Assignment No: B6

Roll No: 4127

Title:

Abstract syntax tree generation

Problem Statement:

Generating abstract syntax tree using LEX and YACC.

Learning Objective:

1. Understand use of abstract syntax tree to represent structure of source program.
2. Understand and implement AST data structure which widely used in compilers.

Learning Outcome:

1. Able to understand use of abstract syntax tree to represent structure of source program.
2. Able to understand and implement AST data structure which widely used in compilers.

Software and Hardware Requirments:

1. PC with the configuration as 64-bit Fedora
2. LEX, YACC utility.

Theory:

Abstract syntax tree:

i. An abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code.

ii. The syntax is "abstract" in not representing every detail appearing in the real syntax. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construct like an if-condition-then expression may be denoted by means of a single node with three branches.

iii. This distinguishes abstract syntax trees from concrete syntax trees, traditionally designated parse trees, which are often built by a parser during the source code

translation and compiling process. Once built, additional information is added to the AST by means of subsequent processing.

iv. Abstract syntax trees are also used in program analysis and program transformation systems.

Design:

i. The design of an AST is often closely linked with the design of a compiler and its expected features.

ii. Core requirements include the following:

- Variable types must be preserved, as well as the location of each declaration in source code.
- The order of executable statements must be explicitly represented and well defined.
- Left and right components of binary operations must be stored and correctly identified.
- Identifiers and their assigned values must be stored for assignment statements.

iii. These requirements can be used to design the data structure for the AST.

Some operations will always require two elements, such as the two terms for addition. However, some language constructs require an arbitrarily large number of children, such as argument lists passed to programs from the command shell. As a result, an AST used to represent code written in such a language has to also be flexible enough to allow for quick addition of an unknown quantity of children.

iv. Another major design requirement for an AST is that it should be possible to unparse an AST into source code form.[why?] The source code produced should be sufficiently similar to the original in appearance and identical in execution, upon recompilation.

v. For example, lets consider an ABT for

```
while b ≠ 0
      if a > b
        a := a − b
      else
        b := b − a
return a
```

So the AST will be given as follows:

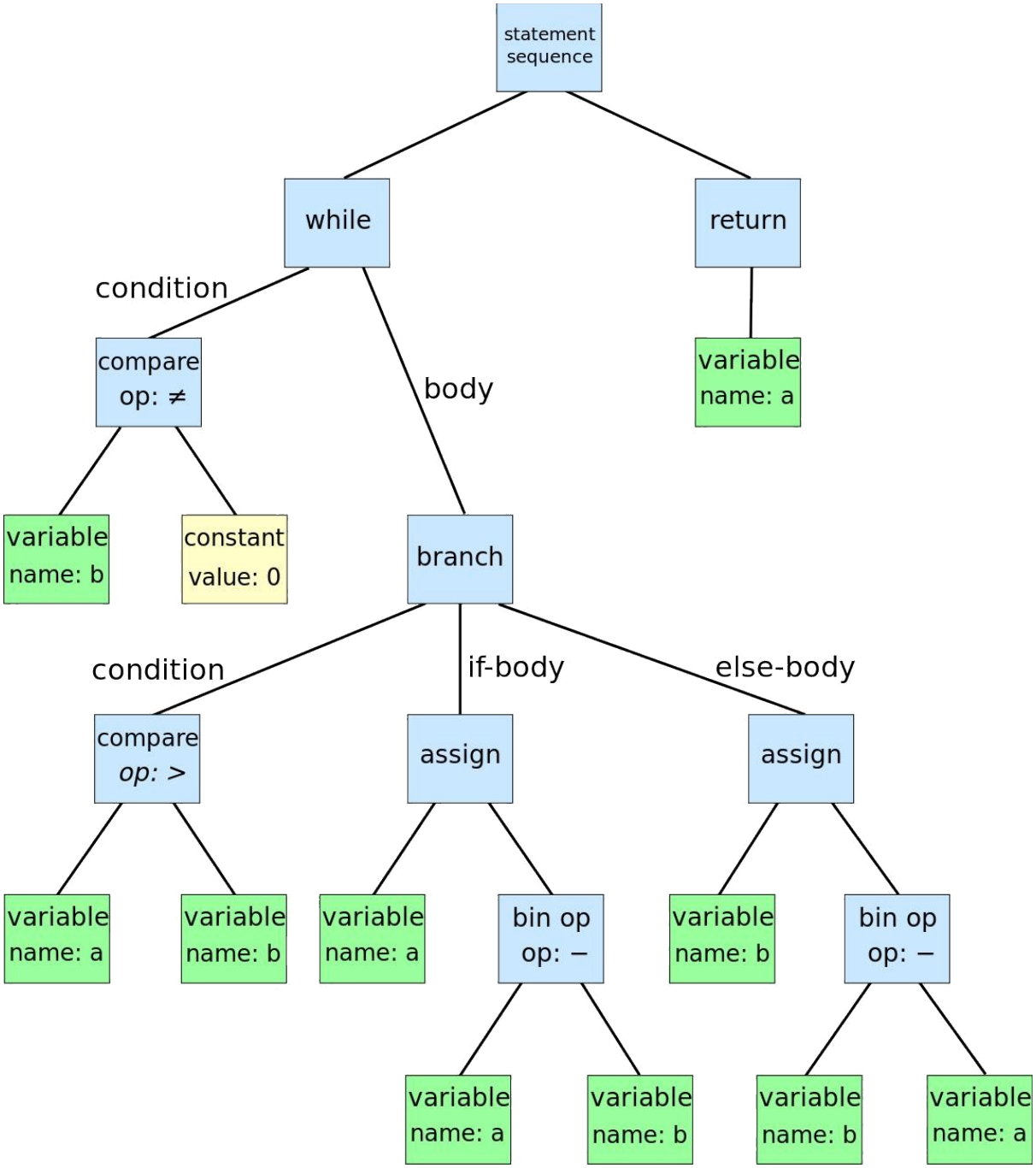


Fig. Abstract Syntax Tree

Mathematical Model :

Let S be the solution for the system.

S= { St , E , X, Y , F, DD , NDD, Sc, Fc}
Where,

St is an initial state such that
St={ ∅ }

E_t is a End state.
E_t = { Y } i.e Symbol table

X= Set of inputs
X = { x1}
x1 -> {[0-9]+[()/*+-]*}
i.e input is arithmetic expression

Y= Set of outputs
Y= {y1}
y1 -> {[0-9]+[()/*+-]*}
i.e. preorder of tree representation of input expression

F=A set of Functions.
F={F1,F2,F3}
where,
F1: function to make nodes in tree
F1(X) -> y'
where, y'= {lp,rp,t} and rp= right pointer, lp=left pointer, t=token
F2: function to print tree in preorder form F2 (y') -> y1

F3: function to define the rules and grammar.
F3(x') -> y1
x' : context free grammer x' -> {[a-z]*,[A-Z]*,
[0-9]*,[special characters]*}

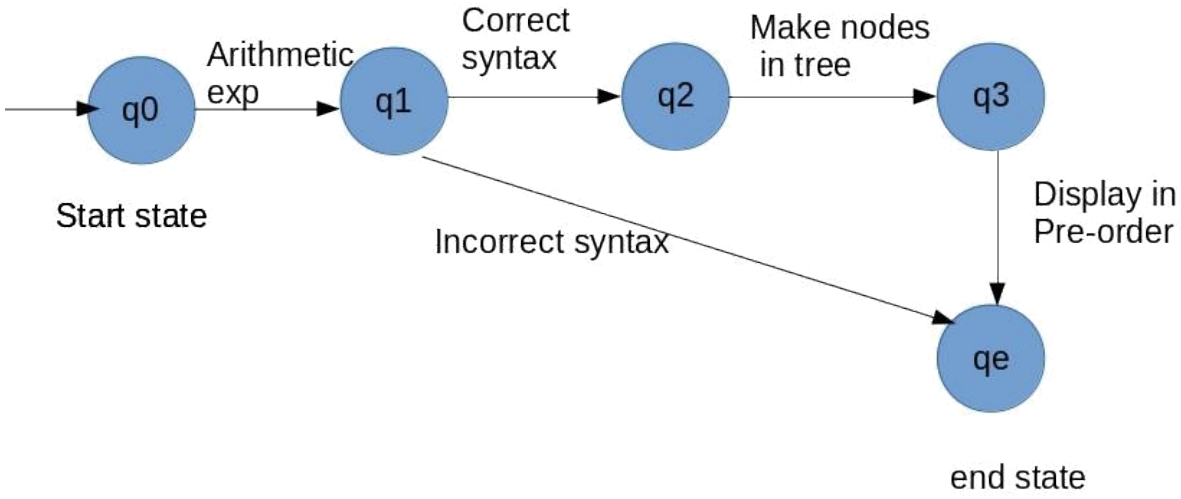
DD=Deterministic Data
DD={x | x is an expression which includes numbers, brackets, arithmetic operators}

NDD= Non-Deterministic Data
NDD={y | y is tree of the expressions of Language}

Sc : Success case
{AST is generated successfully for given expression}

Fc: Failure case
{Expression given is invalid}

State Transition Diagram:



where,
q0: Start state
q1: Accept input expression
q2: Check the syntax using grammer rules
q3: Create nodes for the tree
qe: End state

Conclusion:

Hence we have successfully generated Abstract Syntax Tree for sample language and learned how to implement it.