CS 536 Announcements for Thursday, February 13, 2025

Programming Assignment 2 – due Tuesday, February 18

Homework 2 – is available

Last Time

- why regular expressions aren't enough
- CFGs
  - formal definition
  - examples
  - language defined by a CFG
- parse trees
- Makefiles
- ambiguous grammars
- grammars for expressions
  - precedence
  - associativity
- grammars for lists

Today

- syntax-directed translation
- abstract syntax trees
- implementing ASTs

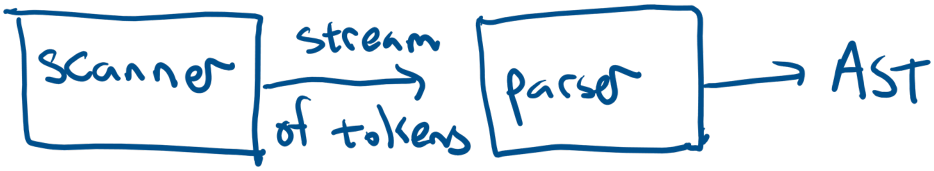
end of Midterm 1 material

CFG review

```
prog   → BEGIN stmts END
stmts  → stmts SEMICOLON stmt
       |  stmt
stmt   → ID ASSIGN expr
expr   → expr PLUS term
       |  term
term   → term TIMES factor
       |  factor
factor → expon POW factor
       |  expon
expon  → INTLIT
       |  LPAREN expr RPAREN
```

precedence
low
↓
high

Overview of CFGs



CFGs for language definition

- the CFGs we've discussed can generate/define languages of valid strings

start by building parse tree

& end with some valid string $w \in L(G)$

CFGs for language recognition

start with string w

& end with yes/no answer depending on whether $w \in L(G)$

CFGs for parsing

start with string w

& end with parse tree for w if $w \in L(G)$

↳ generally use AST instead of parse tree

– need to translate sequence of tokens (w)

Syntax-directed translation (SDT)

= translating from a sequence of tokens into a sequence of actions/other form, could be: AST, value, type, etc.
based on underlying syntax

To define a syntax-directed translation

Augment CFG with ***translation rules*** (at most 1 rule per production)
↳ LHS → RHS

- define translation of LHS non-terminal as a function of
  - constants
  - translations of RHS non-terminals
  - values of token (terminals) on RHS

To translate a sequence of tokens using SDT *

- build parse tree
- use translation rules to compute translation of each non-terminal in parse tree bottom-up ← handle children of node before node
- translation of sequence of tokens is the translation of the parse tree's root non-terminal (ie, start symbol)

The **type** of the translation can be anything: numeric, string, set, tree, ...

* Note: above is how to understand the translation, not how a compiler actually does it

Example: grammar for language of binary numbers

CFG	translation rules
b → 0	b.trans = 0
b → 1	b.trans = 1
b → b 0	b_1.trans = b_2.trans * 2
b → b 1	b_1.trans = b_2.trans * 2 + 1

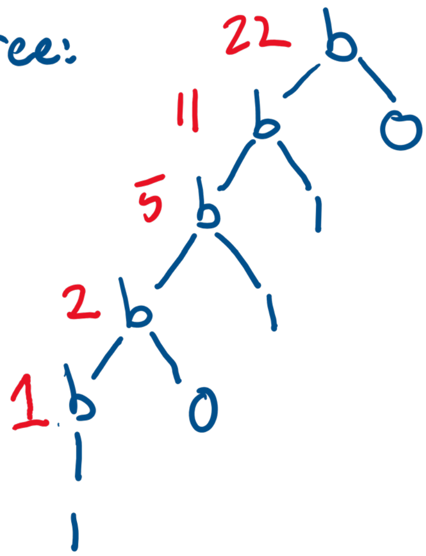
SDT to compute the decimal equivalent of a binary number

Example: input string 10110

Parse tree:

Translation is 22

steps ④ ③ ② ①



Example: grammar for language of variable declarations

CFG			Translation rules
declList	→	ε	$declList.trans = ""$
	|	decl $declList_2$	$declList_1.trans = decl.trans + " " + declList_2.trans$
decl	→	type ID ;	$decl.trans = ID.value$
type	→	INT	
	|	BOOL	

(string concatenation)

Write a syntax-directed translation for the CFG given above so that the translation of a sequence of tokens is a string containing the ID's that have been declared.

