CS 536 Announcements for Thursday, February 20, 2025

Last Time

- syntax-directed translation
- abstract syntax trees
- implementing ASTs ← end of Midterm 1 material

Today

- Java CUP
- approaches to parsing
- bottom-up parsing
- CFG transformations
  - removing useless non-terminals
  - Chomsky normal form (CNF)
- CYK algorithm

Next Time

- Midterm 1, 6:30 – 8 pm (in-class)

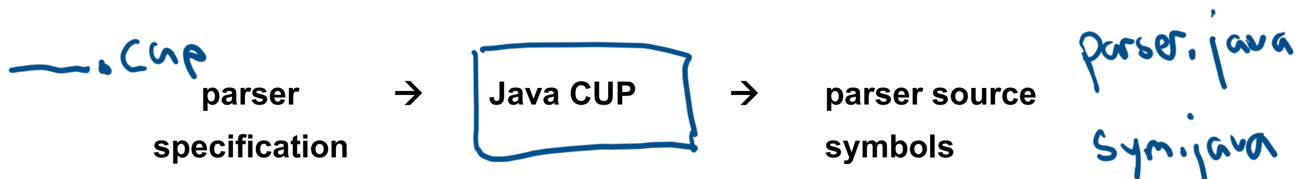
Parser generators

Tools that take an SDT spec and build an AST

- **YACC** Yet Another C Compiler – creates a parser in C
- **Java CUP** Constructor of Useful Parsers – creates a parser in Java

Conceptually similar to JLex:

- Input: language rules + actions
- Output: Java code



Java CUP

`parser.java`

- constructor takes argument of type `Yylex`
- `parse` method
  - if input correct, returns `Symbol` whose `value` field contains translation of root nonterm
  - if input incorrect, quits on first syntax error
- uses output of JLex
  - depends on scanner and `TokenVal` classes
  - `sym.java` defines the communication language
- uses definitions of AST classes (in `ast.java`)

Parts of Java CUP specification

Grammar rules with actions:

```
expr ::= INTLIT
      |   ID
      |   expr PLUS expr
      |   expr TIMES expr
      |   LPAREN expr RPAREN
      ;
```

Terminal and nonterminal declarations:

```
terminal     INTLIT;
terminal     ID;
terminal     PLUS;
terminal     TIMES;
terminal     LPAREN;
terminal     RPAREN;

non terminal expr;
```

Precedence and associativity declarations:

```
precedence left PLUS;
precedence left TIMES;
```

Java CUP Example

defined in ast.java

Assume:

- Java class `ExpNode` with subclasses `IntLitNode`, `IdNode`, `PlusNode`, `TimesNode`
- `PlusNode` and `TimesNode` each have two children
- `IdNode` has a `String` field (for the identifier)
- `IntLitNode` has an `int` field (for the integer value)
- `INTLIT` token is represented by `IntLitTokenVal` class and has field `intVal`
- `ID` token is represented by `IdTokenVal` class and has field `idVal`

defined in bach.jlex

Step 1: add types to terminals and nonterminals

```
/*
 * Terminal declarations
 */
terminal INTLIT;
terminal ID;
terminal PLUS;
terminal TIMES;
terminal LPAREN;
terminal RPAREN;
```

Need type if we want to use value associated with token

terminal InpLitTokenVal INTLIT;

terminal IdTokenVal ID;

from scanner (bach.jlex)

```
/*
 * Nonterminal declarations
 */
non terminal expr;
```

Type required for all non-terms

non terminal ExpNode expr;

from ast.java

Step 2: add precedences and associativities

```
/*
 * Precedence and associativity declarations
 */
precedence left PLUS;
precedence left TIMES;
```

Java CUP Example (cont.)

