Garrett Hall
CSC 173
C Programming
Weeks 3-4

1.0 Overview

The program parses code from an input file and evaluates it. The evaluator can handle basic mathematical operations (`+`, `-`, `*`, `/`, `^`) and trigonometric functions (`Sin`, `Cos`, `Tan`, `Asin`, `Acos`, `Atan`). In addition, the evaluator can handle variables and even solve simple algebraic equalities using a system of substitution rules (described next).

2.0 Evaluation

The evaluation system is described first because it provides examples for the capabilities and syntax of the program as a whole. To run these examples, run the program and type the filename at the prompt.

2.1 Substitution Rules

All algebra in the evaluation is based upon substitution rules which are indicated by the `->` token. If the left side of a rule is matched then the expression is replaced by the right side. For instance, the rule `a-a -> 0` will replace any two identical expressions that are subtracted with a `0`.

example1.1

```
a-a -> 0;
x-x;
```

```
x - x
0
```

Note that substitution rule variables such as `a` can match any complex expression such as `x*3`.

example1.2

```
a-a -> 0;
x*3-x*3;
```

```
(x * 3) - (x * 3)
0
```

Note: Substitution rule variables must be single character a-z and are unrelated to variables appearing outside of the build rule. For instance `x-x -> 0` *would have produced the same output as above.*

2.2 Normal Variables

Unlike variables that appear in a substitution rule, normal variables can bind to numbers. The assignment operator `=` indicates a binding when the variable is on the left and the number is on the right.

example2.1

```
x = 3;
x - 2;
```

```
x = 3
```

```
x - 2
3 - 2
1
```

It is important to note that variables cannot be reassigned and there is no symbol table. Instead, the expression `x=3` is similar to the rule '`x`'`->3` where '`x`' is a text match. For instance:

example2.2

```
x = 3;
x = 5;
```

```
x = 3
x = 5
3 = 5
False
```

The program can also handle multi-variable equations.

example2.3

```
z = y / 2;
y = 4;
```

```
z = (y / 2)
y = 4
z = (y / 2)
z = (4 / 2)
z = 2
```

2.3 Simple Algebra Rules

Rules and variables are combined to make basic algebra systems. The substitution rule algorithm is designed to simplify expressions as much as possible given a set of rules. A simple problem would be $x*2=4$. The evaluator doesn't know anything about dealing with variable expressions until we add some rules. By adding the rule `a*b=c -> a=c/b` evaluator can now deal with a larger subset of problems

example3.1

```
a*b=c -> a=c/b;
x * 2 = 4;
```

```
(x * 2) = 4
x = 2
```

To deal with multiplicands in reverse order ($2*x$) we can add a reflexive rule

example3.2

```
a*b -> b*a;
2 * y = 4;
```

```
(2 * y) = 4
(y * 2) = 4
y = 2
```

The example can be extended with identities

example3.3

```
a/a -> 1;
z * w = z;
```

```
(z * w) = z
(w * z) = z
w = 1
```

Or transitive rules

example3.4

```
a*(b*c) -> (a*b)*c;
2 * u * v = y;
```

```
(2 * (u * v)) = u
(2 * (v * u)) = u
((v * u) * 2) = u
((2 * v) * u) = u
(v * 2) = 1
v = 0.5
```

Note: The steps in red are omitted from output, but I've added them for clarity. This is because the evaluation algorithm is essentially looking two steps ahead.

2.4 Advanced Algebra

Here are some more advanced examples using a larger algebra base found in the `rules` file.

