Project: Compiler and Virtual Machine for a Programming Language
SER 502: Languages and Programming Paradigms
Submission Date:April 14th, 2017

Milestone 2: Name, Design and Grammar: *Your team should submit a PDF document that contains the name, design and the grammar of the language driving the implementation of the compiler and byte-code interpreter/runtime environment. Include information about the interpreter. Discuss the parsing technique you will employ. Also discuss any data structures used by the parser and interpreter. Your team must also submit the contribution.txt from your repository's doc folder detailing the individual contribution of each team member until now and the contribution plan for the final milestone.*

Grammar:

This design document is to explain the grammar and parser used for a new simple language called Infinity. This language is designed to illustrate the semantic features of many high level languages.

An EBNF grammar for Infinity is shown below:

```
<program> ::= {<statement>,<space>};

<statement> ::= <assignment>|<condition>|<loop>|<variable>;

<assignment>                                                              ::=
<identifier>,"=",(<expression>|"'",<character>,"'"|'"',<character>,'"'|"'"
,<string>,"'"|'"',<string>,'"'|<boolean>|<constant>),";";


<condition>                                                               ::=
"if","(",<equality>,")","{",<statement>,{<space>,<statement>},"}",{"elif",
"(",<equality>,")","{",<statement>,{<space>,<statement>},"}"},
["else","{",<statement>,{<space>,<statement>},"};

<equality> ::= <expression>,(">"|"<"|"<="|">="|"=="|"!="),<expression>;

<loop>                                                                    ::=
"while","(",<equality>,")","{",<statement>,{<space>,<statement>},"}";

<expression> ::= <expression>,"+",<term>|<expression>,"-",<term>|<term>;
<term> ::= <term>,"*",<factor>|<factor>;
<factor> ::= <factor>,"/",<factor1>|<factor1>;
```

```
<factor1> ::= “(”,<expression>,”)”|<number>;

<boolean> ::= <term1>,{”||”,<term1>}|<term1>;
<term1> ::= <factor1>,{”&&”,<factor1>}|<factor1>;
<factor1> ::= <constant>|”!”,<factor1>;
<constant> ::= “true”|“false”;


<character> ::= <letter>|<digit>|<symbol>;

<identifier> ::= <letter>,{<letter>|<digit>|<symbol>};

<number> ::= [-],<digit>,{<digit>};

<string> ::= <character>,{<character>};


<letter>                                                                ::=
“A”|”B”|”C”|”D”|”E”|”F”|”G”|"H"|"I"|"J"|"K"|"L"|"M"|"N"|"O"|"P"|"Q"|"R"|"S
"|"T"|"U"|"V"|"W"|“X”|”Y”|”Z”|”a”|”b”|”c”|”d”|”e”|”f”|”g”|”h”|”i”|”j”|”k”|
”l”|”m”|”n”|”o”|”p”|”q”|”r”|”s”|”t”|”u”|”v”|”w”|”x”|”y”|”z”;

<digit> ::= "0"|"1"|”2”|”3”|“4”|”5”|”6”|”7”|”8”|”9”;

<symbol>                                                                ::=
"["|"]"|"{"|"}"|"("|")"|"<"|">"|"'"|'"'|"="|"|"|"."|","|";"|"_";

<space> ::= ? US-ASCII character 32 ?;


<variable> ::= “int”,<space>,<identifier>,”=”,<number>,”;”;
```

The above mentioned grammar includes kinds of declarations (variables, type), statements (assignment, loops, conditional) and expressions. The top level entry point of Infinity is program, which is repetitive of statements. These statements can be assignments, conditional, loop, assignment and print and expression itself. The assignment operator assigns an expression, character, string or boolean values to an identifier. The conditional construct (if-else) evaluates a condition to true or false and then execute the expression depending on the expression evaluated.

