SER 502 - Languages and Program Paradigms (Spring 2017) - Milestone 2

Project - Compiler and Virtual Machine for a Programming Language
Simple Programming Language(SPL)

Team 4

Manju Bisht
Dinaker Prakash Kolipaka
Vidhi Patel
Ramya Varakantham

INDEX

Language Name

We have named our language SPL - Simple Programming Language. It is simple to read, understand, and write programs in SPL. Also it provides basic mathematical operations, hence the name SPL.

Language Design

The design criteria of this language is similar to Python's, for readability and writability purposes. For people learning programming for the first time, SPL would teach the underlying fundamentals behind most programming languages since it borrows heavily from existing languages. It is an imperative programming language. Since it is not complex like most established languages, SPL can be the starting point for any novice programmer. It also solves the purpose of understanding the design of runtime environments for programming languages, especially accessing user defined program values (primarily data), managing runtime storage, and fundamental operations needed to implement a simple programming language, as mentioned in the Project requirement document. The basic features would help us understand the process of compilation, execution etc better.

Design Criteria

The design criteria for this language is Readability and Writability.

1. Readability: In SPL we have made the program more readable by including features like semicolon - to acknowledge the end of a statement. Use of semicolon also makes the language a free-format language.
   For e.g.

   ```
   int a = 10;
   int b = 20;
   ```

   The *If* and *While* blocks are written inside the curly({ }) brackets for readability purpose. An e.g. of while block is

   ```
   while ( i < 10 )
   { int x = 20;
   int y = 30;
   }
   ```

   Use of brackets provides good readability and a clear representation of what statements must be compiled and executed in case of condition evaluating to true.

2. Writability: As SPL provides only the basic mathematical operations (addition, subtraction, multiplication, division, and modulo besides comparison and assignment), the syntax is very similar to mathematical model. It provides an easy way to implement the solution clearly, correctly, concisely, and quickly. For e.g. an equation in mathematics x = 10 + 20 + y + z/10 can be written in SPL as

   ```
   x = 10 + 20 + y + z / 10;
   ```

There is no need for calculating sub-expressions, such as (10 + 20), (z/10), (30 + y) of right-hand-side and then assigning the final value to *x*. For example, in Assembly language, the same equation will require many statements as a single statement handles only one operation at a time. Though Assembly language provides faster execution, it has poor writability.

Language Processing System

As discussed in the class and specified in the Project requirement document, we have implemented a compiler and a runtime environment for the language.

The following diagram (Fig. 1) provides an overview of the language processing system.