$2x^2 = 9y$

$\frac{y}{2z} = \frac{1}{z}$

example4.1

```
2 * x^2 = 9 * y;
y / 2 * z = 1 / z;
```

```
(2 * (x ^ 2)) = (9 * y)
(y / (2 * z)) = (1 / z)
(2 * (x ^ 2)) = (9 * y)
((x ^ 2) * 2) = (9 * y)
((x ^ 2) * 2) = (y * 9)
((4.5 * y) ^ 0.5) = x
(y / (2 * z)) = (1 / z)
(y / (z * 2)) = (1 / z)
(y / (2 * z)) = (1 / z)
((2 * z) / z) = y
((4.5 * y) ^ 0.5) = x
((y * 4.5) ^ 0.5) = x
((4.5 * y) ^ 0.5) = x
((x ^ 2) / y) = 4.5
((2 * z) / z) = y
((z * 2) / z) = y
2 = y
y = 2
((x ^ 2) / y) = 4.5
((x ^ 2) / 2) = 4.5
((x ^ 2) * 0.5) = 4.5
x = 3
```

The output is a rather verbose series of manipulations, but the correct values are finally arrived at.

Note: The evaluator cannot handle systems of inequalities or equations, although it would be possible to extend it to do so. Multivariable equations are only solvable if assignments can be found for all variables except one.

Here we solve for the double angle identities. Note these identities are not rules themselves, but are found by expansion

example4.2

```
Cos(x)^2+Sin(x)^2;
```

```
((Cosx) ^ 2) + ((Sinx) ^ 2)
((Cosx) * (Cosx)) + ((Sinx) * (Sinx))
Cos(x - x)
Cos(0)
1
```

The `gcc` math library for `acos` returns a single number

example4.3

```
0 = Cos(x)^2-Sin(x)^2;
```

```
0 = (((Cosx) ^ 2) - ((Sinx) ^ 2))
0 = (((Cosx) ^ 2) - ((Sinx) ^ 2))
0 = (Cos(2 * x))
(x * 2) = (Acos0)
(x * 2) = (Acos0)
(x * 2) = 1.5708
x = 0.785398
```

2.5 Evaluation Algorithm

Because evaluation is not the main focus of this assignment I won't discuss it in much detail although it is the most complex aspect of the program. The evaluator transverses the AST built by the parser (detailed later). At each node, the evaluator checks if the node matches any substitution rule. When multiple substitutions are possible the algorithm performs a breadth-first search of fixed depth on the substitution rules. The resulting node with the best heuristic value is chosen. This heuristic favors eliminating variables, simplifying expressions, and moving variables to the left side of an expression (i.e. `x=2` is better than `2=x`). If numbers are found on either side of an operator node, the operator is replaced by the numeric result of the operation.

3.0 Implementation

The program does the following:

1. Reads input from file specified in `main.c`
2. Tokenizes it in `scanner.c` using a DFA
3. Parses it in `parser.c` using a recursive-descent LL(1) parser
4. Builds abstract symbol trees (binary tree data structures described in `AST.c`)
5. Evaluates the code in `evaluate.c`

3.1.0 Language

Below the language is described in terms of the regular expressions accepted by the scanner and the context free grammar (CFG) accepted by the parser.

3.1.1 Character Classes

$digit \rightarrow 0|1|2|3|4|5|6|7|8|9$

lower → a | b | ... | z

upper → A | B | ... | Z

3.1.2 Accepted Tokens

token → *number* | *variable* | *function* | −> | *white* |) | * | + | / | − | ^ | (| ; | *eof*

number → *digit* *digit** (. *digit**) | ε

variable → *lowercase* (*lowercase* | *uppercase* | *digit*)*

function → *uppercase lowercase lowercase**

(Function keywords are: `Cos`, `Sin`, `Tan`, `Acos`, `Asin`, and `Atan`)

white → *tab* | *newline* | *space*

comparison → < | <= | = | >= | >

3.1.3 Context-free Grammar

statement → *rule* ; *statement* | ε

rule → *expression* −> *expression* | *expression*

The -> token indicates a special substitution rule.

expression → $field_0$ *comparison* $field_0$ | $field_0$

Comparisons will evaluate to `true` or `false` although '=' will act as an assignment to an unbound variable. For example `x=2` will act as an assignment statement if `x` was previously unassigned. Otherwise it will be a comparison.

$field_0$ → $field_1$ + $field_0$

$field_0$ → $field_1$

$field_0$ → ($field_0$)

The $field_i$ are non-terminals which represent the order of operations. Specifically *i* is the priority of the operation. The non-terminals are right-recursive which allows LL(1) parsing (no infinite loops) and creates parse trees.