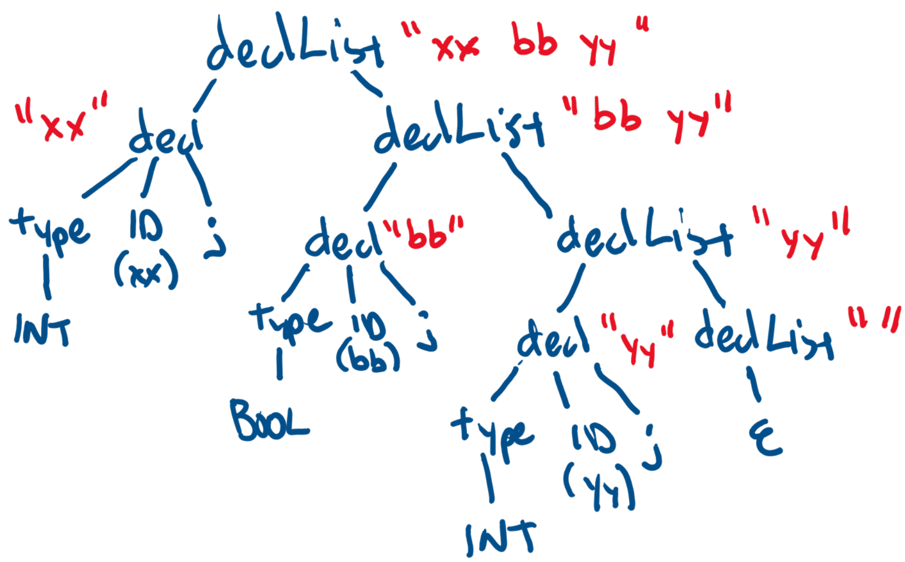
Example input

int xx;
bool bb;
int yy;

Output

"bb xx yy"
(in any order)

Parse tree



Translation is "xx bb yy"

Example: grammar for language of variable declarations

CFG			Translation rules
declList	→	ε	declList.trans = ""
	|	decl declList	$declList_1$.trans = decl.trans + " " + $declList_2$.trans
decl	→	type ID ;	decl.trans = type.trans ? ID.value : "" *
type	→	INT	type.trans = true
	|	BOOL	type.trans = false

Modify the previous syntax-directed translation so that only declarations of type `int` are added to the output string.

Example input

int xx;
bool bb;
int yy;

Output

"xx yy"
(in any order)

Note:
1) different non-terms can have different types as their translation
2) translation rules can be conditional

* x = a ? b : c; equiv to if (a)
x = b;
else
x = c;

SDT for parsing

Previous examples showed SDT process assigning different types to the translation

- translate tokenized stream to an integer value
- translate tokenized stream to a string

For parsing, we'll need to translate a tokenized stream to an **abstract-syntax tree (AST)**

Abstract syntax trees

AST = condensed form of parse tree

- operators are internal nodes (not leaves)
- chains of productions are collapsed
- lists are flattened
- syntactic details are omitted
  - ↳ eg ; parens