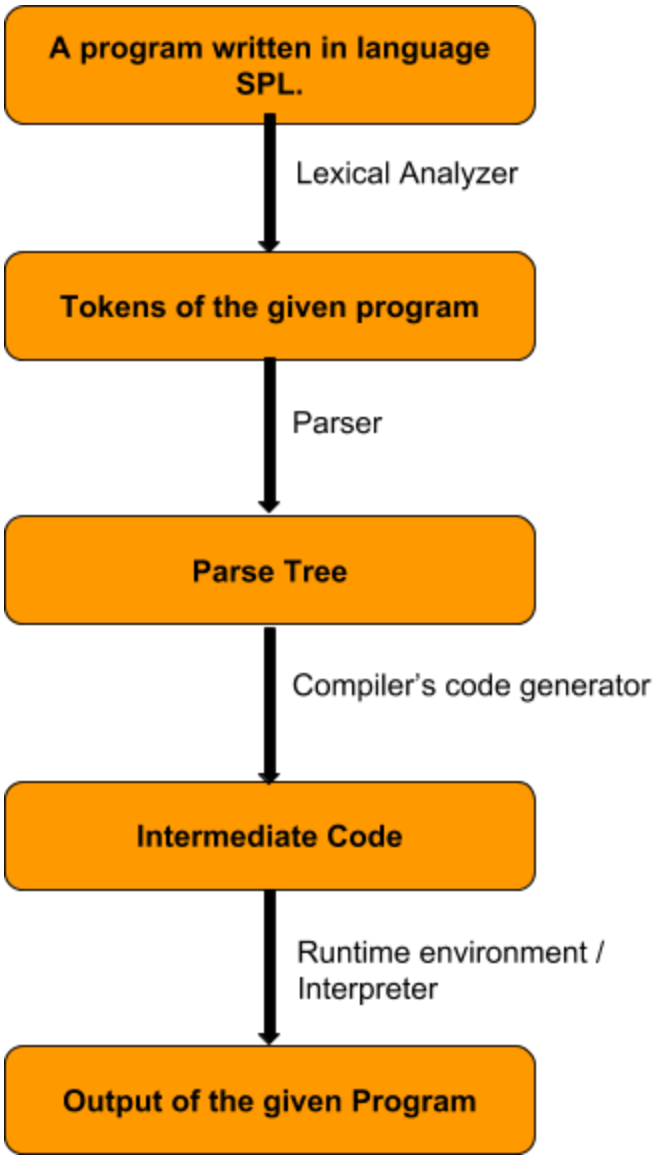


Fig. 1 - Language Processing System Overview

The input to the compiler is a program written in SPL. The program file should have an .spl extension for successful compilation and interpretation of the program. The compiler program has three parts : a lexical analyzer, a parser, and an intermediate code generator. Each part has been explained in detail in the following sections. The interpreter takes the intermediate code as the input and produces the output of the program.

Lexical Analyzer

The lexical analysis is the first step of compilation. The lexical analyzer reads the given SPL program and generates the tokens based on the Lexer specification (grammar). Language Specification contains all the tokens in the language SPL. The analyzer is hand-written in Prolog, that converts each program statement into a List of Tokens. The analyzer creates an intermediate file that contains the tokenized programs statements as Prolog Lists. The List is an obvious choice since all the tools are written in Prolog, and hence it is easy to read and write with List.

Lexer Specification

The lexer will understand the following tokens in the program which will be used to convert the SPL program to an intermediate code.

Token category	Token
Datatypes	‘int’, ‘bool’
Assignment operator	‘=’
Comparison operator	‘==’
Arithmetic operators	‘+’, ‘-’, ‘*’, ‘/’, ‘%’
Logical operators	‘<’, ‘>’, ‘<=’, ‘>=’, ‘!=’
Conditional	‘if’, ‘then’, ‘else’
Iteration	‘while’
Comments	‘#’
End of Statement	‘;’
Print	‘print’
Block Start	‘{‘
Block end	‘}’
Condition End	‘)’

Condition Start	‘(‘
Identifier	letter (letter | number)*
End of Line	‘\n’

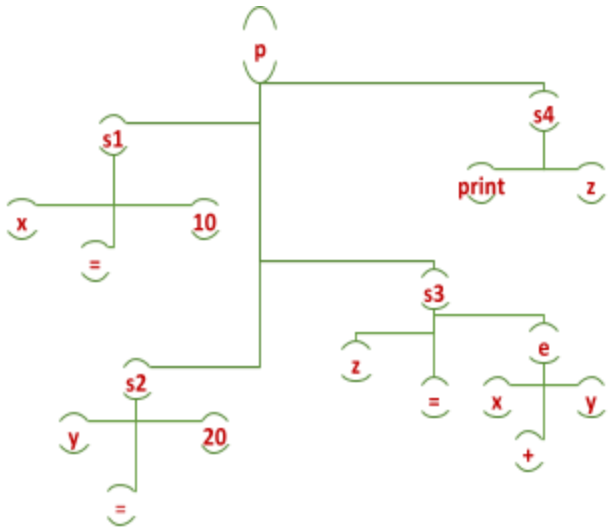
To understand the working of the lexical analyzer, let’s take a code example and generate the Tokens. There are five statements in the program, hence the analyzer will generate five lists, one for each statement.

