Moti Begna

CSCI 5106

**Homework 2**

Problem 1:

<A> ::= <K> ‘=’ <A> | <E>

<E> ::= <E> ‘||’ <F> | <F>

<F> ::= <F> ‘&&’ <G> | <G>

<G> ::= <G> ‘<’ <H> | <G> ‘<=’ <H> | <G> ‘>=’ <H> | <G> ‘>’ <H> | <H>

<H> ::= <H> ‘+’ <I> | <H> ‘-’ <I> | <I>

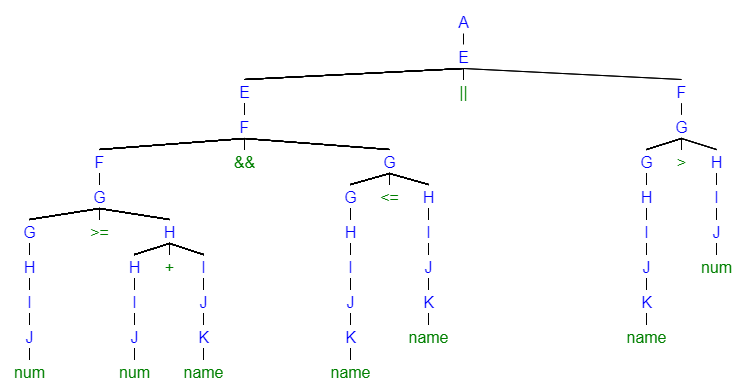
<I> ::= <I> ‘\*’ <J> | <I> ‘/’ <J> | <I> ‘%’ <J> | <J>

<J> ::= num | ‘(‘ <A> ‘)’ | <K>

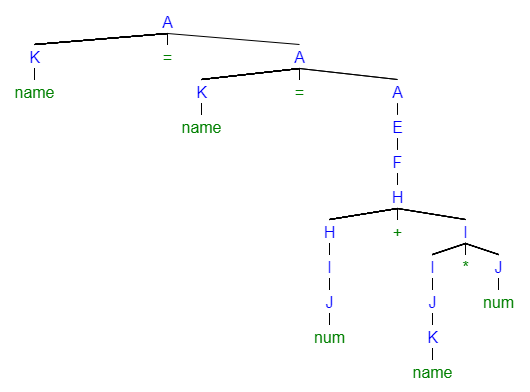
<K>::= name

Problem 2:

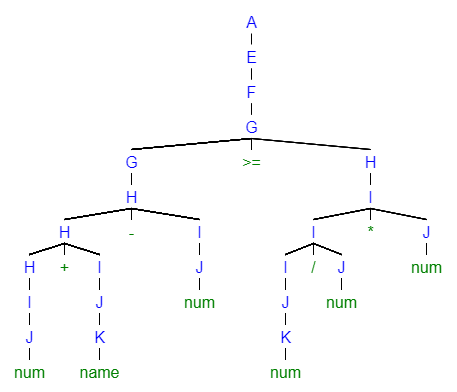
i >= 0 + j && x <= y || x > 0



y = z = 5 + x \* 4

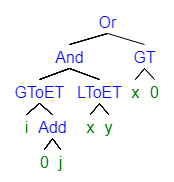


5 + x - 4 >= y / 3 \* 4



Problem 3

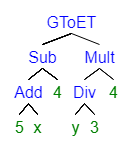
To really capture the essence of the expression i >= 0 + j && x <= y || x > 0, an abstract syntax tree is beneficial to depict the underlying steps in evaluating the Boolean result. Each step is a Boolean expression itself, thus the tree ensures that the precedence of evaluating the entire expression is maintained. Abbreviations for some of the operators used in this tree are (Greater Than: GT, Greater Than or Equal To: GToET, Less Than or Equal To: LToET).



With the expression y = z = 5 + x \* 4 , a linear notation is useful in showing the values that are inevitably assigned to the variable y.

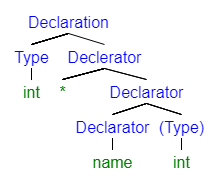
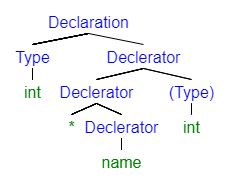
**Assign(y, Assign(z, Add(5, Mult(x, 4))))**

For the last expression 5 + x - 4 >= y / 3 \* 4, another abstract syntax tree is useful to once again depict the Boolean expression in order to evaluate it. Less Than or Equal To is abbreviated here as well.



Problem 4

a. Possible String: int \* name(int)

**Tree 1** **Tree 2**

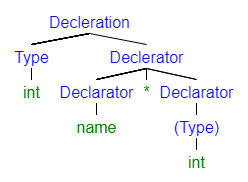
b. Declaration ::= Type Dec1;

Type ::= **int** | **char**

Dec1 ::= ∗ Dec1 | Dec2

Dec2 ::= Dec2 ‘[‘ **number** ‘]’ | Dec2 ‘(‘ Type ‘)’ | ‘(‘Dec1 ‘)’ | **name**

c. The resulting grammar is now unambiguous since changing \* to be a postfix operator ensures that strings that are read with the \* symbol are left associative. Thus, the \* symbol will ensure that generated parse trees are always the same. For example, the string int name \* (int) would only have the following parse tree:



Problem 5:

a. <S> ::= []

| **id** := **expr**

| **if** **expr** **then** <SL><G><H> **end**

| **loop** SL **end**

| **while** **expr** **do** SL **end**

<SL> ::= <S><G>

<G> ::= [] // For 0 or more repetitions (replacing braces)

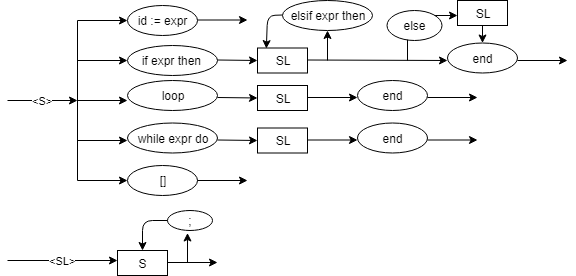
| **elsif** **expr** **then** <SL><G>

| ; <S><G>

<H> ::= [] // For optional parts (replacing brackets)

| **else** <SL>

b.



Problem 6

If operators at the same level of precedence were to not have the same associativity (e.g. left associativity and right associativity), then there would be no way to determine evaluation order for the expression. This can in turn cause different evaluations of the same string if no parenthesis are used. For example, the string 5\*2%4 can be evaluated as (5 \* 2) % 4 = 2 or 5 \* (2 % 4) = 10. Thus, having operators at the same level have the same associativity ensures that there is no ambiguity in evaluation.