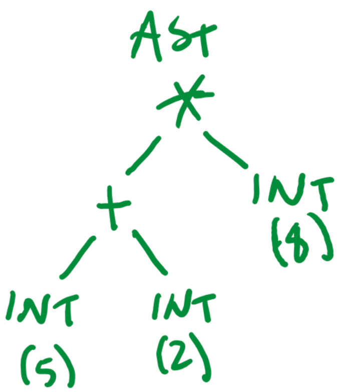
AST Example

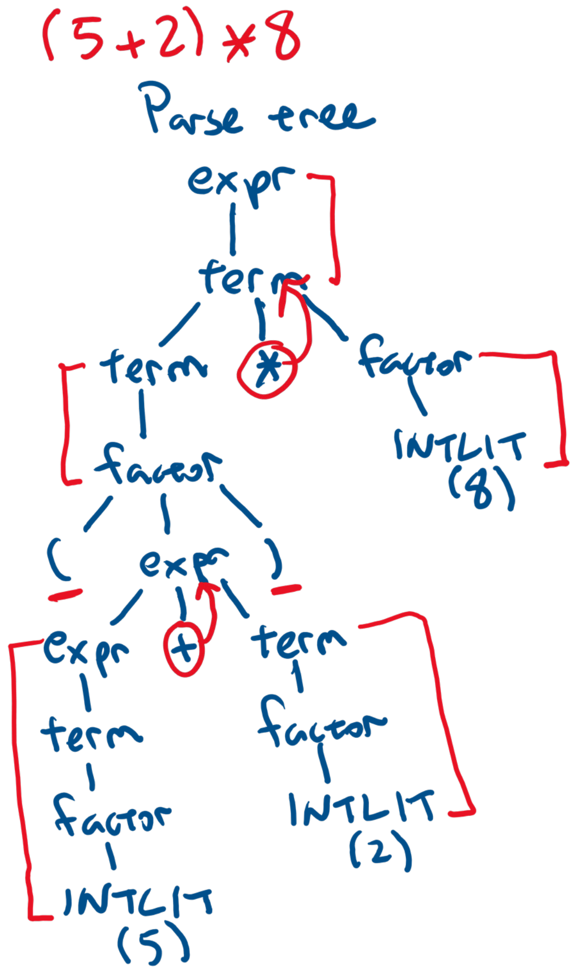
<u>CFG</u>

expr → expr PLUS term
     | term

term → term TIMES factor
     | factor

factor → INTLIT
     | LPAREN expr RPAREN





Résume at 7:30

SDT review

SDT = translating from a sequence of tokens into a sequence of actions/other form,
based on underlying syntax

To define a syntax-directed translation

- augment CFG with ***translation rules*** lhs → rhs (contains terms, non-terms, ε)
    - define translation of LHS non-terminal as a function of:
        - constants 2, " "
        - translations of RHS non-terminals rhs.trans
        - values of terminals (tokens) on RHS TOKEN.value

To translate a sequence of tokens using SDT (conceptually)

- build parse tree
- use translation rules to compute translation of each non-terminal (bottom-up)
- translation of sequence of tokens = translation of parse tree's root non-terminal

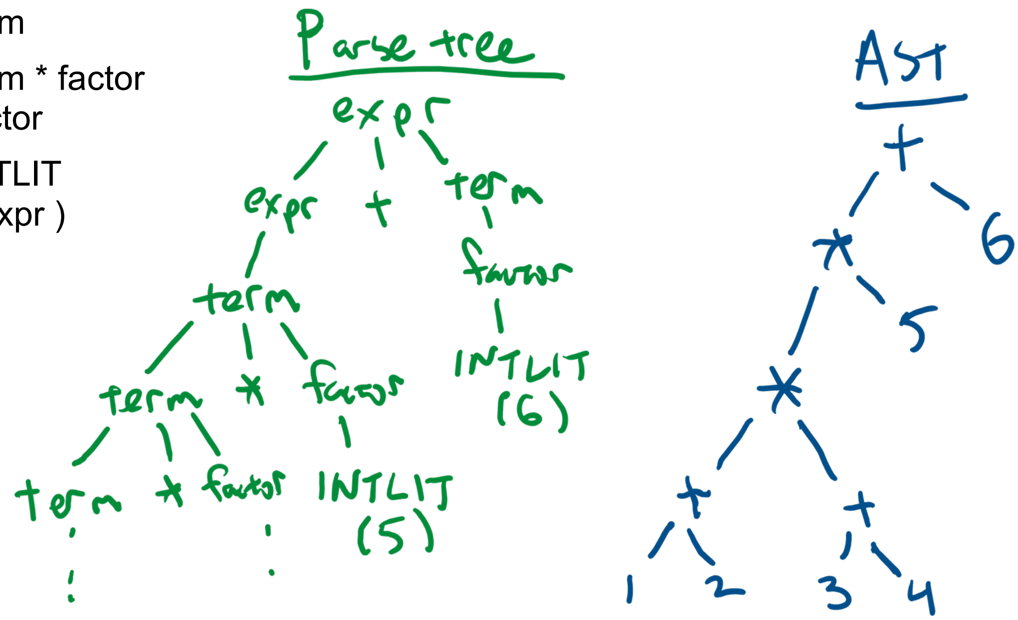
For parsing, we'll need to translate tokenized stream to **abstract-syntax tree (AST)**

Example (1+2)*(3+4)*5+6

expr → expr + term
| term

term → term * factor
| factor

factor → INTLIT
| (expr)



translation rule:

$expr_1.trans = MkPlusNode(expr_2.trans, term.trans)$

AST for parsing

We've been showing the translation in two steps:

token stream → parse tree → AST then throw away parse tree

In practice we'll do

token stream → AST

Why have an AST?