Step 3: add actions to CFG rules

```
/*
 * Grammar rules with actions
 */
expr ::= INTLIT:i                  <- type is IntLitTokenVal
         {:
           RESULT = new IntLitNode(i.intVal);
         :}
       | ID:i
         {:
           RESULT = new IdNode(i.idVal);
         :}
       | expr:e1  PLUS  expr:e2
         {:
           RESULT = new PlusNode(e1, e2);
         :}
       | expr:e1  TIMES  expr:e2
         {:
           RESULT = new TimesNode(e1, e2);
         :}
       | LPAREN  expr:e  RPAREN
         {:
           RESULT = e;
         :}
       ;
```

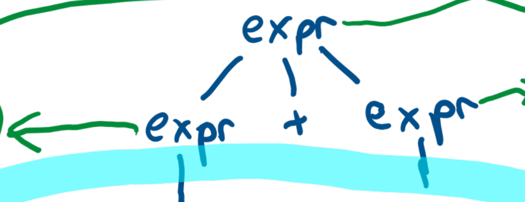
Subclasses of ExpNode

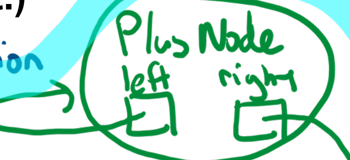
General format:

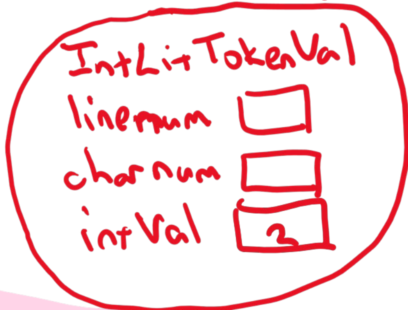
```
nonterm ::= rule1
            {: //action for rule1
               RESULT = ... ;
            :}
          | rule2
            {:
               RESULT = ... ;
            :}
          ⋮
          ;
```

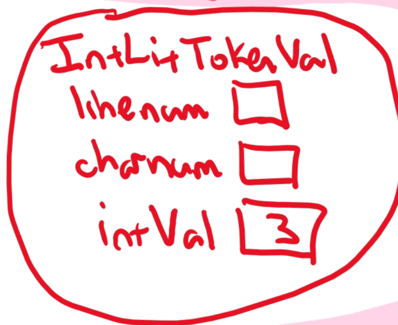
Java CUP Example (cont.)

Input: 2 + 3









Translating lists

Example ← left recursive

idList → idList COMMA ID | ID

Left-recursion or right-recursion?

- for top-down parsers must use right recursion
  left-recursion leads to an infinite loop
- for Java CUP use left recursion
  → bottom-up parser

Example

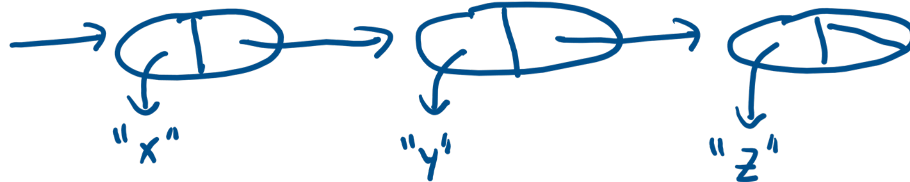
CFG: idList → idList COMMA ID | ID

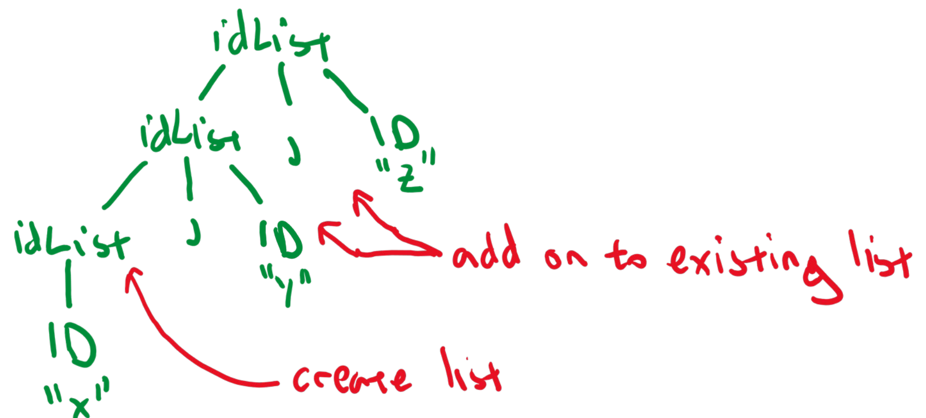
Goal: the translation of an `idList` is a `LinkedList` of `String`s

