ASTs, Regex, Parsing, and Pretty Printing

CS 2112 Fall 2016

1 Algebraic Expressions

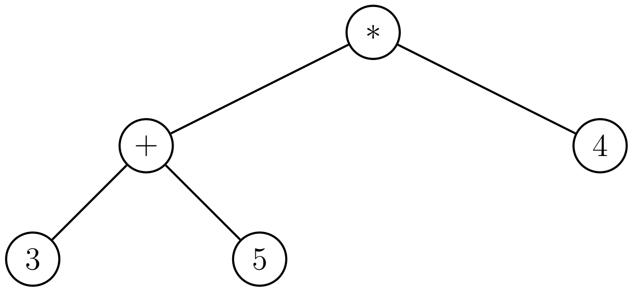
To start, consider integer arithmetic. Suppose we have the following

1. The *alphabet* we will use is the digits $\{0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9\}$.
2. We will allow the unary operator (one argument) $-$ for negation.
3. We will allow the binary operators:

$+$	addition
$-$	subtraction
$*$	multiplication
$/$	division

For ease and clarity of presentation, when we represent expressions as Abstract Syntax Trees (AST), we will only allow for two children for a given operation. In this is the maximum allowed, as we only have binary operators. However, we will see ASTs later that will be more flexible.

Consider the following AST for integer algebra



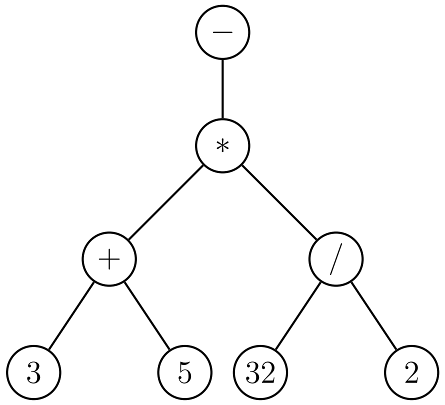
We parse it left to right, and bottom up, so it represents the expression

$$(3+5)*4$$

equally important, it is very much **not**

$$3+5*4$$

Another example:



This example represents the expression

$$-(\ (3+5)\ *\ (32/2)\ )$$

Before continuing with more examples, lets revisit regular expressions.

2 Regular Expression Review

2.1 The "alphabet" of regular expressions

The alphabet of a regular expression is denoted by Σ. This *set* is what defines what are "valid" components of the regular expression. For example,

1. $\Sigma = \{0,\ 1\}$

   This is the alphabet of all *binary* strings, encoded as *0*'s and *1*'s.

2. $\Sigma = \{a,\ b,\ c\}$

   This is the alphabet of all strings that contain either

   i) all a's
   ii) all b's
   iii) all c's
   iv) all a's and b's
   v) all a's and c's

vi) all b's and c's
vii) all a's and b's and c's

As you can see, the ability to define the alphabet of a regular expression using set notation is exceptionally convenient.

2.2 The ***Kleene Star*** ($*$) is a post-fix operator

This operator represents *0 or more* occurrences of whatever it operates on. For example,

1. $d^* \longrightarrow \{\varepsilon,\ d,\ dd,\ ddd,\ \cdots\}$
2. $(01)^* \longrightarrow \{\varepsilon,\ 01,\ 0101,\ 010101,\ \cdots\}$
3. If $\Sigma = \{a,\ b\}$, then $\Sigma^* = \{\varepsilon,\ a,\ b,\ ab,\ ba,\ aab,\ aba,\ baa,\ \cdots\}$

where ε is the *empty* string. Later, we will use $\varnothing$ to represent the empty set.

2.3 A plus sign ($+$) represents a ***union***

Recalling that regular expressions are *sets*, we use the notation

$$A + B \longrightarrow A \cup B$$

For example,

1. $\{a,\ b\} + \{c,\ d\} = \{a,\ b,\ c,\ d\}$
2. $\{a,\ b,\ c\} + \{b,\ c,\ d\} = \{a,\ b,\ c,\ d\}$

When we use $+$, it is a binary operator (it requires two arguments). Additionally, note that the unions produced do not have duplicated elements – in the second example, we do not have duplicate b or c.

