CS 536 Announcements for Monday, March 4, 2024

Last Time

- approaches to parsing
- bottom-up parsing
- CFG transformations
  - removing useless non-terminals — can't derive a seq of tokens; can't be derived from start non-term
  - Chomsky normal form (CNF) → $X \to T$ $X \to ab$ only start can derive ε
- CYK algorithm

Today

- wrap up CYK
- classes of grammars
- top-down parsing

Next Time

- building a predictive parser
- FIRST and FOLLOW sets

Parsing (big picture)

Context-free grammars (CFGs) Given CFG G

- language generation: $G \to w \in L(G)$
- language recognition: given w, is $w \in L(G)$?

Translation

- given $w \in L(G)$, create a parse tree for w
- given $w \in L(G)$, create an AST for w
  → passed on to next phase of our compiler

CYK algorithm

Step 1: get grammar in Chomsky Normal Form (CNF)

Step 2: build all possible parse trees bottom-up

- start with runs of 1 terminal
- connect 1-terminal runs into 2-terminal runs
- connect 1- and 2-terminal runs into 3-terminal runs
- connect 1- and 3- or 2- and 2-terminal runs into 4-runs
- …
- if we can connect entire tree, rooted at start symbol, we've found a valid parse

Pros: able to parse an arbitrary CFG including ambiguous grammars

Cons: $O(n^3)$ time complexity ← too slow!

For special classes of grammars, we can parse in $O(n)$ time

→ eg LL(1) & LALR(1)

Classes of grammars

LL(1)

scan from L to R — leftmost derivation — 1 token lookahead

LALR(1)

look ahead (technical detail of LR parsers) — scan L to R — rightmost derivation (in reverse) — 1 token lookahead

CFG
⊃
LR(1)
⊃
LALR(1)
⊃
LL(1)
⊃
RG
= regular grammar
– langs that can be recognized by DFAs

Both are accepted by parser generators

LALR(1)

- parsed by bottom-up parsers
- harder to understand

Java CUP generates a LALR(1) parser

LL(1)

- parsed by top-down parsers

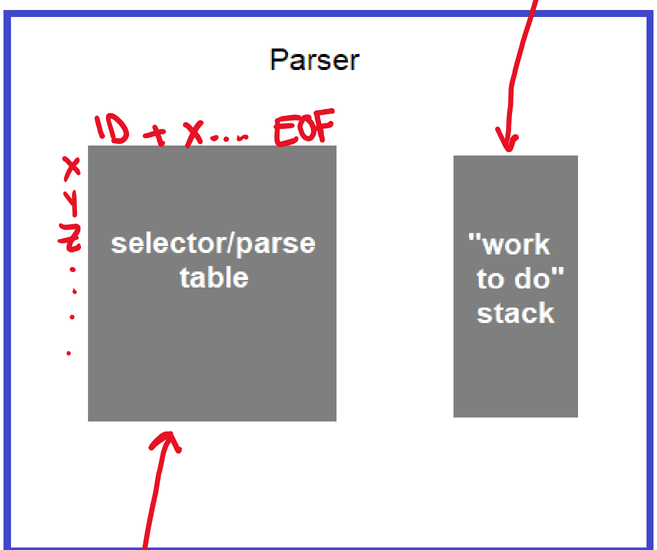
  ↳ or predictive parsers or recursive descent parsers

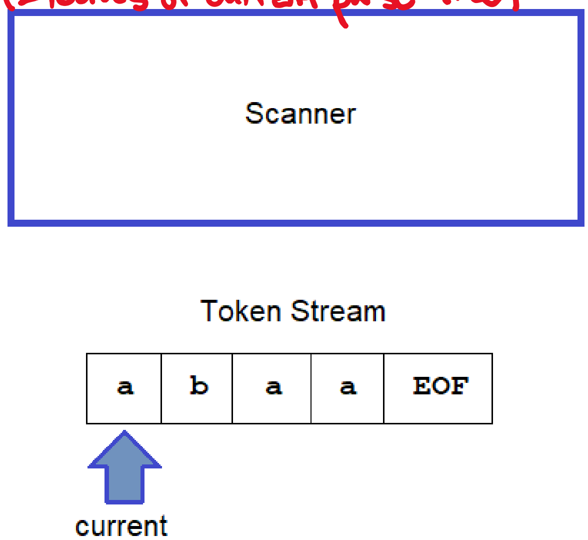
Top-down parsers

- Start at start symbol
- Repeatedly "predict" what production to use

Predictive parser overview

Contains tokens & nonterms (= leaves of current parse tree)





indexed by [nonterms, token]

table entry row X is either
- empty (bad input)
- contains RHS for grammar rule for X (ie X → RHS)

