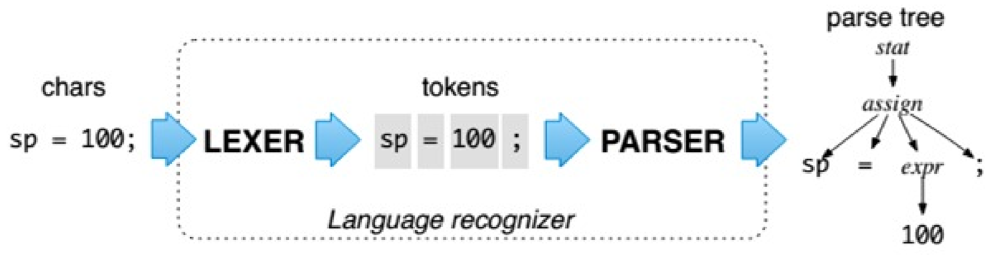
Abstract Syntax Tree for Shell Scripting

Akshara Neeruganti-IMT2016022
Soumya Kariveda-IMT2016053
Lakshmi Manonmaie-IMT2016057

1 Introduction

Lexer:
A lexer is a software program that performs lexical analysis. Lexical analysis is the process of separating a stream of characters into different words, which in computer science we call ”tokens”.
Parser:
A parser goes one level further than the lexer and takes the tokens produced by the lexer and tries to determine if proper statements have been formed i.e., A parser defines rules of grammar for these tokens and determines whether these statements are semantically correct



2 Syntax Tree

There are two kinds of Syntax trees, **Concrete Syntax Tree** and **Abstract Syntax Tree**.

2.1 Concrete Syntax Tree(CST)

These are traditionally designated parse trees, which are typically built by a parser during the source code translation and compiling process.

2.2 Abstract syntax tree(AST)

This is also referred as just syntax tree. It is a tree representation of the abstract syntactic structure of source code written in a programming language.The syntax is ”abstract” in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural, content-related details.

2.3 Advantages of AST:

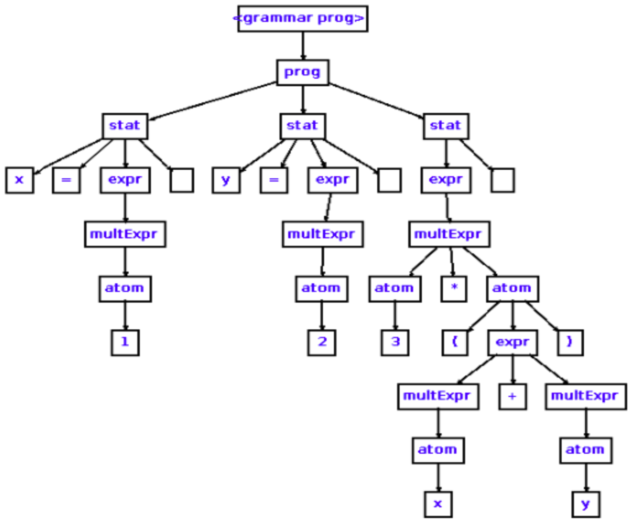
- Easy to visualize the flow of source code
- Since logic is separated from implementation it is easy to understand and thus debug