- captures essential structure
- easier to work with

AST implementation

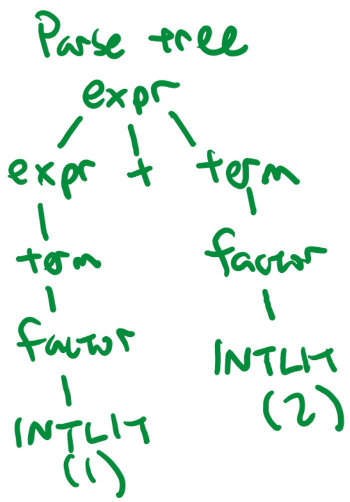
expr → expr + term $expr_1.trans$ = MkPlusNode($expr_2.trans$, term.trans)

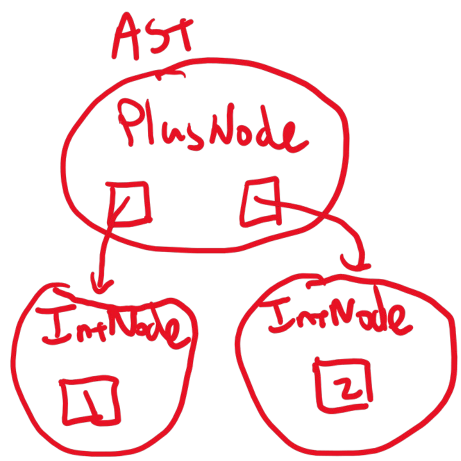
Define a class for each kind of AST node

Create a new node object in some rules

- new node object is the value of LHS.trans
- fields of node object come from translations of RHS non-terminals

Given 1+2





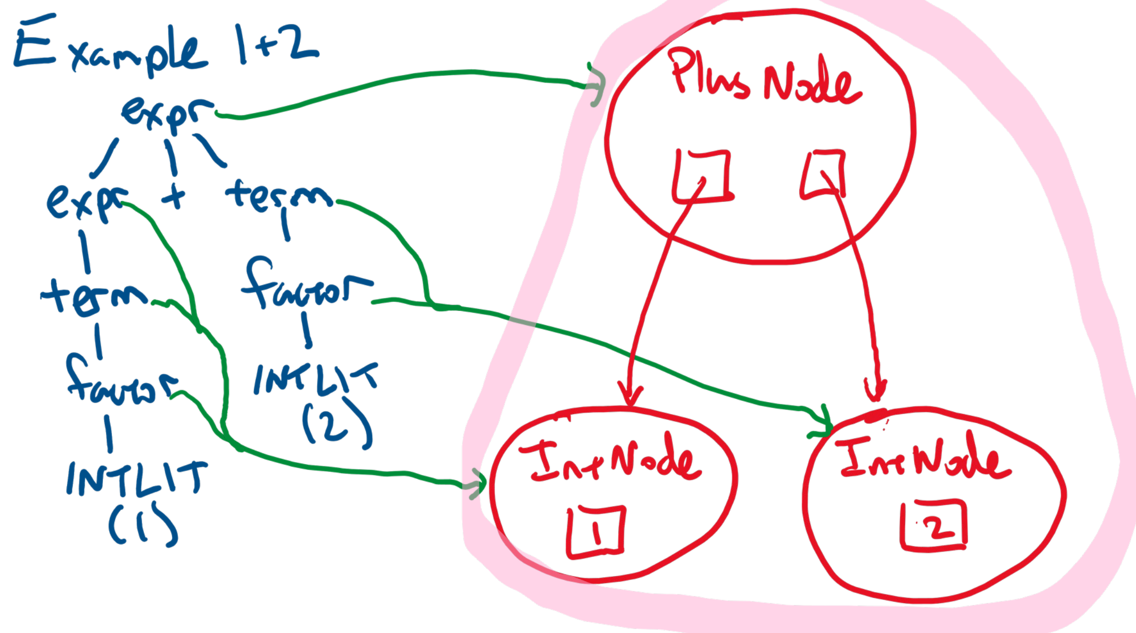

```
class PlusNode {
  IntNode left;
  IntNode right;
}
class IntNode {
    int value;
}
```

Need class hierarchy & make subclasses of ExpNode

```
class PlusNode extends ExpNode {
    ExpNode left;
    ExpNode right;      → put into ExpNode
}
class IntNode extends ExpNode {
    int value;
}
```

