a tab character
```

- `String constant`: A string is a sequence of characters in the alphabet enclosed in double quotes (`"and "`).

  Example: "Welcome", "Welcome to the CS F363 course", "1+2bc", etc.

1.3 Keywords

The toy language provides the following keywords:

```
int, float, if, else, while, for, main, begin, end, input, output, program,
    VarDecl, inc, dec
```

Keywords cannot be used as variable names.

1.4 Data types

The language allows only two types of data; `int` (integer values) and `char` (character constants).

1.5 Variables and Identifiers

An identifier in the toy language must begin with a lowercase letter ($a - z$) and contain only lowercase letters, digits ($0 - 9$), and underscores (_). However, at most, one underscore (_) is allowed.

Example:

```
Valid variable names: age, count, tax_12, net_income, is_ready;

Invalid variable names: _sum, sum_curr_total, 1sum;
```

Declaration of variables: All variables in the program must be defined within a block called `VarDeclaration` as given below:

```
begin VarDecl:
    (var_1, type); //var_1 is the name of the variable, and type is int or char.
    (var_2, type); //var_2 is the name of the variable, and type is int or char.
    ....
    ....
    ....
    (var_k, type); //var_k is the name of the variable, and type is int or char.
end VarDecl
```

All variables must be defined in the block that starts with `begin VarDecl:` and ends with `end VarDecl`. Each variable must be defined as an ordered pair with the first element as the name of the variable and the second element as its type, `int or char`. Note that each pair ends with a semicolon `(;)`; a line may contain several declarations of variables.

Note that in the variable declaration, assigning a value to one or more variables is not allowed.

A variable cannot be used in the program if it is not defined in this block.

A variable name cannot be the same as any of the keywords.

1.6 Operators

- **Arithmetic Operators**:

```
+  //adition
-  // subtraction
*  // multiplication
/  // division ; only takes quotient
%  // remainder
```

  Note that these operators are allowed only on int data type, and we are allowing three different integer type data (decimal, binary, and octal). So, when performing any athematic operation on two integers, the result must be of the highest precedence type; decimal is higher order than octal, and octal is higher than binary.

  Some of the examples are given below:

```
(12, 10) + (11, 2) = (15, 10)
(12, 10) + (11, 8) = (21, 10)
(11, 2) + (11, 8) = (14, 8)
```

- **Relational Operators**

```
= (equal to), >, <, >=, <=, <> (not equal to)
```

  The above note also holds for relational operators.

- **Assignment Operators**

```
:=, +=, -=, *=, /=, %=
```

  The above note also holds for assignment operators.

- **Separators**

```
( ) , ; { } " @
```

1.7 Statements

- **Input and Output**: `print` is used to output a string constant or a formatted text.
  - The string constant is the same as defined before. Some examples are given below:

```
print("Welcome to CS F363");
print("Some random text for understanding");
```

– Formatted text is a combination of a text string (made of symbols in the alphabet and k (some integer greater than or equal to 1) occurrences of a special symbol `@`, between ‘‘ and ’’, followed by a list of k variables and constants. Whenever we print an integer, the value must be in decimal equivalent.

Some examples are given below:

```
print("The values are @, @, and @: ", a, b, c); //Valid print statement.
print("The value of a is : @", (10, 2)); //Valid print statement.
print("The value of a is : @", 'x'); //Valid print statement.
print("a = @,  b = @", (10, 2), (15, 8)); //Valid print statement.
print("The value of a is: @," (10, 2)) // at the end semicolon is missing,
    so invalid statement.
print("a = @,  b = @", (10, 2)); //Invalid print statement, only one value
    is listed.
```

Similarly, scan is used to read a list of variables from the user; this contains a formatted text with only a list of `@` symbols followed by the same number of variables. Further, if you are reading an integer from the user, assume that the read value is always a decimal type.

Some examples are given below:

```
scan("@, @, @", a, b, c); //Valid
scan("@, @", a, b); //Valid
scan("@, @", a, b) //Invalid since ; is missing at the end.
scan("@, @", a); //Invalid since only one variable is given in the list,
    but there are 2 @ symbols in the formatted string.
```

- **Assignment statement:** To assign a value to a variable, follow this syntax:

```
variable_name assign_operator expression;
```

where `assign_operator` and the `expression` is a constant, a variable, or an arithmetic expression over constants/variables with any combination of operators mentioned above.

- **Block of statements:** A set of one or more statements is consider as a block and each block starts with **begin** and ends with **end**.