2.4 A dot product ($\cdot$) represents a ***concatenation***

We use the notation

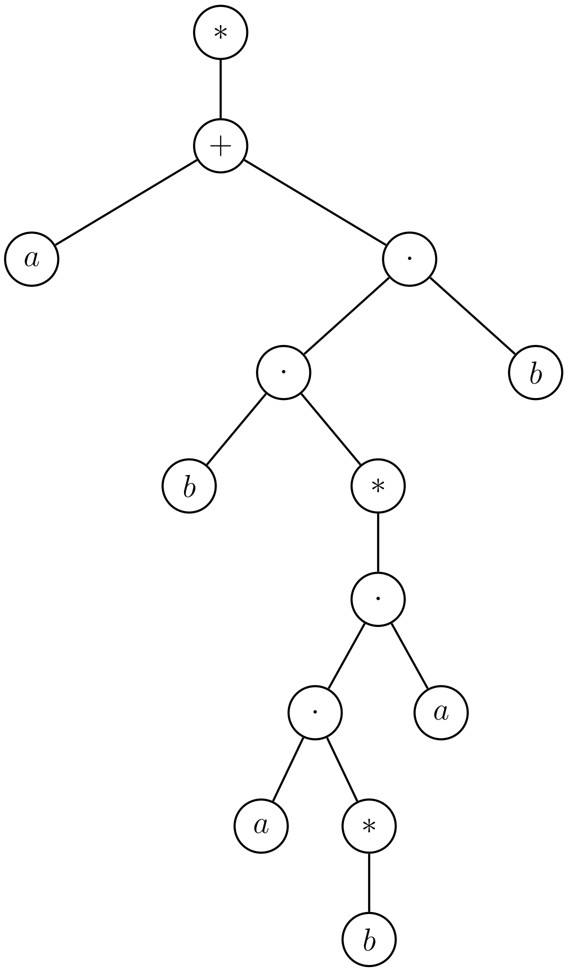
$$A \cdot B \longrightarrow \{xy \mid x \in A,\ y \in B\}$$

For example,

1. $\{a,\ b\} \cdot \{c,\ d\} = \{ac,\ ad,\ bc,\ bd\}$
2. $\{a,\ b,\ c\} \cdot \{b,\ c,\ d\} = \{ab,\ ac,\ ad,\ bb,\ bc,\ bd,\ cb,\ cc,\ cd\}$

3 AST's and regular expressions

Consider the AST



This describes the regular expression

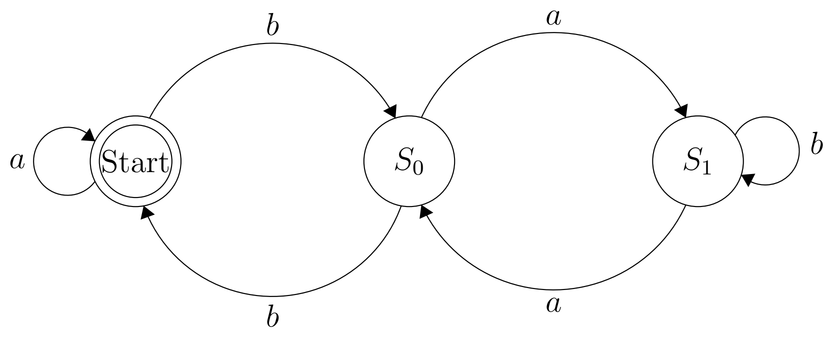
$$(a + b (ab^*a)^* b)^*$$

which, when written as parsed from above with parentheses to group (parentheses colored to help hit be a little more understandable):

$$(a + ((b \cdot ((a \cdot b^*) \cdot a)^*) \cdot b))^*$$

4 Alternate interpretation

As it turns out, we can use (deterministic) finite state machines to represent the regular expression presented in Section 3. The state machine would be as follows:



So in this example, the start state and the accept state (double circle) are the same thing. Convince yourself that the regular expression

$$(a + b (ab^*a)^* b)^*$$

is correctly represented by the finite state machine above by following the arrows. For example, if we saw the string "ba" then we would go from the "Start" state, to state "S_0", to state "S_1". This string would not be accepted by the finite state machine, which is good because it is not described by the regular expression.

On the other hand, consider the string "*abbbabbbbbbbab*". Follow the edges to see. The input will put you at the following states:

a	b	b	b	a	b	b	b	b	b	b	b	a	b
Start	S_0	Start	S_0	S_1	S_1	S_1	S_1	S_1	S_1	S_1	S_1	S_0	Start

So we can see that there is a direct relationship between the Kleene Star and self-loops in a state machine. Abstract Syntax Trees and Finite State Machines are both excellent conceptual models for understanding regular expressions, but they are not the only ones. In fact, they are just barely the tip of the iceberg!

5 Pretty Printing

When it comes to pretty printing ASTs, recursion should feel the most natural. Accompanying this PDF is a full version of the snippet below, in `PrettyPrint.java`.

An excerpt from the program is sufficient for explaining the idea:

```
/* Class Node has
 *
 *     Node[] children = new Node[2];
 *
 * where the following invariant is maintained:
 *
 * - Index 0 and 1 are null:               terminal node
 * - Index 0 has value and index 1 is null: unary operator
 * - Index 0 is null and index 1 has value: post-fix operator
 * - Index 0 and index 1 both have values:  binary operator
 *
 * Clearly this tactic does not scale well, but illustrates the concept.
 */
public static void prettyPrint(Node n) {
    // terminal nodes just get printed
    if(n.isTerminal()) {
        System.out.print(n);
    }
    // print the unary op followed by its only child
    else if(n.isUnaryOp()) {
        System.out.print(n);
        prettyPrint(n.children[0]);
    }
    // post-fix operators get printed after their only child
    else if(n.isPostFixOp()) {
        prettyPrint(n.children[1]);
        System.out.print(n);
    }
    // binary operators, print the left child first, the op, then right
    else if(n.isBinaryOp()) {
        System.out.print("(");
        prettyPrint(n.children[0]);
        System.out.print(" " + n + " ");
        prettyPrint(n.children[1]);
        System.out.print(")");
    }
}
```