Input

```
x=1
y=2
3*(x+y)
```

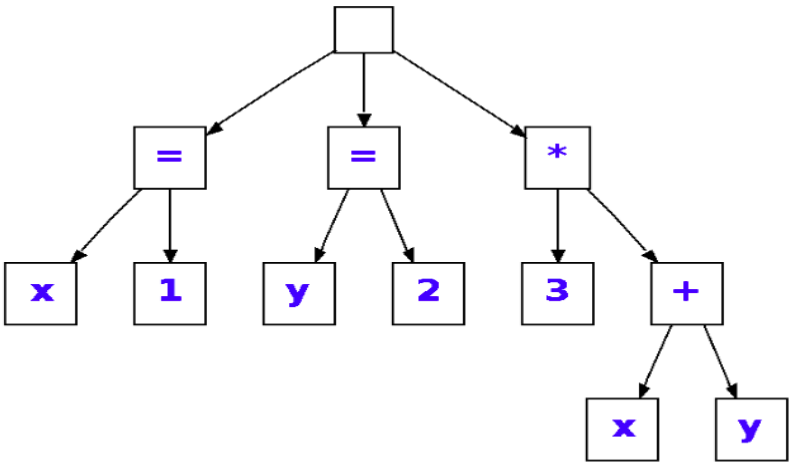
Parse Tree

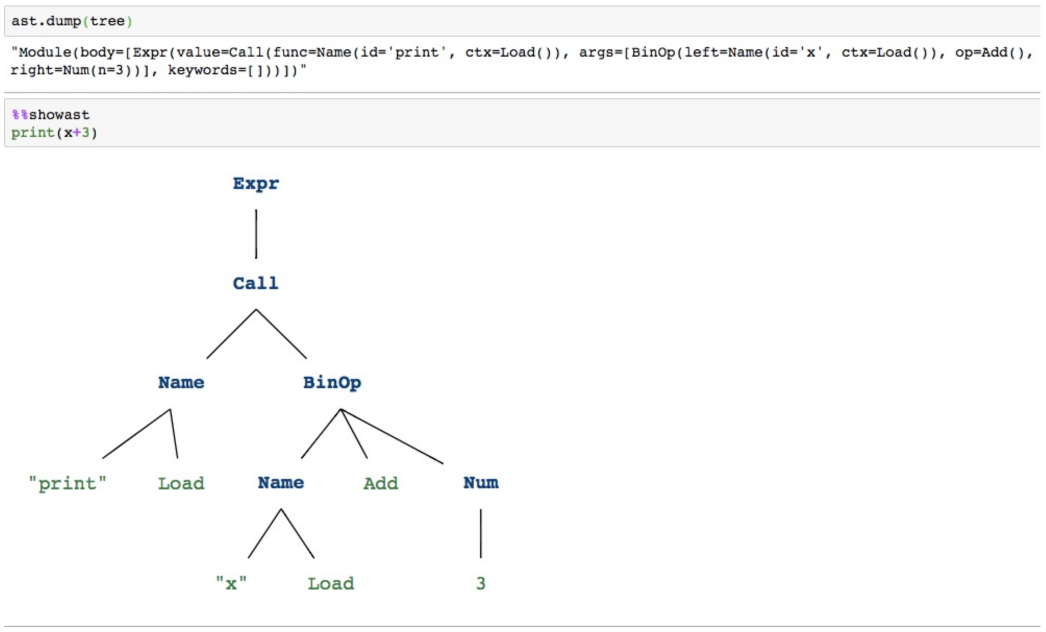
The parse tree is a concrete representation of the input. The parse tree retains all of the information of the input. The empty boxes represent whitespace, i.e. end of line.



AST

The AST is an abstract representation of the input. Notice that parens are not present in the AST because the associations are derivable from the tree structure.





```
ast.dump(tree)
```

```
"Module(body=[Expr(value=Call(func=Name(id='print', ctx=Load()), args=[BinOp(left=Name(id='x', ctx=Load()), op=Add(),
right=Num(n=3))], keywords=[]))])"
```

```
%%showast
print(x+3)
```

3 AST Construction

3.1 Description

We built an AST for all basic constructs of shell scripting source code (which includes while, if, if-else, echo and mathematical expressions) using OCaml. We are taking input as list of tokens assuming lexers job is already done.
The main aim of our project is to parse these given tokens and give it's corresponding AST.

3.2 Implementation

3.2.1 Input Format

The input will be a list of type **generic_obj** which include

- Echo_generic - for **echo** keyword
- If_generic - for **if** keyword
- While_generic - for **while** keyword
- Tokens_generic - for list of type tokens
  Tokens will include
  - Op - for operators (+, -, x, /)
  - Num_tok - for constant values
  - Var_tok - for variables
- Expr_generic - for computed set of tokens
- Cond - for condition in if and while
- Others - for tokens that are to be ignored in AST ($, {, }, :)

3.2.2 Approach

Construction of AST:

- **for expressions:**
  Parsing of expressions is done is 2 parts
  - Infix to Prefix notation:
    The operator in infix notation is in between the operands, this is commonly used so the source code will use expressions of this type.Where as prefix is notation where the operator is before it's operands, this form will helps to parse better.
    We are recursively converting the given infix expression to prefix, by matching with operator and then shift the operands.