$field_1$ → $field_2$ − $field_1$ | $field_2$ | ($field_0$)

$field_2$ → $field_3$ / $field_2$ | $field_3$ | ($field_0$)

$field_3$ → $field_4$ * $field_3$ | $field_4$ | ($field_0$)

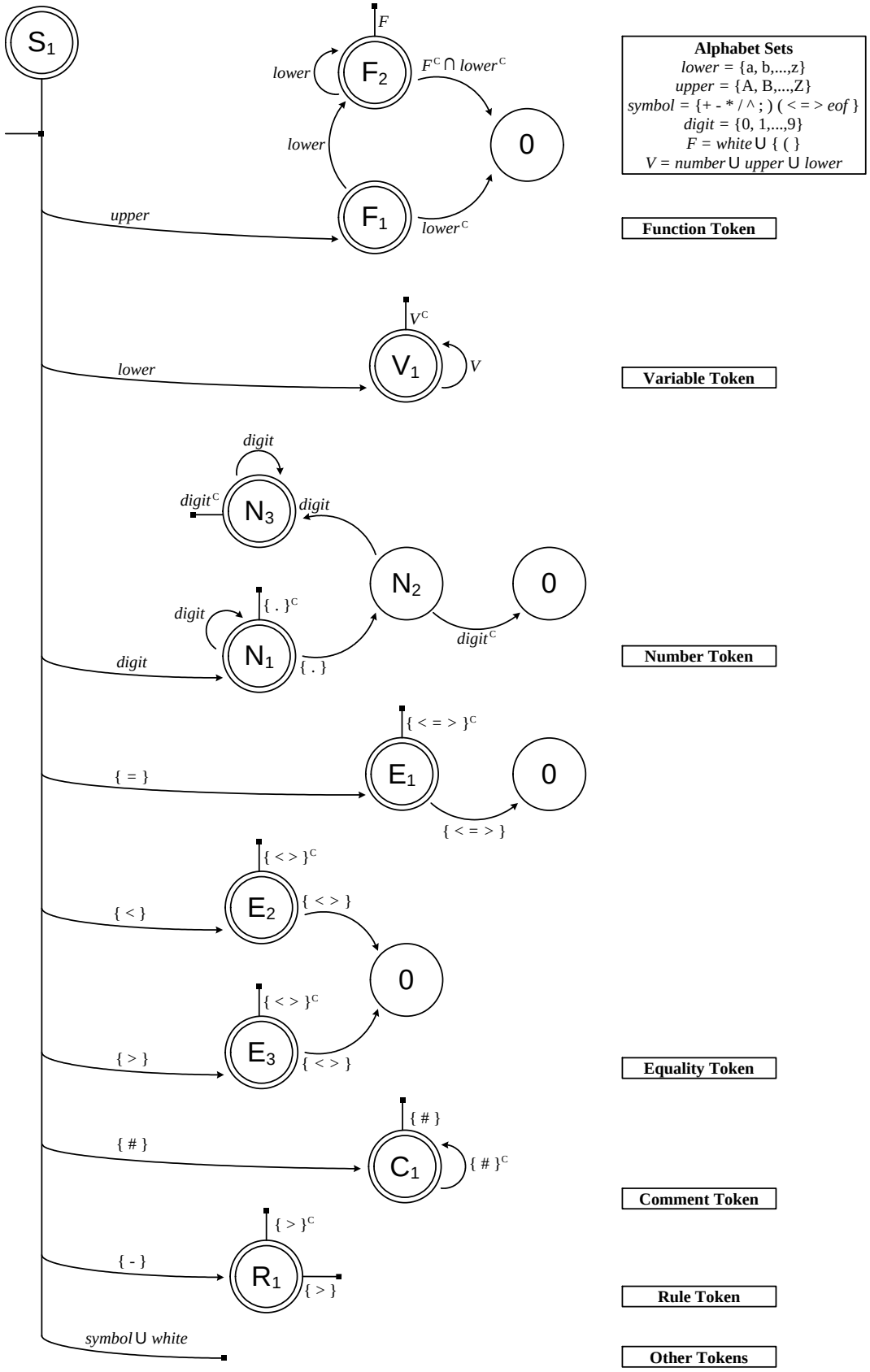
$field_4$ → $field_5$ ^ $field_4$ | $field_5$ | ($field_0$)

