REFERENCE MANUAL

SER 502: LANGUAGES AND PROGRAMMING PARADIGMS

TEAM 30

PROFESSOR: Dr. AJAY BANSAL

Slice Programming Language

TEAM MEMBERS:

1.Vaishak Vellore

2. Darshan Prakash

3. Avinash Mathad Vijayakumar

4.Sanay Devi

Table of Contents

1. Introduction

The design goal of Slice is to create a new language which is easy to learn and implement. Just how everyone loves digging into a slice of pizza, we believe all the users of Slice would love digging into the easy user-friendly functionalities of Slice.

The first step while designing any programming language is the grammar. Initially, we wrote down the grammar in BNF form and created parse trees by hand to see whether our syntactic structure is correct or not. After numerous trial and error methods we finally derived a grammar which was correct and sound. We then incorporated ANTLR into Intellij Idea and wrote the grammar in a file which had a g4 format. After fixing a few errors, we finally came up with the grammar, the screenshots for the parse tree derived using ANTLR are attached in the further sections.

Slice is a high-level language which is compiled, interpreted and creates an intermediate code. The intermediate code will be read line by line and executed in a run-time environment.

2. Compiling and Running Slice Programs

Slice source file should have a .sl extension and the Intermediate file will have a. sauce as extension

3. **Operators of Slice**

Operators	Description
“ + ”	Addition
“ - ”	Subtraction
“ * ”	Multiplication
“ / “	Division
“ % “	Modulus
“ and ”	AND
“ or ”	OR
“ = “	Assignment
“ == “	Comparison
“ != “	Not Equal To
“ >= “	Greater Than Equal To
“ <= “	Lesser Than Equal To
“ > “	Greater Than
“ < “	Lesser Than
T	True
F	False
“ { “	Left Parenthesis
“ } “	Right Parenthesis
“ </ “	Begin Block
“ /> “	End Block

Keywords

Keywords	**Actual meaning**
Num	Integer
Bool	Boolean
takein	Input
giveout	Output
if	if
else	else
stack	Stack
.push	Push
.pop	Pop
while	While Loop

4. Grammar

```
program : '</'block '/>';
block:(assignment | condition | loop | noreturnOp | stackDec | stackOp )* ;

// call
noreturnOp : 'giveout' (datatype | stackOp );
input : 'takein';
datatype : (Num | Bool | Id);

// stack specs
stackDec : 'stack' Id;
stackOp : Id stackfunc ;
stackfunc : (push | pop | empty);
push : '.push' '{' (datatype | boolExpr | expr) '}';
pop : '.pop' '{' '}';
empty : '.isEmpty' '{' '}';

//if else statement
condition : (ifpart) (elsepart)?;
ifpart : 'if' '{' conditionCheck '}' '</' block '/>';
elsepart : 'else' '</' block '/>';

//Loop
loop : 'while' '{' conditionCheck '}' '</' block '/>' ;



//Comparison functions
```

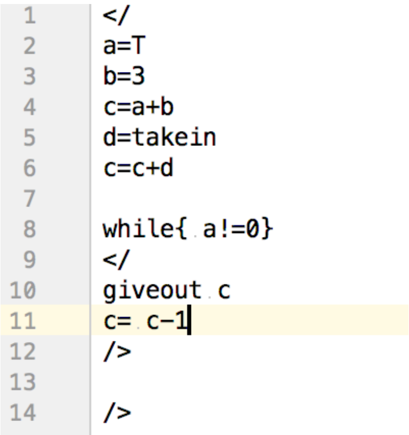
```
conditionCheck : ( boolCompare | integerCompare | stackOp CompareInt Num) ;
integerCompare : expr CompareInt expr ;
boolCompare : boolExpr CompareBool boolExpr;

//Assignment
assignment : Id '=' (input | expr | boolExpr | stackOp) ;

//Integer expression
expr : (term)(subExpr)* ;
subExpr : AdditionOp term ;
term : (factor) (subTerm)*;
subTerm : MultiplicationOp factor;
factor : (Id | Num | '{' expr '}');

// Boolean expression
boolExpr : (boolTerm) (boolSubExpression)*;
boolSubExpression : BooleanOR boolTerm ;
boolTerm : (boolFactor) (subBoolTerm)*;
subBoolTerm : BooleanAnd boolFactor;
boolFactor : (Id | Bool | '(' boolExpr ')');

//Operators
MultiplicationOp : ('*' | '/' | '%');
AdditionOp : ('+' | '-') ;
CompareInt : ('>' | '<' | '==' | '<=' | '>='| '!=');
BooleanAnd : 'and' ;
BooleanOR : 'or' ;
CompareBool : 'is' ;
```

```
//Types

//boolean

Bool : ('T' | 'F') ;


//integer

Num : [0-9]+ ;

//valid identifiers (letters of either case and numbers)

Id: ('a'..'z' | 'A'..'Z') ('a'..'z' | 'A'..'Z' | '0'..'9')*;


//whitespace

Emptyspace: [ \t\r\n]+ -> skip;


//comments

Comments : '//' ~( '\r' | '\n' )* -> skip;
```