Translation rules to build ASTs for expressions

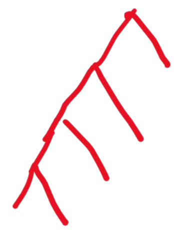
CFG			Translation rules
expr	→	expr + term	expr$_1$.trans = new PlusNode(expr$_2$.trans, term.trans)
	|	term	expr.trans = term.trans
term	→	term * factor	term$_1$.trans = new TimesNode(term$_2$.trans, factor.trans)
	|	factor	term.trans = factor.trans
factor	→	INTLIT	factor.trans = new IntNode(INTLIT.value)
	|	(expr)	factor.trans = expr.trans

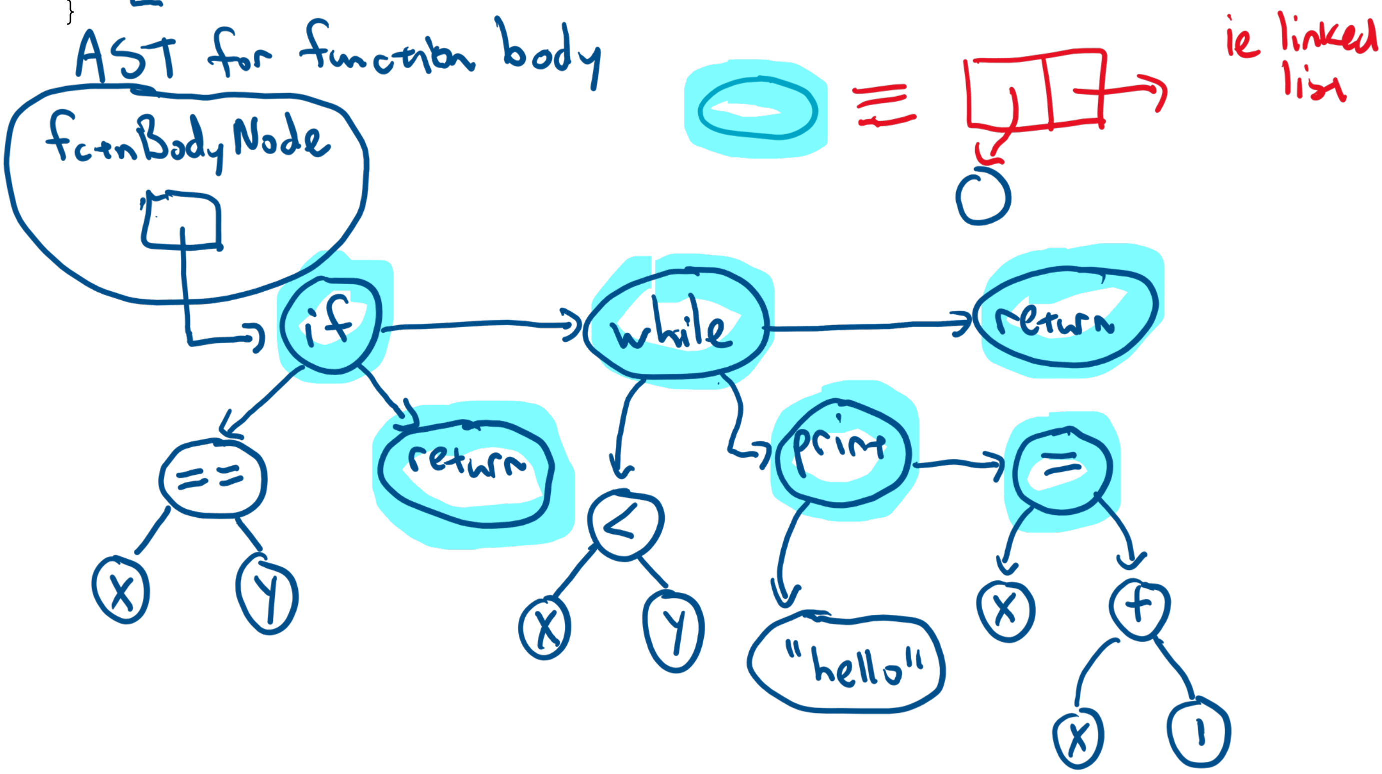


ASTs for non-expressions

Example

```
void foo(int x, int y) {
    if (x == y) {
        return;
    }
    while (x < y) {
        cout << "hello";
        x = x + 1;
    }
    return;
}
```