Program	***Tokens List***
int x = 10;	*[‘int’, ‘x’, ‘=’, ‘10’, ‘;’]*
int y = 20;	*[‘int’, ‘y’, ‘=’, ‘20’, ‘;’]*
int z;	*[‘int’, ‘z’, ‘;’]*
z = x + y;	*[‘z’, ‘=’, ‘x’, ‘+’, ‘y’, ‘;’]*
print z;	*[‘print’, ‘z’, ‘;’]*

Parser

Parsing can be defined as a method of analyzing the stream of tokens as per the given grammar and understanding the meaning of each token, and thus providing the meaning to complete program. In SPL, the Parser, written in Prolog, takes the input of token lists generated by the analyzer and creates a parse tree. One parse tree corresponds to one program. For the language to be consistent, a given program should have the same meaning no matter how many times it is executed, i.e. the Parser should generate the same tree for the same program every time. The following parse tree correspond to the above given token lists.

Parse Tree



Parsing Technique

We will use recursive-descent parsing technique for our grammar. For each of the above grammar statements, the rules are applied recursively which lead to execution of the overall program. The rule for parsing will be longest substring matching which means that match for the longest substring will be evaluated.

Intermediate Code

Intermediate code generator is a part of compiler, that generates a low-level language for the interpreter. The code generator takes the parse tree as input, generates and saves the intermediate code a separate text file with the same name as the SPL file. The code generator is written in Prolog.
Currently, we are considering the parse tree as our intermediate code. Later, we might make changes to show the intermediate code in a more readable format. We will have the concept of registers to save the values of expressions. For example, for expression *x+y, x* will be loaded to register R, and value of *y* will be added to this register R, in order to get the value of the expression.

An example of intermediate code is:

Program	***Intermediate code***
int x = 10;	*(x)* *=(x, 10)*
int y = 20;	*(y),* *=(y, 10)*
int z; *z = x + y;*	*(z)* *=(R, x)* *+(R, y)*

print z;	*=(z, R)* *print(z)*

Interpreter/Runtime Environment

Interpreter is the last step in the language processing system. It takes the intermediate code as the input and generates the output. The Interpreter is also written in Prolog. Currently, the interpreter takes the parse trees as input, and evaluation predicates generate the output. Later, we might change our evaluation predicates to take low-level intermediate code as input. The evaluation predicates implement the operational semantic rules for mathematical operations, token-to-terminal conversion axioms, reduction rules for terminal and expression values, and reduction rules for if-then-else and while statements.

The environment in the Interpreter is implemented using Dictionary (Dicts in Prolog). Prolog supports the concept of dicts, as a structure with key-value pairs. The syntax for a dict is illustrated below. Tag is either a variable or an atom. As with compound terms, there is **no** space between the tag and the opening brace. The keys are either atoms or small integers.The values are arbitrary Prolog terms which are parsed using the same rules as used for arguments in compound terms.
Tag{Key1:Value1, Key2:Value2, ...}
A dict can *not* hold duplicate keys.
The following functions are defined on dicts:
Get(Key): Gets the value for the specified key

```
?- write(t{a:x}.get(a)).
x
```

Put([key=value]): Replaces value if key already exists. If not, adds a new key-value pair.

```
?- A = point{x:1, y:2}.put([x=3]).
A = point{x:3, y:2}.
```

An example of program, intermediate code, and the output (with environment).

Program	***Intermediate code***	***Output (*** E_0 *={ undef })*
int x = 10;	*(x)*	$E_1(x)$
	=(x, 10)	$E_2(x=10)$
int y = 20;	*(y)*	$E_3(x=10, y)$
	=(y, 10)	$E_4(x=10, y=10)$
int z;	*(z)*	$E_5(x=10, y=10, z)$
z = x + y;	*=(R, x)*	$E_6(x=10,y=10, z, R=10)$
	+(R, y)	$E_7(x=10,y=10, z, R=30)$
print z;	*=(z, R)*	$E_8(x=10, y=10, z=30)$
	print(z)	$E_9(I) = \{30, I = z\}$

Final state of the program is given the environment for value of identifier, *I = z*.