$field_5$ → −*atom* | *atom*

atom → *number* | *variable* | *function* ($field_0$)

The −*atom* is a signed atom and uses a flag so that it can only occur in the first atom. Otherwise the negative sign could be infinitely nested. Function keywords are `Cos`, `Sin`, `Tan`, `Acos`, `Asin`, and `Atan`.

3.2 Scanner DFA

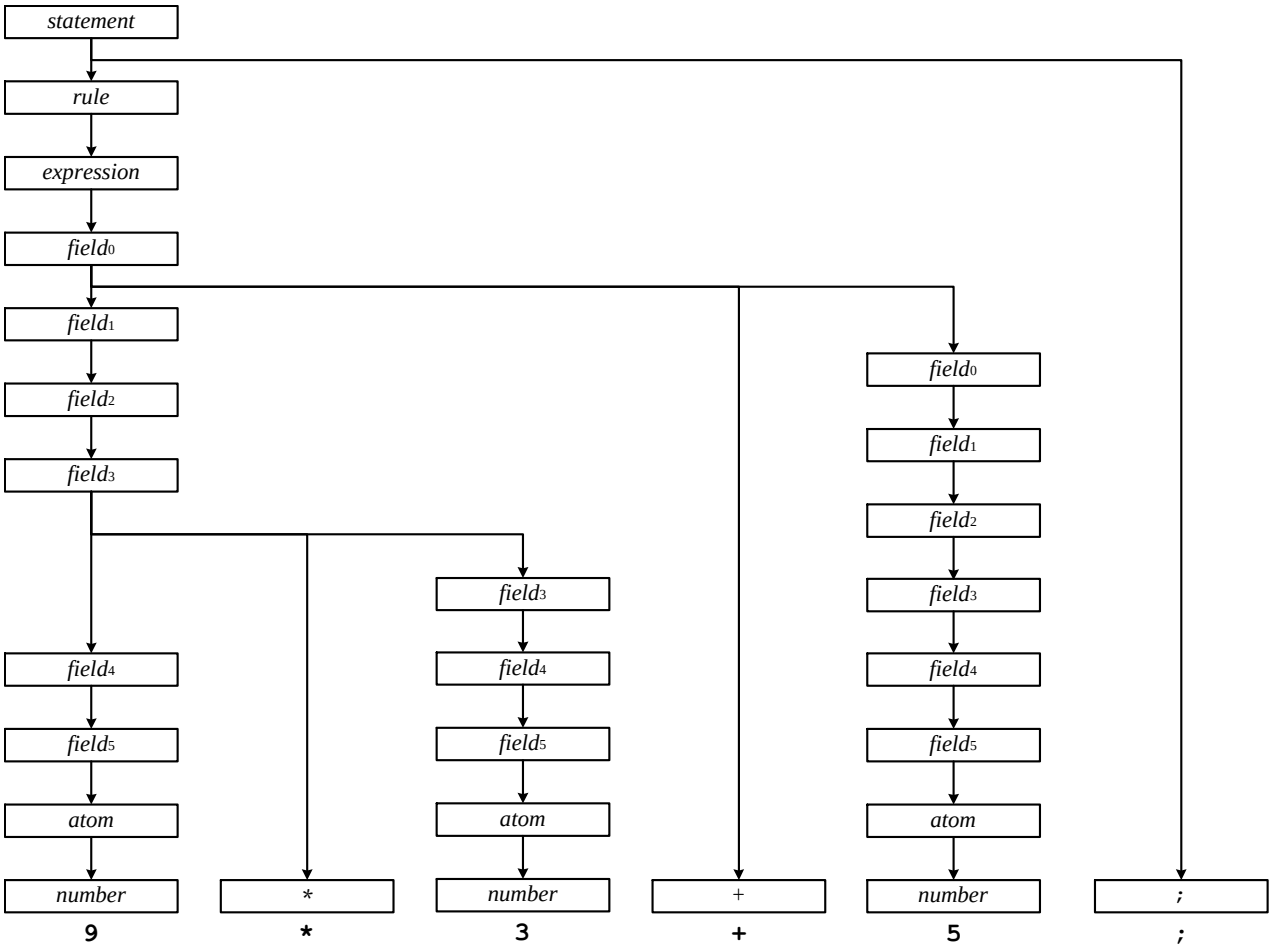


3.3.0 Abstract Symbol Tree (AST)

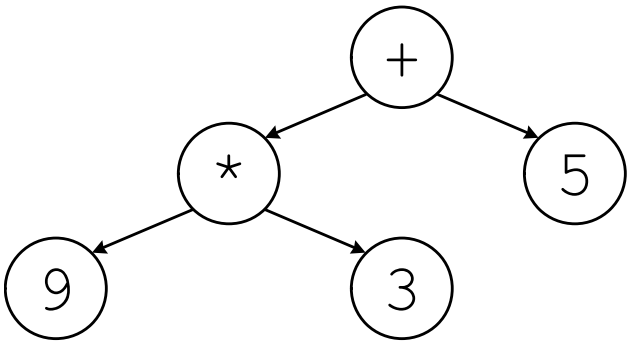
The parser which implements the CFG generates a binary tree. This is not the concrete parse tree because redundant non-terminals are not added to the tree, so it is an AST. For example, the production arrived at through $\rightarrow field_3 \rightarrow field_4 \rightarrow field_5 \rightarrow atom \rightarrow number$ is reduced to simply *number*. Each node in the AST has an associated class:

- **`AST_BINARY:`** a binary operation on two nodes '`+`'
- **`AST_UNARY:`** a unary operation on one node '`Cos`'
- **`AST_COMPARE:`** comparison between two nodes '`<=`'
- **`AST_NUMBER:`** single node that is a number '`9.32`'
- **`AST_VARIABLE:`** single node that is a variable '`myVar`'
- **`AST_BOUND:`** single node that is an assigned variable '`myVar = 9.32`'

This is the parse tree for the statement `9*3+5;` Notice the order of operations inherent in the tree structure:



The AST is simply:



3.3.1 Generating the AST