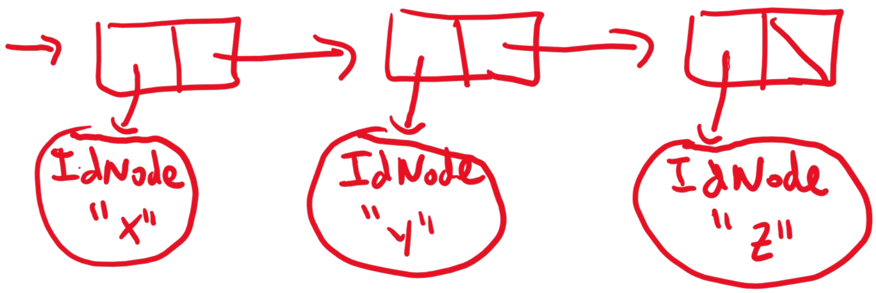
ASTs for lists

<u>CFG</u>

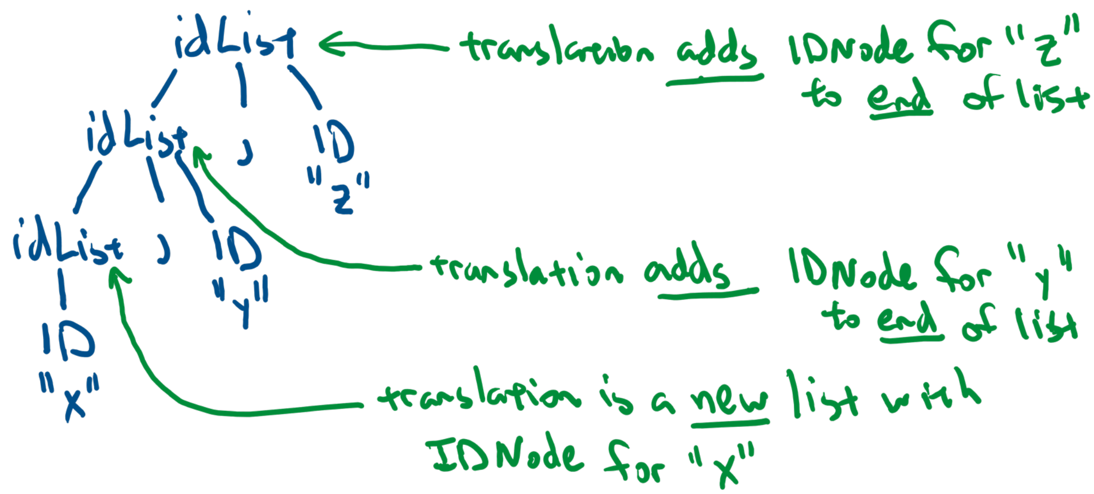
idList → idList COMMA ID
| ID

Input x, y, z

Want AST to be



Parse tree



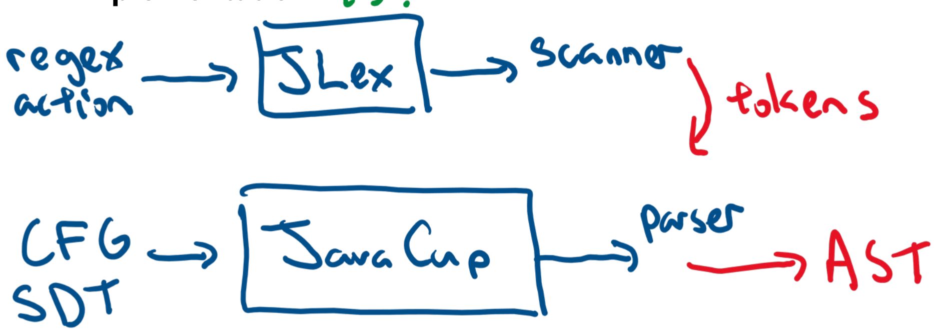
The bigger picture

Scanner

- **Language abstraction**: regular expressions
- **Output**: token stream
- **Tool**: JLex
- **Implementation**: interpret DFA using table (for δ), recording most_recent_accepted_position & most_recent_token

Parser

- **Language abstraction**: CFG
- **Output**: AST (by way of a syntax-directed translation)
- **Tool**: JavaCup ← next time
- **Implementation**: ??? ← next time



Next Time

- Java CUP
- approaches to parsing
- bottom-up parsing
- CFG transformations
- CYK algorithm