  - Prefix notation to AST:
    This is also constructed recursively, using pattern matching. If token matches with an operator then the next two objects are made as operands of that operator(at same level in the tree).

- **for echo:**
  Parsing of echo is done in the function **parser_echo** which takes 2 arguments, which are the string **"echo"** and **statement** to be printed.
  This will return object of type **Generic_echo**, which will have 2 children, 1st is string "echo" and 2nd is what is to be printed.

- **for If and If-else:**
  Parsing of if block is done in these two functions **parser_if** and **parse_only_if**.

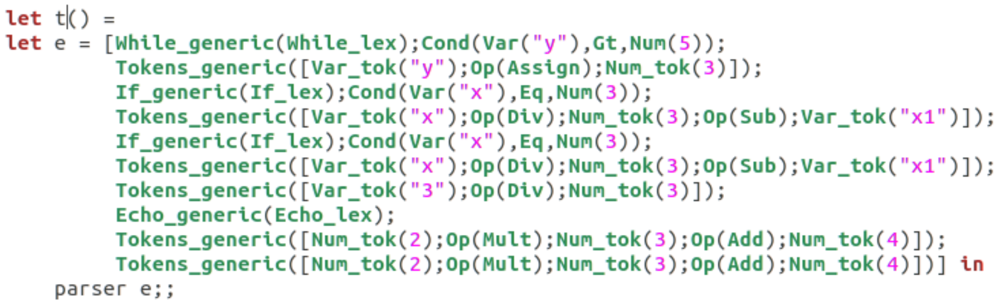
  parser_if is for if-then-else block and takes 3 arguments, which are **condition** for if block, **statements** to be executed if the condition is true and **statements** to be executed if condition is false.
  This will return object of type **Generic_if**, which will have 3 children, 1st is condition, 2nd is body of if block and 3rd is body of else block.

  parser_only_if is for if block without else, which is almost similar to if-then-else, but will not have 3rd argument of parser_if(else part)
  This will return object of type **Generic_only_if**,with only 2 children.

- **for while:**
  Parsing of while block is done in the function **parser_while** which takes 2 arguments, which are **condition** of while block and **statements** to be executed if conditions are met.
  This will return object of type **Generic_while**, which will have 2 children, 1st is condition and 2nd is body of while block.

- **for whole code:**
  While parsing we are matching tokens with constructs of shell script and take required number of next tokens as arguments of the corresponding functions.
  If the token matches with **"if"**, then we are checking for the 3rd next argument, if it is of type expression then we are considering it as **if-then-else**, else consider it as only **"if"**(without else) and take that 3rd next token to be as a new block(while or echo or another if)
  If the tokens match with type Others(to be ignored like $,},{,:) then we are just parsing the rest of the tokens, ignoring that token.

The test case will look like:

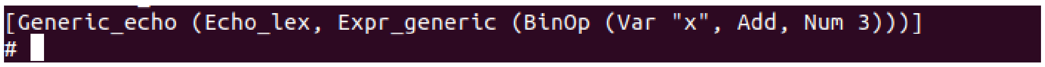


```
let t() =
let e = [While_generic(While_lex);Cond(Var("y"),Gt,Num(5));
         Tokens_generic([Var_tok("y");Op(Assign);Num_tok(3)]);
         If_generic(If_lex);Cond(Var("x"),Eq,Num(3));
         Tokens_generic([Var_tok("x");Op(Div);Num_tok(3);Op(Sub);Var_tok("x1")]);
         If_generic(If_lex);Cond(Var("x"),Eq,Num(3));
         Tokens_generic([Var_tok("x");Op(Div);Num_tok(3);Op(Sub);Var_tok("x1")]);
         Tokens_generic([Var_tok("3");Op(Div);Num_tok(3)]);
         Echo_generic(Echo_lex);
         Tokens_generic([Num_tok(2);Op(Mult);Num_tok(3);Op(Add);Num_tok(4)]);
         Tokens_generic([Num_tok(2);Op(Mult);Num_tok(3);Op(Add);Num_tok(4)])] in
    parser e;;
```