5. Lexical Analysis

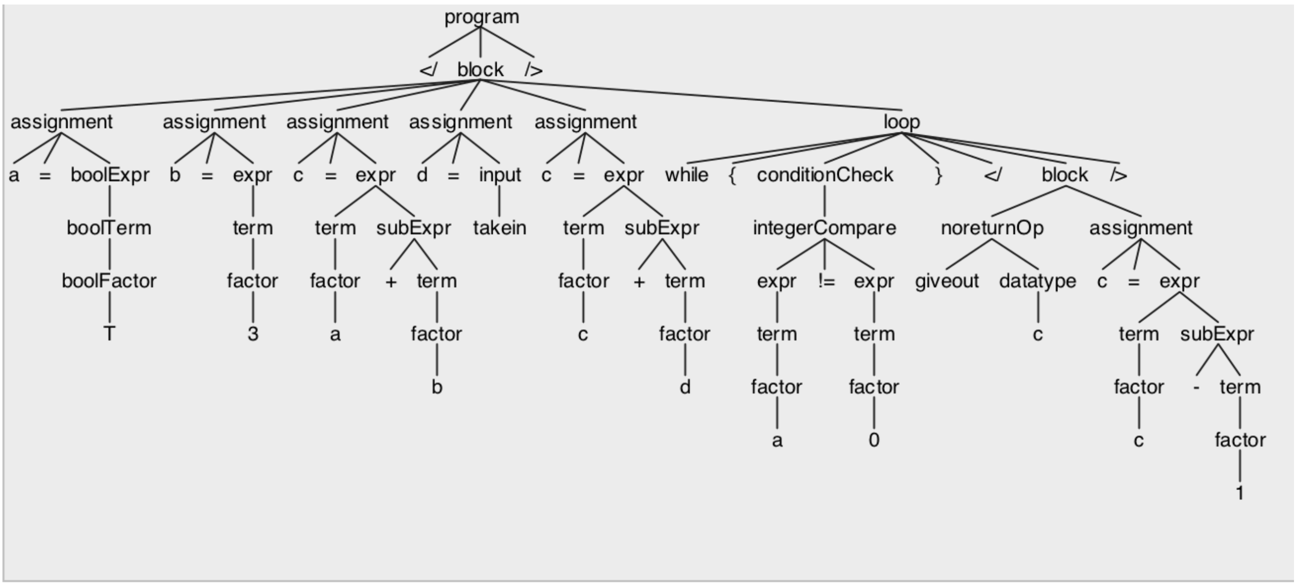
This is the first phase of compilation. In this we convert a sequence of characters into a sequence of tokens. Consider the following sample examples:

Sample program 1: Showing while looping constructs.

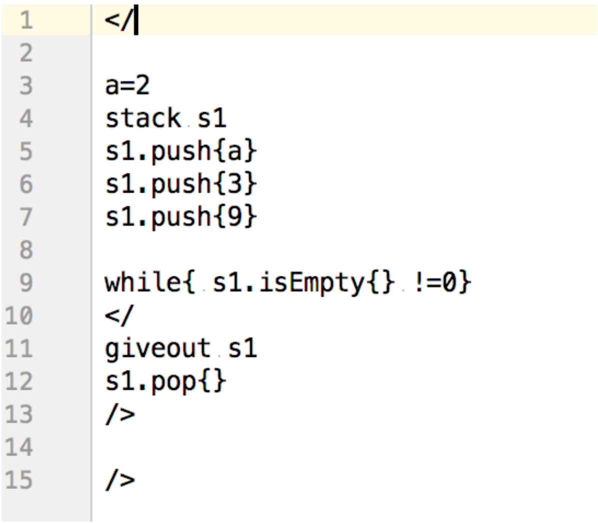


```
</
a=T
b=3
c=a+b
d=takein
c=c+d

while{ a!=0}
</
giveout c
c= c-1
/>

/>
```

Parse Tree Generated Using ANTLR:

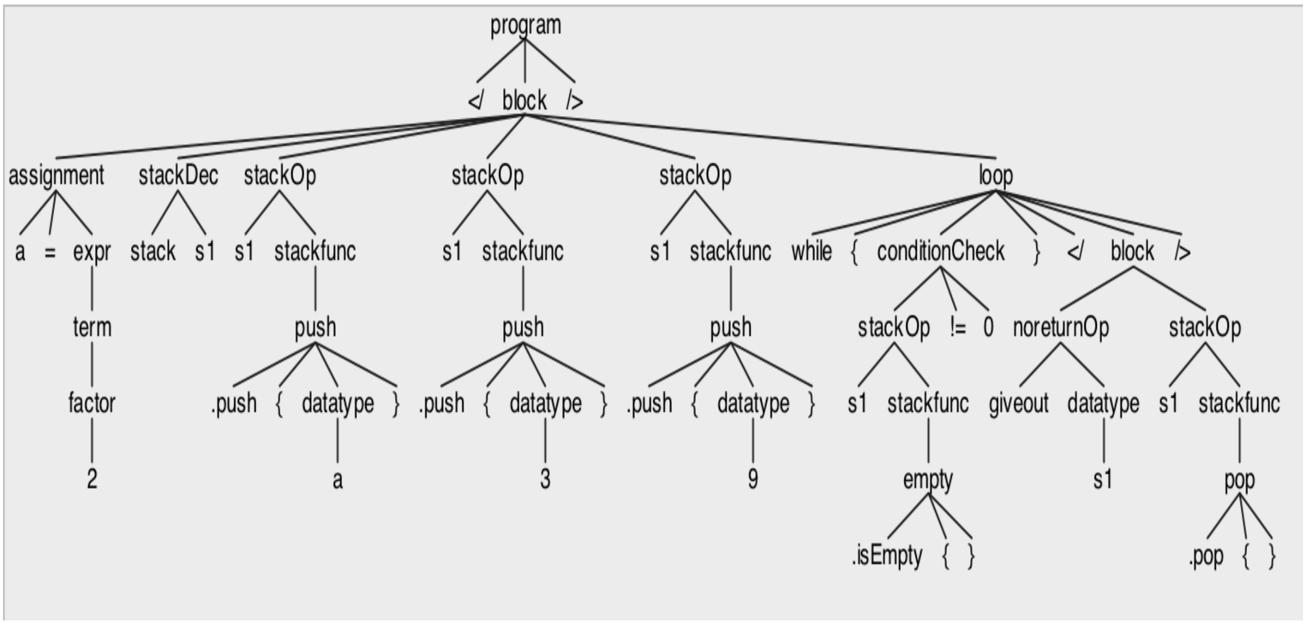


Sample program 2: Showing stack operations.

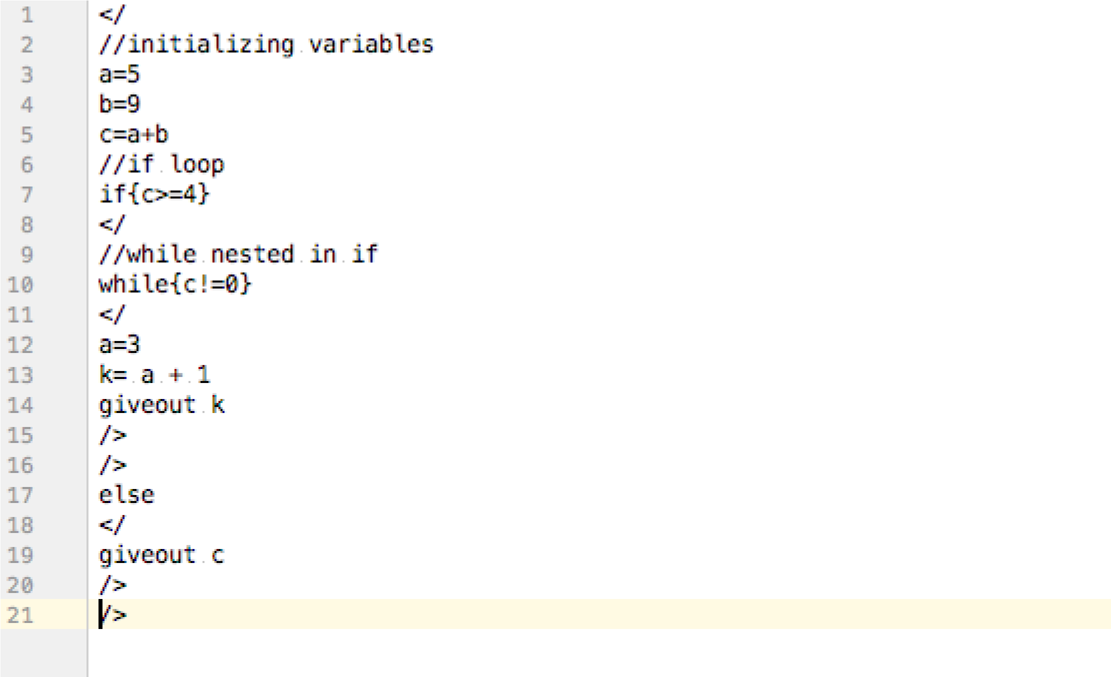


```
</

a=2
stack s1
s1.push{a}
s1.push{3}
s1.push{9}

while{ s1.isEmpty{} !=0}
</
giveout s1
s1.pop{}
/>

/>
```