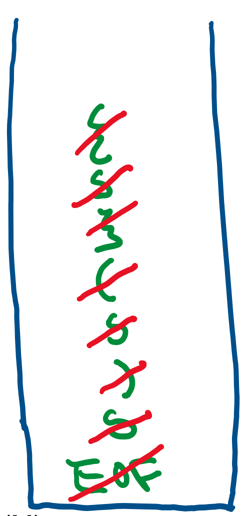
Example

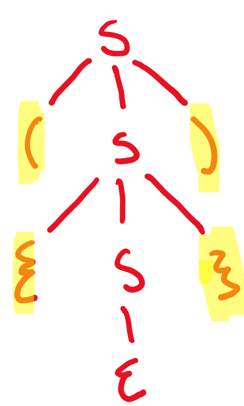
CFG: s → (s) | { s } | ε

Parse table:

	(	)	{	}	EOF
s	(s)	ε	{ s }	ε	ε

Input: ({ }) EOF





Predictive parser algorithm

```
stack.push(EOF)
stack.push(start nonterm)
T = scanner.getToken( )

repeat

    if stack.top is terminal Y
       match Y with T
       pop Y from stack
       T = scanner.getToken()

    if stack.top is nonterminal x
       get table[x, current token T]
       pop x from stack
       push production's RHS (each symbol from R to L)

until one of the following:
    stack is empty
    stack.top is a terminal that does not match T
    stack.top is a nonterm and parse-table entry is empty
```

initial work stack

note: don't push ε

— accept input

reject input

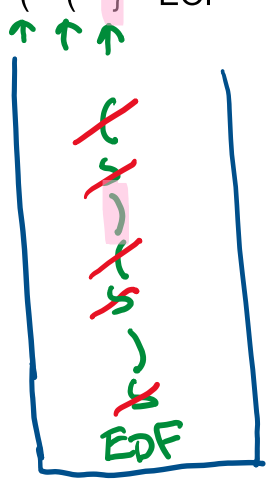
Example

CFG: s → (s) | { s } | ε

Parse table:

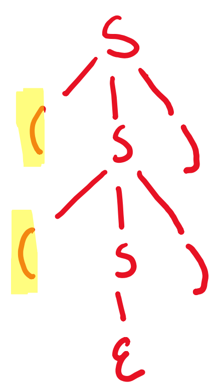
	(	)	{	}	EOF
s	(s)	ε	{s}	ε	ε

Input: ((} EOF



no match

reject input



Consider

CFG: s → (s) | { s } | ( ) | { } | ε

Parse table:

	(	)	{	}	EOF
S					

(s) or ( ) ?

If could look ahead 2 tokens, we could make a good choice

This grammar is not LL(1), but it is LL(2)

Some grammars are not LL(k) for any k

Two issues

1) How do we know if the language is LL(1)?
2) How do we build the selector table?

Answer: If we can build a parse table (selector table) & each entry has (at most) 1 production in it, then the grammar is LL(1)

Converting non-LL(1) grammars to LL(1) grammars

Necessary (but not sufficient conditions) for LL(1) parsing

- free of left recursion – no left-recursive rules
- left-factored – no rules with a common prefix, for any nonterminal

Left recursion

- A grammar G is *recursive* in nonterm X iff $X \Rightarrow^+ \alpha X \beta$
- A grammar G is *left recursive* in nonterm X iff $X \Rightarrow^+ X \beta$
- A grammar G is *immediately left recursive* in X iff $X \Rightarrow X \beta$

Why left-recursion is a problem

Consider: xlist → xlist ID | ID

Current parse tree

xlist

Current token

ID

How to grow tree (top-down)?

