CS 536 Announcements for Thursday, February 20, 2025

Last Time

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

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**
- **Java CUP**

Conceptually similar to JLex:

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

parser specification → **Java CUP** → **parser source symbols**

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

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`

Step 1: add types to terminals and nonterminals

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

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

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
         {:


         :}
      |  ID
         {:


         :}
      |  expr     PLUS  expr
         {:


         :}
      |  expr     TIMES  expr
         {:


         :}
      |  LPAREN  expr     RPAREN
         {:


         :}
      ;
```

Java CUP Example (cont.)

Input: `2 + 3`

Translating lists

Example

idList → idList COMMA ID | ID

Left-recursion or right-recursion?

- for top-down parsers

- for Java CUP

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.)

Java CUP specification for this syntax-directed translation

Terminal and nonterminal declarations:

Grammar rules and actions:

```
idList ::= idList        COMMA        ID
           {:





           :}
        |  ID
           {:






           :}
        ;
```

Handling unary minus

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




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

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

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)
- Requires grammar to be in Chomsky Normal Form

Chomsky Normal Form (CNF)

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

Why CNF is helpful?

- nonterminals in pairs

- nonterminals (except start) can't derive epsilon

CYK : Dynamic Programming

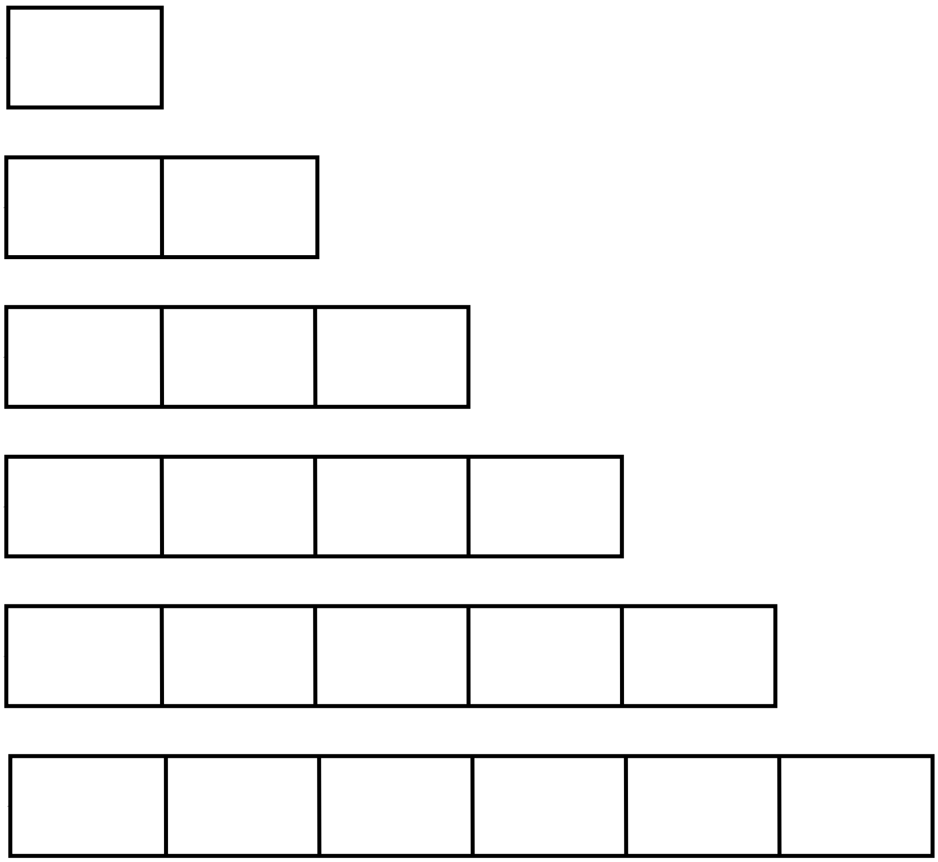
x → **T**

x → *a b*

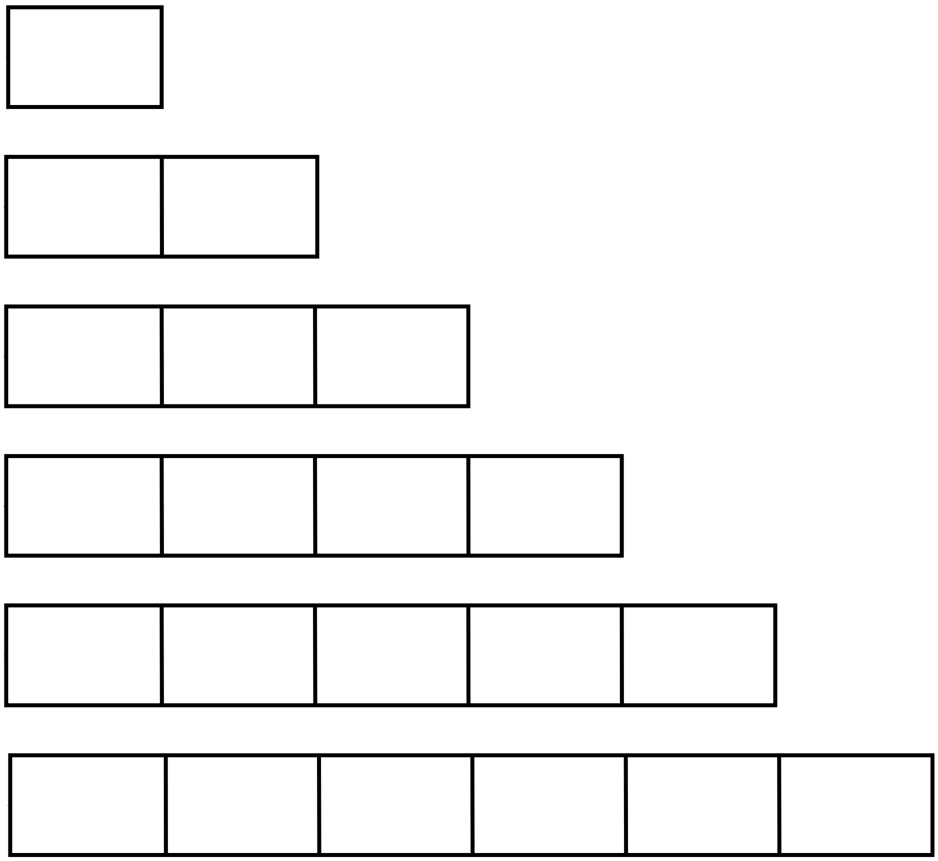
Running CYK

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
```

Example

s → *x* | *y*

x → **( )**

y → **(** *y y* **)**

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
```

Example

s → *a b*

a → **+** | **–** | ε

b → **digit** | *b* **digit**

c → **.** *b*

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

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

<u>Example 1</u>

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

<u>Example 2</u>

x → *a* **X** *a* **Y** *a*
a → ε
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*

If there are multiple rules with *a* on the LHS,

- for each rule of the form *b* → δ, add a → δ
- remove *a* → *b*

Example

a → *b* **X**

| *c b*

| *b*

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*

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