Example

Input: `x , y , z`

Output:





Example (cont.)

idList → idList COMMA ID
| ID

Java CUP specification for this syntax-directed translation

Terminal and nonterminal declarations:

```
terminal IdTokenVal ID;
terminal            COMMA;
non terminal LinkedList<String> idList;
```

Grammar rules and actions:

```
idList ::= idList:L     COMMA          ID:i
           {:
              L.addLast(i.idVal);
              RESULT=L;
           :}
        |  ID:i
           {:
              LinkedList<String> L=new LinkedList<String>();
              L.add(i.idVal);
              RESULT= L;
           :}
        ;
```

Handling unary minus

```
/*
 * precedences and associativities of operators
 */
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
precedence nonassoc UMINUS;
```

binary minus has lowest precedence

"phony" token (never returned by scanner)

unary minus has highest precedence

```
/*
 * grammar rules
 */
exp  ::= . . .
      |    MINUS exp:e
        {: RESULT = new UnaryMinusNode(e);
        :} %prec UMINUS
      |    exp:e1 PLUS exp:e2
        {: RESULT = new PlusNode(e1, e2);
        :}
      |    exp:e1 MINUS exp:e2
        {: RESULT = new MinusNode(e1, e2);
        :}
          . . .
      ;
```

Precedence of a rule is that of the last token in the rule, unless assigned a specific precedence using %prec

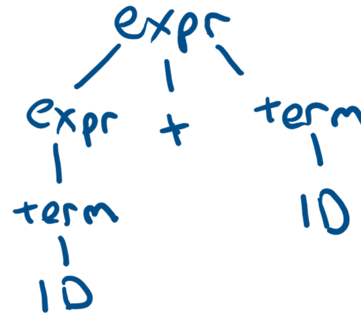
Resume at 7:40pm

Full-sized versions of diagrams for running the CYK algorithm (pages 11-12) available

Parsing: two approaches

Top-down / "goal driven"

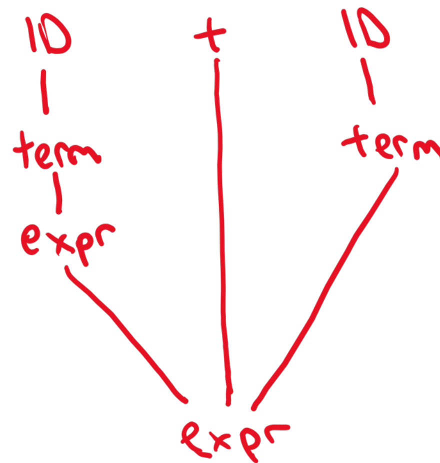
- start at start nonterminal
- grow parse tree downward until entire sequence is matched



Bottom-up / "data driven"

- start with terminals (sequence)
- generate ever larger subtrees until get to single tree whose root is the start nonterminal

(note: parse tree is upside down)



Example:

CFG: expr → expr + term | term

term → term * ID | ID

Derive: ID + ID

Cocke – Younger – Kasami (CYK) algorithm

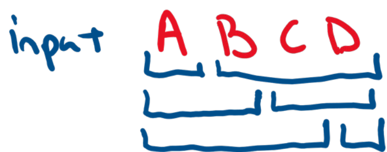
- Works bottom-up
- Time complexity : $O(n^3)$ n = length of input (# of tokens in sequence)
- Requires grammar to be in Chomsky Normal Form

Chomsky Normal Form (CNF)

- all rules must be in one of two forms
  - *x* → **T** (T is a terminal)
  - *x* → *a b*
- only rule allowed to derive epsilon is the start symbol *s*

Why CNF is helpful?