xlist → ID or xlist → xlist ID

use if no more IDs after this

use if there are more IDs

Don't know without lookahead

Parse table

	ID	
xlist	xlist ID	

~~xlist~~
ID
~~xlist~~
ID
~~xlist~~
ID
~~xlist~~
EOF

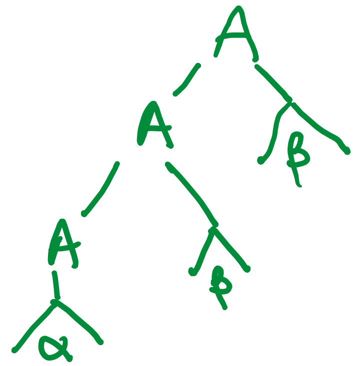
Stack overflow

Removing left-recursion

We can remove immediate left recursion without "changing" the grammar:

<u>Consider</u>: $A \rightarrow A\ \beta$
$\quad\quad\quad\quad\ \mid \alpha$

doesn't start with nonterm A

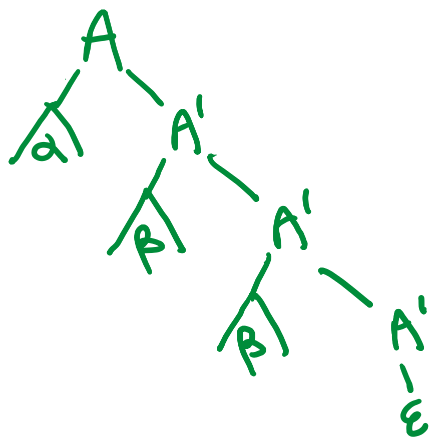


<u>Solution</u>: introduce new nonterminal A'
and new productions:

$A \rightarrow \alpha A'$

$A' \rightarrow \beta A' \mid \varepsilon$

Messes up associativity of parse tree
(we'll fix this when we build the AST)



More generally,

$$A \rightarrow \underline{\alpha_1} \mid \underline{\alpha_2} \mid \ldots \mid \underline{\alpha_n} \mid A\ \underline{\beta_1} \mid A\ \underline{\beta_2} \mid \ldots \mid A\ \underline{\beta_p}$$

transforms to

$$A \rightarrow \underline{\alpha_1} A' \mid \underline{\alpha_2} A' \mid \ldots \mid \underline{\alpha_n} A'$$

$$A' \rightarrow \underline{\beta_1} A' \mid \underline{\beta_2} A' \mid \ldots \mid \underline{\beta_p} A' \mid \boxed{\varepsilon}$$

Grammars that are not left-factored

If a nonterminal has two productions whose right-hand sides have a common prefix, the grammar is not left-factored.

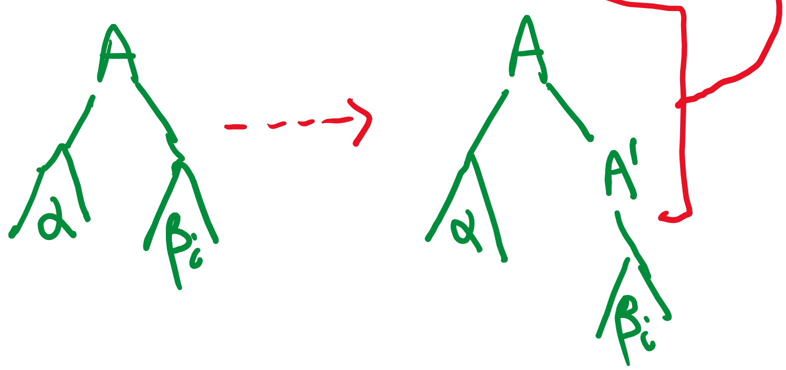
Example: $s \rightarrow (\, s\,) \mid (\,)$

$S \rightarrow (\, s'$
$s' \rightarrow s\,) \mid\,)$

Given: $A \rightarrow \alpha\, \beta_1 \mid \alpha\, \beta_2$
transform it to

$A \rightarrow \alpha\, A'$
$A' \rightarrow \beta_1 \mid \beta_2$

gets collapsed when AST is built



More generally,

$$A \rightarrow \alpha\, \beta_1 \mid \alpha\, \beta_2 \mid \ldots \mid \alpha\, \beta_n \mid \delta_1 \mid \delta_2 \mid \ldots \mid \delta_p$$

transforms to

$A \rightarrow \alpha\, A' \mid \delta_1 \mid \delta_2 \mid \ldots \mid \delta_p$
$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

Combined example

exp → **(** exp **)**
  | exp exp
  | **( )**

exp → (exp) exp' | ( ) exp'
exp' → exp exp' | ε

exp → (exp''
exp'' → exp) exp' |) exp'
exp' → exp exp' | ε