Syntax Grammar

We have written Context Free Grammar for the syntax. As CFG is simple to write and understand, we think it is a good choice for a simple programming language such as SPL.

All the nonterminals start with an uppercase letter, terminals have lowercase letters, tokens are written inside single quotes, '|' means or, → means 'defined as'.

Grammar Rules

1)	*Program*	→	*Statement-List*
2)	*Statement-List*	→	*Statement ';' Statement-List* *| Statement ';'*
3)	*Statement*	→	*Assignment* *| Declaration* *| If-Statement* *| While-Statement* *| Print-Statement* *| Comment*
4)	*Assignment*	→	*Identifier '=' Expression* *| Identifier '=' Boolean* *| Identifier '=' Comparison*
5)	*Declaration*	→	*Type Identifier* *| 'int' Identifier '=' Expression* *| 'bool' Identifier '=' Boolean* *| 'bool' Identifier '=' Comparison*
6)	*Comparison*	→	*Expression Comparison-Operator Expression*
7)	*Expression*	→	*Term operator Expression* *| Term*
8)	*If-Statement*	→	*'if' '(' Condition ')' 'then' Block* *| 'if' '('Condition')' then' Block 'else' Block*
9)	*While-statement*	→	*'while' '(' Condition ')' Block*
10)	*Print-Statement*	→	*'print' Expression* *|'print' Comparison* *|'print' 'nl'*

11) Type	→	*int* *| bool*
12) Block	→	*'{' Statement-List '}'*
13) Condition	→	*Comparison* *| Boolean* *| Terminal*
14) Comparison-Operator	→	*'=='* *| '>='* *| '<='* *| '<'* *| '>'* *| '!='*
15) Operator	→	*'+'* *| '-'* *| '/'* *| '*'* *| '%'*
16) Term	→	*Identifier* *| Terminal*
17) Identifier	→	*Letter Identifier-Term* *| Letter*
18) Identifier-Term	→	*Letter Identifier-Term* *| Number Identifier-Term* *| Letter* *| Number*
19) Terminal	→	*Terminal Number* *| Number*
20) Number	→	*'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'*
21) Letter	→	*'a' | 'b' | 'c' |...| 'z' | 'A' | 'B' | 'C' |...| 'Z'*
22) Boolean	→	*'true' | 'false'*

Description of the above mentioned grammar rules:

1) This rule states that the program in SPL is defined as a statement list.
2) This rule states that the statement list is nothing but a list of statements, where in each valid statement ends with a ;