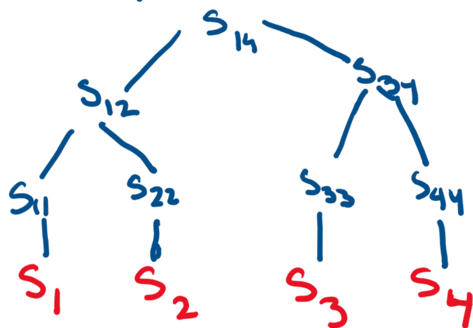
- nonterminals in pairs → can think of a subtree as a subspan
- nonterminals (except start) can't derive epsilon → each subspan has at least 1 token

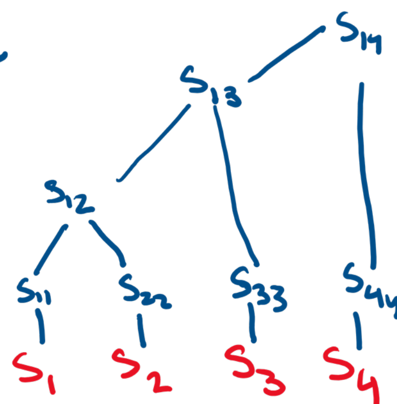


CYK : Dynamic Programming

x → **T** → forms leaf of parse tree

x → *a b* → binary interior node of parse tree

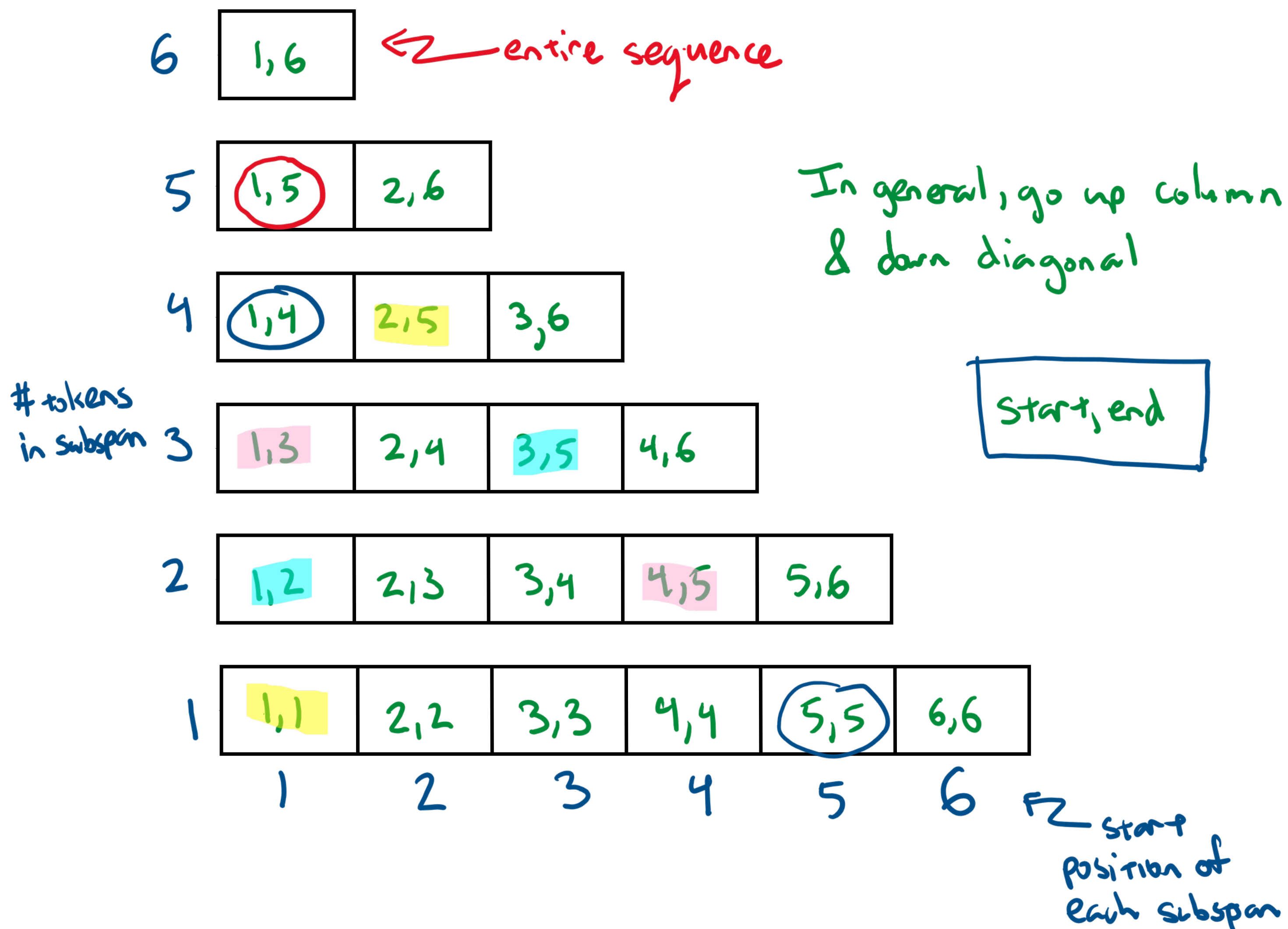




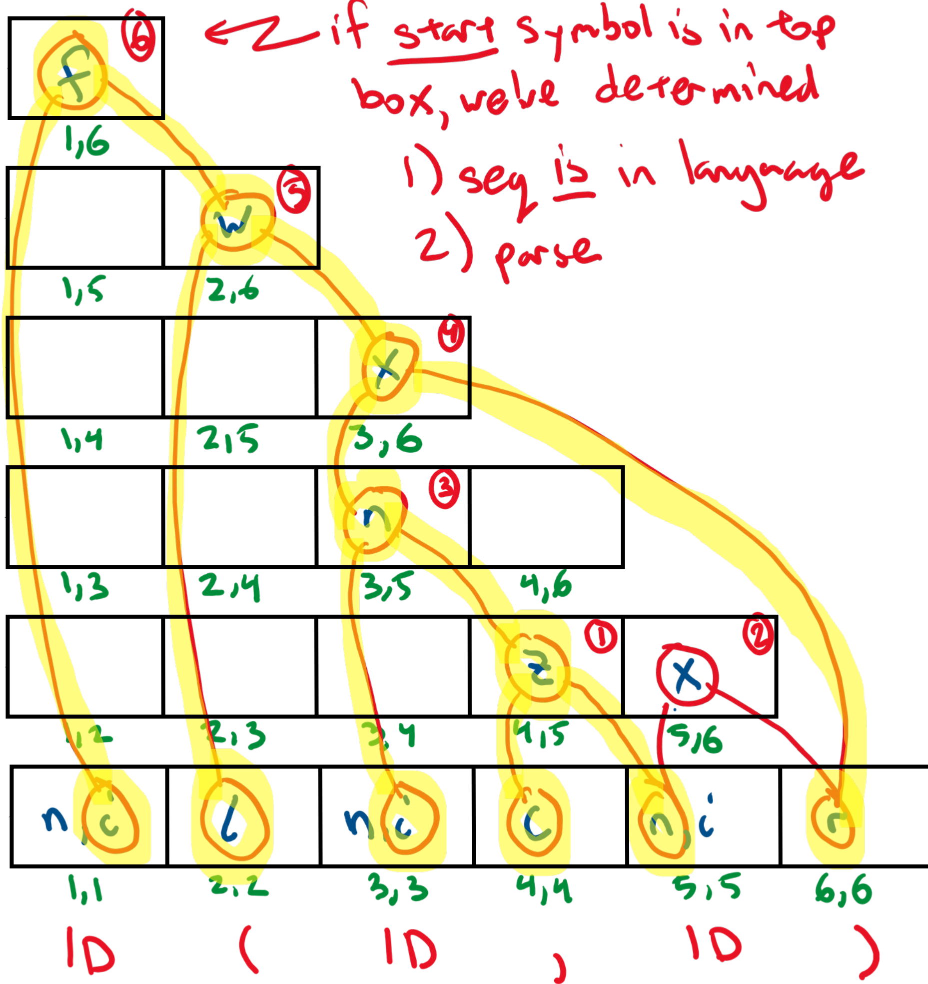
Running CYK

Track every viable subtree from leaf to root.