```
begin
    statement_1;
    ....
    ....
    statement_k;

end
```

- **Conditional statements:** We allow if-then and if-then-else statements as given below.

  – **Simple if**: `if (condition) statement;`
  where `condition` is a variable or relational expression and `statement` is a compound statement (block of statements) which will execute only when the condition value is non-zero. See an example below:

```
i:= (10, 2);
if i > (20, 10) then
```

```
    begin
     i :=(1, 10);
     i := i - (1, 10);
     print("@", i);
    end;
```

– **if-then-else**: `if (condition) statement_1 else statement_2;`
where `condition` is a Boolean or relational expression, and `statement_1` (will execute if condition value is non-zero) and `statement_2` (will execute if condition value is zero) are compound statements (block of statements). See the following example.

```
i:= (10, 2);
if i > (20, 10) then
   begin
     i :=(1, 10);
     i -= (1, 10);
     print("@", i);
   end
else
   begin
       i:=(20, 2);
       print("@", i);
    end;
```

For the sake of simplicity, we do not consider nested if statements.

- **Looping statements:** We allow loops while and for doing as given below. For the sake of simplicity, we do not consider nested loops.

  – **while-do**: `while (condition) S;` where `condition` is a variable or relational expression and `S` is a compound statement (block of statements) that will run as long as the condition value is nonzero. See an example below:

```
 while (number>0)
    begin
        sum := sum + number;
        number := number - 1;
    end;
```

  – **for-do**: `for variable_name := t_1 to t_2 inc/dec t_3 do S;`
where the `variable_name` specifies the loop-parameter, also called as a control variable or index variable; `t_1` and `t_2` the integers (can be decimal, octal, or binary) (or arithmetic expressions) that the control variable can take from, `inc t_3` (resp. `dec t_3`) denotes that `t_1` is increased (resp. decrease) by `t_3` every time and subject to the limit `t_2` and S is the body of the for-do loop that is a group of statements/block statements. See examples below:

```
b:=(20, 10);
for a := (10, 8) to b+(10, 2) inc (1, 10) do
    begin
        print("The value of a = @", a);
    end;

for a := a+(5, 8) dec b-(15, 10) do
```

```
begin
    print("The value of a = @", a);
end;
```

- **Arrays**: we consider only one-dimensional arrays, and we consider a static declaration (inside the variable declaration section as defined above) of an array. The syntax is given below:

```
(array_name[t], type);
```

  where `array_name` is an identifier (variable), and `t` is an integer assumed to be a positive decimal number. This creates an array to store `t` constants of the same `type` (moreover, if the type is `int`, all values are assumed to be decimal. In addition, assume that the array indexing starts with one. See the following example.

```
(A[6], int); //valid array with name A and stores six integers of decimal type.
(B[12], char); //valid array with name B and store 12 character constants.
```

- **Comments:** the language allows `C-language` type comments; single line and multi-line comments.

2 Program structure

The program starts with `begin program:` and ends with `end program`. The main program should start with the variable declaration section; all the variables used in the program must be declared in this section. Then followed by the statement blocks, like athematic, conditional statements, looping statements, input-output, etc.

```
begin program: // main program block starts


begin VarDecl:
    (var_1, type);
    (var_2, type);
    ....
    ....
    ....
    (var_k, type);
end VarDecl

some statements that are according to the above mentioned rules.


end program // the end of the main program block
```

See the example given below.

```
begin program:

    begin VarDecl:
      (num1, int);
      (num2, int);
```

```
  (sum, int);
end VarDecl


print("Enter the first number: ");
scan("@", num1);

print("Enter the second number: ");
scan(num2);

// Perform addition
sum := num1 + num2;

// Display the result
Write("The sum = @ ", sum);

end program
```

3 Tasks

1. ***[6 marks]*** - Write a LEX program that breaks the input program into tokens: identifiers, operators, numbers, keywords, constants, etc.

   Further, print if the following lexical errors are present in the given input program;

   - an integer constant is found, but not in the correct format. That is, decimal numbers contain digits from `[0-9]`, octal contains digits from `[0-7]`, and binary contains `[0-1]`.

     ```
     (123, 10) // Valid
     (123, 8) // Valid
     (129, 8) // Invalid since 9 is not allowed in octal representation
     (1010, 2) // Valid
     (131, 2) // Invalid since 3 is not allowed in binary representation
     (124, ) // Invalid, base is missing
     ```

   - if input-output statements are not in the correct format (see examples given in the respective section).
   - if a variable is defined more than once or not defined but used in the program after the declaration section.
   - if a keyword is used as a variable.
   - if any variable name is not valid.

2. ***[4 marks]*** Write a context-free grammar that generates all the valid programs in this language.

This is the first part of the project; in the second part (post-mid semester), you will be working with yacc/bison to do syntax analysis, semantic analysis, generating three-address code, generating abstract-syntax tree, etc.

4 Submission guidelines

Will be updated by 16 Feb 2025 EoD.