Parse Tree Generated Using ANTLR:

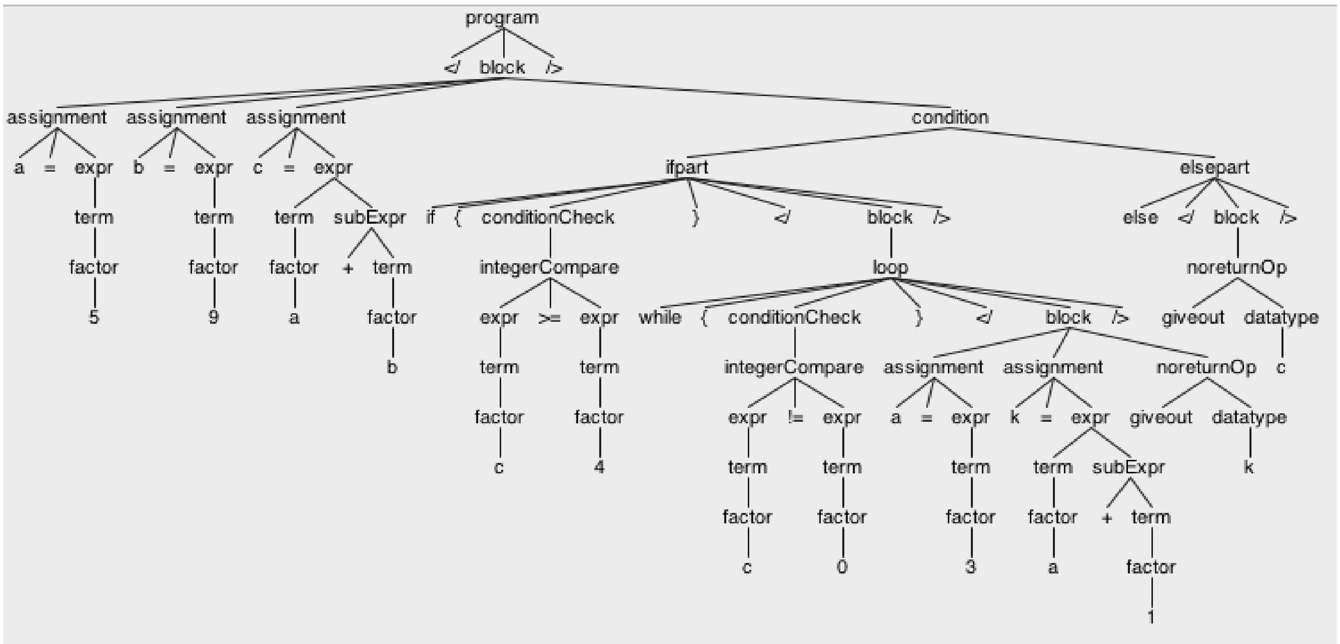


Sample program 3: Showing if and while looping constructs.



```
</
//initializing variables
a=5
b=9
c=a+b
//if loop
if{c>=4}
</
//while nested in if
while{c!=0}
</
a=3
k= a + 1
giveout k
/>
/>
else
</
giveout c
/>
/>
```

Parse Tree Generated Using ANTLR:


program
</ block />
assignment assignment assignment condition
a = expr b = expr c = expr ifpart elsepart
term term term subExpr if { conditionCheck } </ block /> else </ block />
factor factor factor + term integerCompare loop noreturnOp
5 9 a factor expr >= expr while { conditionCheck } </ block /> giveout datatype
b term term integerCompare assignment assignment noreturnOp c
factor factor expr != expr a = expr k = expr giveout datatype
c 4 term term term term subExpr k
factor factor factor factor + term
c 0 3 a factor
1

6. Interpreter

We intend to use python language for developing an interpreter.

7. Parsing

We would use LR parsers for the top down approach. We would eliminate the left recursion in the grammar for this purpose.