3) The statements in SPL can be assignment statements, declaration statements where in variables can be declared, if-statement, while statement, print statement and comments.
4) This rule states that the Assignment statement can only contain a variable on the left-hand side of the '=', and can contain either expressions or variable on the right-hand.
5) In SPL, we can declare variables of int or bool types. The data type followed by the variable name forms a declaration statement. Assigning a default value to a variable during declaration is also allowed in SPL.
6) In SPL, we can compare a variable with another variable, a variable with an expression and also an expression with an expression. The result of such comparison is a boolean.
7) An expression in SPL contains a set of operators that are applied on values or variables resulting in an integer value. All the expressions in SPL can evaluated to a single integer value. This is a recursive method.
8) This rule states that the block of code in the 'then' will only be executed if the condition in 'if' evaluates to true, otherwise the block of code in the 'else' part is executed (assuming the else part has been specified). In SPL else part is optional.
9) The while loop in SPL keeps executing the block of code inside it as long as the condition holds true. The control exits when the conditions becomes false.
10) We can print values using the print statement in SPL. An expression can also be passed to the print, which then prints the resultant value of the expression.
11) This rule states that SPL supports only int and bool data types.
12) A Block in SPL starts with an '{', consists of a list of statements and then ends with a '}'. These blocks are used in if else statements and while loops in SPL.
13) The condition in SPL always results in a boolean value. These conditions are used in the if else statements and while loops.
14) SPL supports many comparison operators that when applied on appropriate inputs, result in a boolean value. These operators are '==' (returns true if the value on LHS is same as RHS), '>=' (returns true if the value on LHS is greater than or equal to the value on RHS), '<=', '<', '>', '!='
15) SPL supports the basic arithmetic operators for addition, subtraction, division, multiplication, and modulus.
16) This rule states that in an expression, the terms can either be identifiers or terminals.
17) The concept of variables or identifiers is supported by SPL. The identifier name has to start with an alphabet (either uppercase or lowercase), and can contain a series of alphabets or integers.
18) This Identifier-Term is a recursive method that would generate identifiers in such a way that the identifier only starts with a letter, contains numbers in between, and ends with either a letter or a number.
19) Terminal is a recursive method that generates numbers that contain either single digits or that contain two or more digits.
20) Numbers range from 0 to 9. Negative numbers are not valid in SPL.
21) In SPL, letters can be either lowercase or uppercase English alphabet.
22) Boolean can either hold 'true' or 'false'.

Semantics

Program

As can be seen from the grammar, the program consists of a list of statements. A statement ends with a semicolon. The semantics of the program is given by the final state of the environment.

Types

An Identifier is a variable in the program which is mutable. An Identifier can be introduced in 2 ways in the program:

- By Type declaration where the Identifier's type is declared explicitly. Examples are:
  - *int x;*
  - *bool b;*
- By Assignment to either integer, boolean or an identifier. Examples are:
  - *int x = 10;* (only positive integers are allowed).
  - *x = y;*
  - *bool z = true;*

All declarations of ID's are static but can be assigned a value (terminal) later.

Conditional Statement

A condition is an expression which evaluates to boolean value or a boolean itself. Few examples are:

- *x > y* (Here, x and y are Identifier's previously initialized).
- *x != 9*
- *3 == 5*
- *x + 8 != y * 1 + 2*
- *true*

An if-statement has the following (standard) semantics:

1. The condition is evaluated.
2. If the condition evaluates to true, then the body of the if-statement is executed(which is in {}), then the next statement following the if-statement is executed.
3. If the condition evaluates to false, then the else statement following the if-statement is executed.
4. Nested if-else loops are also supported.

The syntax for 'if-statement' is given below:

If (condition) then { statement 1; statement 2;}
[else { statement 3; statement 4;}];

Loops

In SPL, we are implementing the while loop. The block of statements is executed as long as the condition in while loop evaluates to true. The block begins with a '{' and ends with '};'. The syntax for 'while' is shown below:

while (condition) { block };

Our grammar supports nested while loops.

Comments

Comments in SPL start with '#' symbol. Anything written after '#' is ignored till a new line character is encountered. For example:

- *# This is a comment.*
- *# 1001, true…#&*(>, is also a comment.*

Print

In order to print something, the 'print' statement is used. For example:

1. *x=10;*
   print x;
   ***output**: 10*

2. *bool b = false;*
   print b;
   ***output**: false*

3. *print 3 + 5*
   ***output**: 8*

A print statement can print values that are given or stored and can also print a single value after performing evaluating the expression.

Error Handling

Any statement that is not supported by the grammar will result in the following error output:

ERROR IN PROGRAM.

For example, the below program is not supported by the SPL:

program:

int x = 10;
print x;

output: ERROR IN PROGRAM.

Example Program:

bool b;

```
int x;
int z = 10;
x = 10;
x = 5 + 10; #Computes the Right hand part and assigns it to x
b = x < 20;
if ( b )
then { print x; print b; }
else { print 0; };
while ( z >= 0 )
{
        z = z - 1;
        print z;
};
```

Output: 15true9876543210-1