Examples showing how to use Infinity:
This code snippet shows a small Infinity program to swap two integers which includes three declarations, assignments, type declarations and expressions

(2.1) Sample program 1

```
int(a) = 10;
int(b) = 5;
int(temp) = 0;
temp = a;
a = b;
b = temp;
```

The following code snippet shows using Infinity to solve complex statements:

(2.2) Sample Program 2

```
int(a)=10;
int(b)=15;
while(b>a)
{
a=a+1;
}
if(a==15)
{
b=9;
}
else
{
b=10;
}
```

Antlr for Lexical analysis and Parse tree generation:

ANTLR is a tool that translates the grammar given to a lexer and the runtime needed by the generated parser. ANTLR uses LL(K) recursive-descent parsers. Recursive descent parsers, turns the nonterminals into a group of mutually recursive procedures whose actions are based on the right-hand sides of the BNFs. The greatest advantage of using LL(k) is that terms can be inserted anywhere within the grammar. LL(k) is easy to understand because it is essentially a direct translation of the EBNF grammar.

Because ANTLR has same recognition mechanism for lexing and parsing ANTLR-generated lexers are much stronger than DFA-based lexers such as those generated by DLG and lex. ANTLR generates predicated- LL(k) lexers because of which you can have syntactic and semantic predicates.

The other advantages of using ANTLR are:

- You can actually read and debug the output as its very similar to what you would build by hand.
- The syntax for specifying lexical structure is the same for lexers, parsers, and tree parsers.
- You can have actions executed during the recognition of a single token.
- You can recognize complicated tokens such as HTML tags or "executable" comments like the javadoc `@`-tags inside `/** ... */` comments. The lexer has a stack, unlike a DFA, so you can match nested structures such as nested comments.

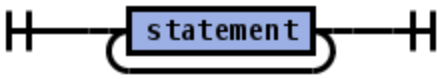
The following is the grammar in Antlr compatible format:

```
grammar infinity
        ;

program: statement+
         ;

statement: (assignment|loop|condition|variable) Newline
           ;

assignment: identifier Assign (expression | Number | Letter | character) ';'
           ;

equality: (expression|identifier) (GT | LT | LE | EQ | NE | GE) expression
          ;

loop: 'while' '(' equality ')' Newline '{' Newline statement+ '}'
      ;

variable: 'int' '(' identifier ')' (Assign (expression | Number | Letter |
character))? ';'
          ;

condition: 'if' '(' equality ')' Newline '{' Newline statement+ '}' (Newline
'else' Newline '{' Newline statement+ '}')?
           ;

expression: expression (MUL | DIV | MOD)
           expression
           | expression (ADD | SUB)
           expression
           |NOT expression
           |expression AND expression
           |expression OR expression
           | Number
           | identifier
```

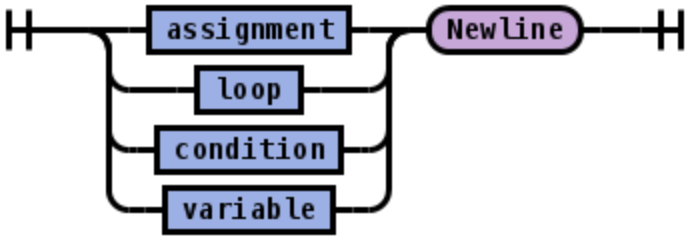
```
            |Constant
            | '(' (expression|identifier) ')'
            ;


identifier: (Letter|Number)+
             ;

character: (Letter|Number)
           ;

Letter: [a-zA-Z]+
        ;

Number: [0-9]+
        ;

Newline: '\r'? '\n'
         ;

Constant: 'True' | 'False'
          ;

Assign: '='
        ;
GT: '>'
    ;
LT: '<'
    ;
EQ: '=='
    ;
NE: '!='
    ;
GE: '>='
    ;
LE: '<='
    ;
MUL: '*'
     ;
ADD: '+'
     ;
SUB: '-'
     ;
DIV: '/'
     ;
NOT: '!'
     ;
AND: '&&'
     ;
OR: '||'
    ;
MOD: '%'
     ;
```