All subspans for a sequence (string) with 6 terminals



CYK Example



f → i w

f → i y

w → l x

x → n r

y → l r

n → **ID**

n → i z

z → c n

i → **ID**

l → **(**

r → **)**

c → **,**

Eliminating useless nonterminals

Avoid unnecessary work – remove ***useless*** rules

1. If a nonterminal cannot derive a sequence of terminal symbols, then it is ***useless***
2. If a nonterminal cannot be derived from the start symbol, then it is ***useless***

Nonterminals that cannot derive a sequence of terminal symbols

```
mark all terminal symbols
repeat
     if all symbols on the RHS of a production are marked
          mark the LHS nonterminal
until no more nonterminals can be marked
```

(every where it shows up)

Example

s → *x* | *y*
x → ( )
y → (*y y*)

S → X
| Y
ie S → X
S → Y

y is useless

S → X
X → ( )

Nonterminals that cannot be derived from the start symbol

```
mark the start symbol
repeat
     if the LHS of a production is marked
          mark all RHS nonterminals
until no more nonterminals can be marked
```

(wherever they show up)

Example

s → *a b*
a → **+** | **–** | ε
b → **digit** | *b* **digit**
c → **.** *b*

← c is useless

Chomsky Normal Form

Four steps

- eliminate epsilon productions
- eliminate unit productions
- fix productions with terminal on RHS (along with other symbols)
- fix productions with > 2 nonterminals on RHS

ok to have start S → ε
but if so, can't have start on RHS

if we have
S → ε
a → ... S ...
⇓ translate to
make new start symbol S0 & add production
S0 → S
S → ε
a → ... S ...

Eliminate (most) epsilon productions

If nonterminal *a* immediately derives epsilon

- make copies of all rules with *a* on RHS
- delete all combinations of *a* in the copies

Example 1

f → **ID (** *a* **)**
a → ε
a → *n*
n → **ID**
n → **ID ,** *n*

f → ID(a)
f → ID(a) ⇒ f → ID()
a → n
n → ID
n → ID, n

Example 2

x → *a* **X** *a* **Y** *a*
a → ε
a → **Z**

x → aXaYa
| aXaY
| aXYa
| XaYa
| aXY
| XaY
| XYa
| XY
a → Z

Chomsky Normal Form (cont.)

Eliminate unit productions

Productions of the form *a* → *b* are called ***unit productions***

If this is the only rule with *a* on the LHS
- place *b* anywhere a could have appeared
- remove the unit production *a* → *b*

Example

f → **ID (** *a* **)**
f → **ID ( )**
a → *n*
n → **ID**
n → **ID ,** *n*

f → ID (n)
  | ID ( )
n → ID
  | ID , n

If there are multiple rules with *a* on the LHS,
- for each rule of the form *b* → δ, add a → δ
- remove *a* → *b*

note: on-line reading is incorrect on how to handle this

Example

a → *b* **X**
  | *c b*
  | *b*
b → **Z Y**
  | **Y** *c*
c → **Z** *a*

a → b X
  | c b
  | Z Y
  | Y c
b → Z Y
  | Y c
c → Z a

Chomsky Normal Form (cont.)

Fix RHS nonterminals

For productions with terminals and something else on the RHS

- for terminal **T**, add rule *x* → **T**
- replace **T** with *x* in those productions

Example

f → **ID (** *n* **)**
f → **ID ( )**
n → **ID**
n → **ID ,** *n*

i → ID
l → (
r →)
c → ,
f → i l n r
f → i l r
n → ID
n → i c n

For productions with > 2 nonterminals on the RHS

- replace all but the 1st nonterminal with a new nonterminal
- add rule with new nonterminal on LHS and replaced nonterminal sequence on RHS
- repeat (as necessary)

Example

f → i l n r ⇒ f → i w, w → l n r ⇒ f → i w, w → l x, x → n r

Left for you: f → i l r
n → i c n