3.2.3 Results

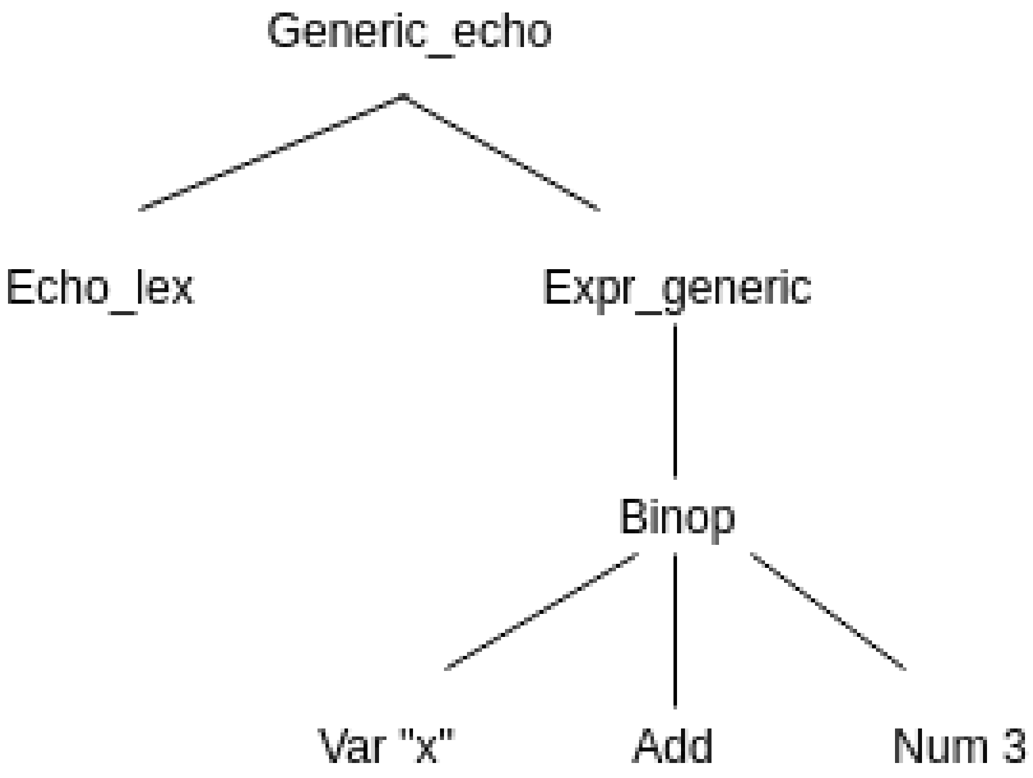
- **echo** $x + 3$

  Our output:



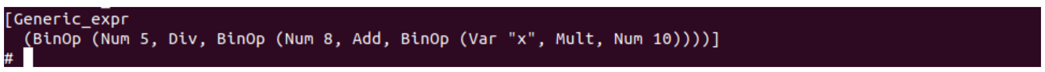
```
[Generic_echo (Echo_lex, Expr_generic (BinOp (Var "x", Add, Num 3)))]
# 
```

  Interpretation:



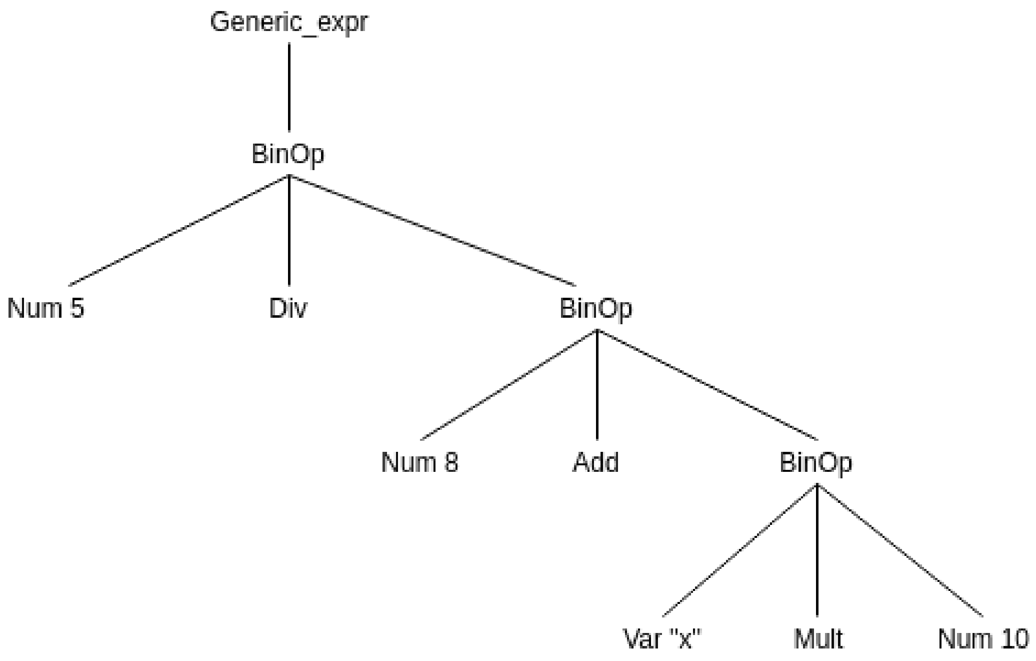
- $5/8 + x * 10$

  Our output:



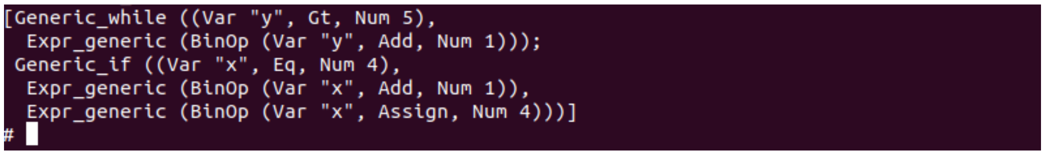
```
[Generic_expr
  (BinOp (Num 5, Div, BinOp (Num 8, Add, BinOp (Var "x", Mult, Num 10))))]
# 
```

  Interpretation:



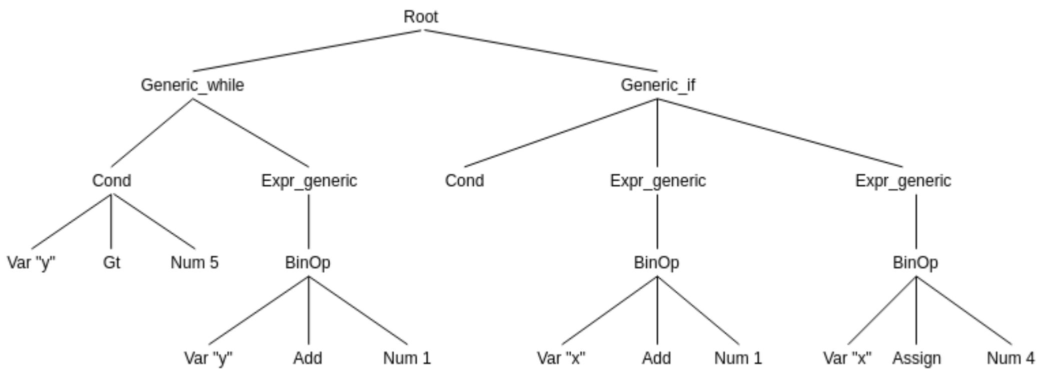
- **while**($y < 5$)**:**
  $y = y + 1$
  if($x = 4$)**:**
  $x + 1$
  else:
  $x = 4$

  Our output:

```
[Generic_while ((Var "y", Gt, Num 5),
  Expr_generic (BinOp (Var "y", Add, Num 1)));
 Generic_if ((Var "x", Eq, Num 4),
  Expr_generic (BinOp (Var "x", Add, Num 1)),
  Expr_generic (BinOp (Var "x", Assign, Num 4)))]
# 
```

  Interpretation:

3.3 Limitations

- Expressions are evaluated without considering BODMAS rules.
- There cannot be more than one expression in if or else or while block

4 Future Work

- To be able to construct AST for source code, lexer should be implemented to give tokens as per our requirement.
- Extend to all constructs of shell scripting by expanding the vocabulary.

5 References

- https://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html.
- https://stackoverflow.com/questions/22737031/this-pattern-matching-is-not-exhaustive-in-ocaml
- https://docs.python.org/3/library/ast.html
- https://github.com/hchasestevens/show_ast
- https://github.com/sujitkc/PL