Syntax or railroad diagrams are a way to represent a context-free grammar. They represent a graphical alternative to EBNF. The following the syntax diagrams for Infinity:

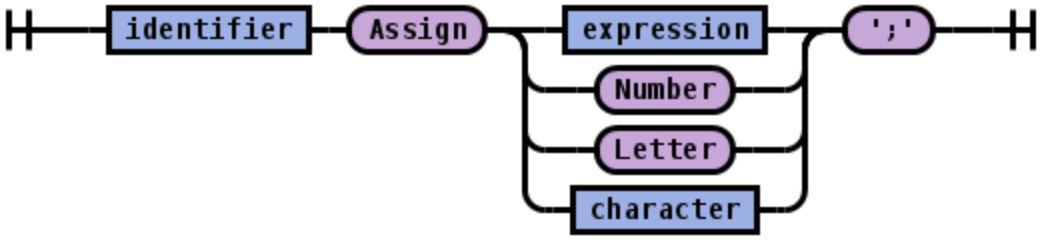
Program:



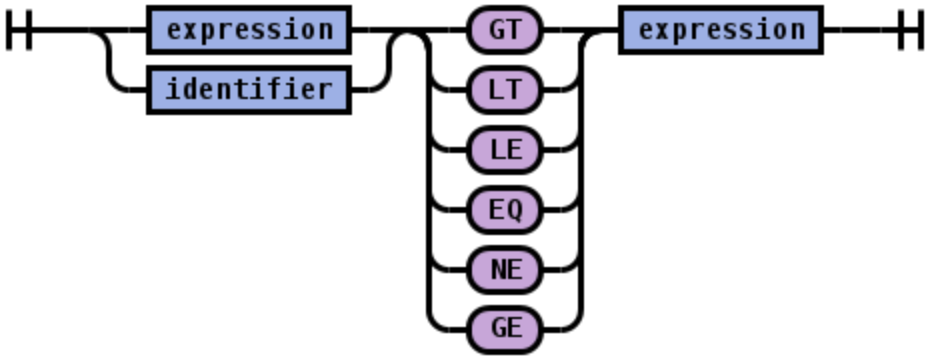
Statement:



Assignment:

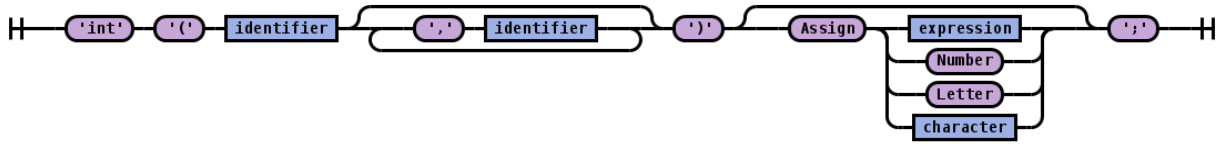


Equality:

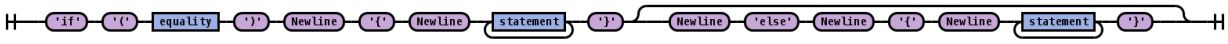


Loop:

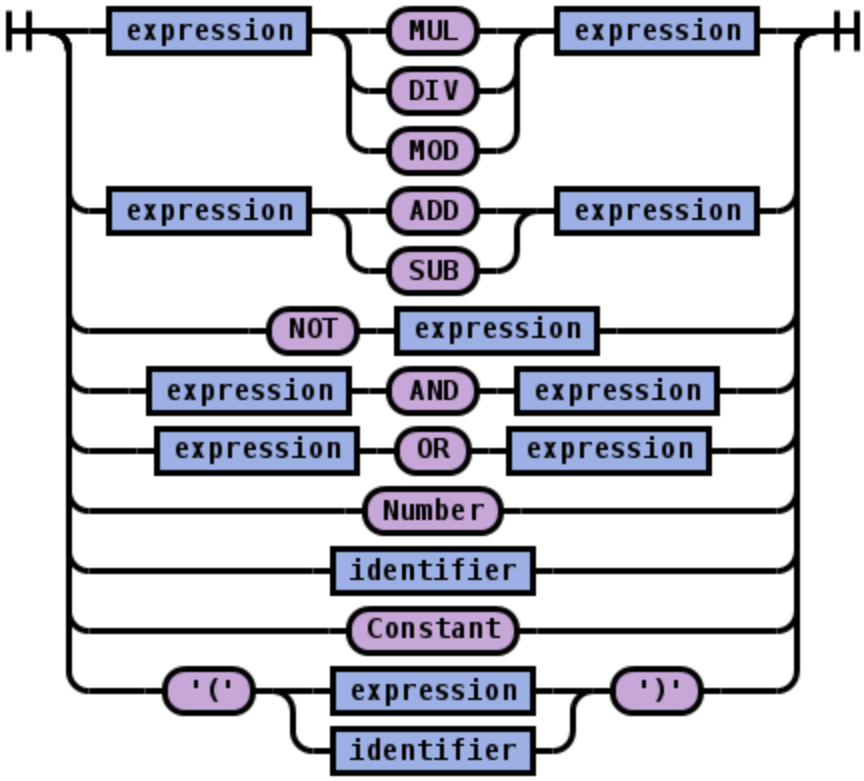
Variable:



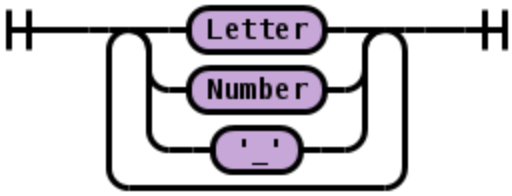
Condition:



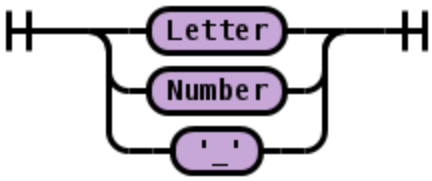
Expression:



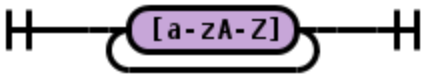
Identifier:



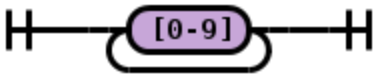
Character:



Letter:

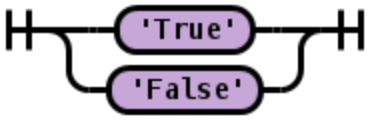


Number:



NewLine:

Constant:

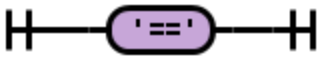


Assign:

GT:

LT:

EQ:



NE:

GE:

LE:

MUL:

ADD:

SUB:

DIV:

NOT:

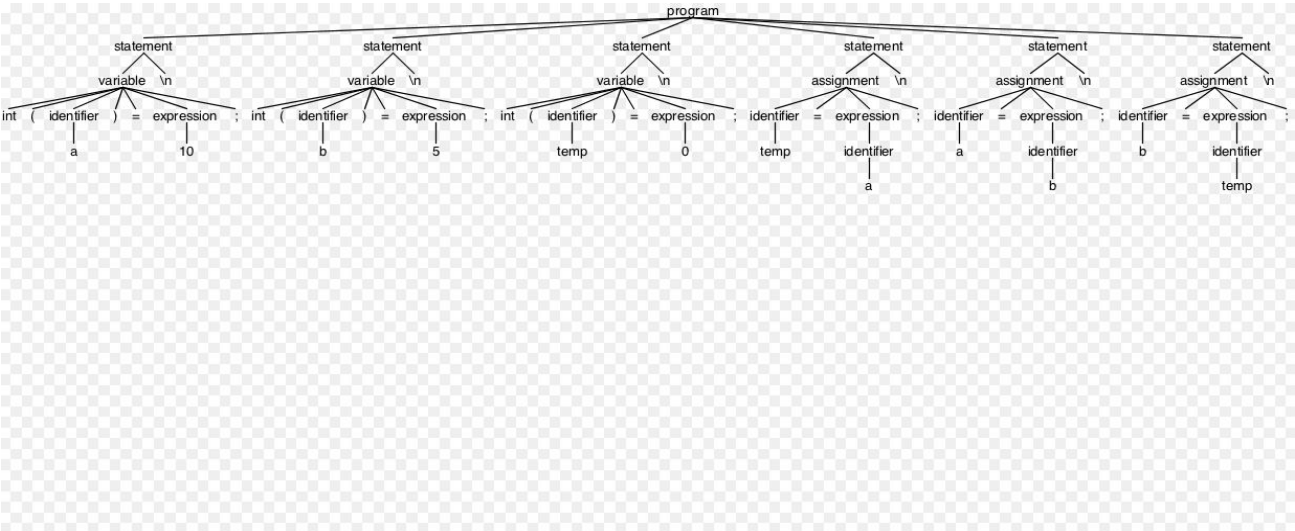
AND:

OR:

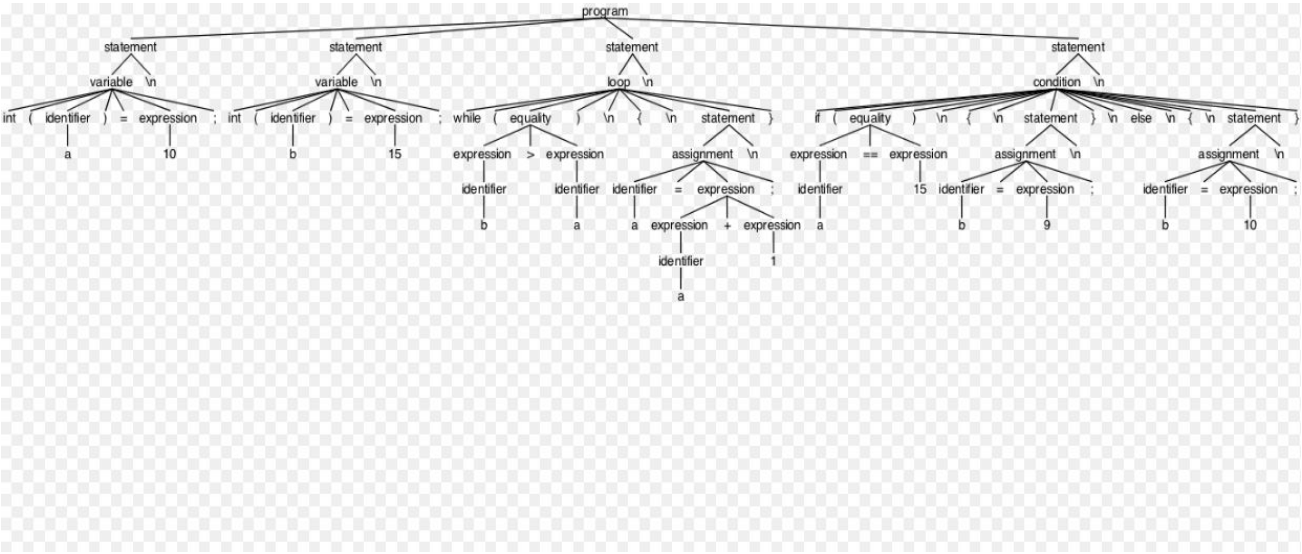
MOD:

WS:

A parse tree is an ordered, rooted tree that represents the syntactic structure of program according to given context-free grammar. Following are the parse trees for the sample programs (2.1 & 2.2) mentioned above



(2.3) Parse Tree for sample Program1



(2.4) Parse Tree for sample program 2

Future work

In further iteration of the project we need to use the syntax tree generated by the semantic Analyzer (Antlr) to generate an intermediate code. This intermediate code is supplied to the run time environment which would be implemented tentatively in Java and Symbol table to store the environment.