To view the tree-structure of input the flag in `main.c` must be changed by changing the line `tree_flag(0);` to `tree_flag(1);`

codefile
`1*2+3^4/(5+6);`

```
+
 *
  1
  2
 /
  ^
   3
   4
  +
   5
   6
```

The output is in prefix notation. If the flag is turned off `tree_flag(0);` the code will simply evaluate. In this case the result is in fully parenthesized infix notation:

```
(1 * 2) + ((3 ^ 4) / (5 + 6))
2 + ((3 ^ 4) / (5 + 6))
2 + (81 / (5 + 6))
2 + (81 / 11)
2 + 7.36364
9.36364
```

4.0 Error Handling

Below are a few examples of possible errors.

Comments must always begin and close with the pound sign.

```
# Good #
# Bad
```

```
Error end of file in comment at line 2, col 0
```

Because testing for equality and assignments are interchangeable only the token "=" should be used.

```
x == 4;
```

```
Comparison '==' should be '=' at line 1, col 2
```

Operators should not occur next to each other. Parentheses avoid this problem.

```
- 2 / + 3;
```

```
Atom expected '+' found at line 1, col 6
```

```
- 2 / (+ 3);
```

```
(-2) / (+3)
-2 / (+3)
-2 / 3
-0.666667
```

The format of functions must include parentheses and the first letter must be capitalized.

```
Cos0;
```

```
'0' cannot be in function name at line 1, col 0
```

```
cos(0);
```

```
';' expected '(' found at line 1, col 3
```

```
Cos 0;
```

```
'0' cannot be in function name at line 1, col 0
```

```
Cos(0);
```

```
Cos0
1
```