CS 536 Announcements for Monday, March 4, 2024

Last Time

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

- language generation:
- language recognition:

Translation

- given $w \in L(G)$, create
- given $w \in L(G)$, create

CYK algorithm

Step 1: get grammar in Chomsky Normal Form

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

Cons: O(n^3) time complexity

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

Classes of grammars

LL(1)

LALR(1)

Both are accepted by parser generators

LALR(1)

- parsed by bottom-up parsers
- harder to understand

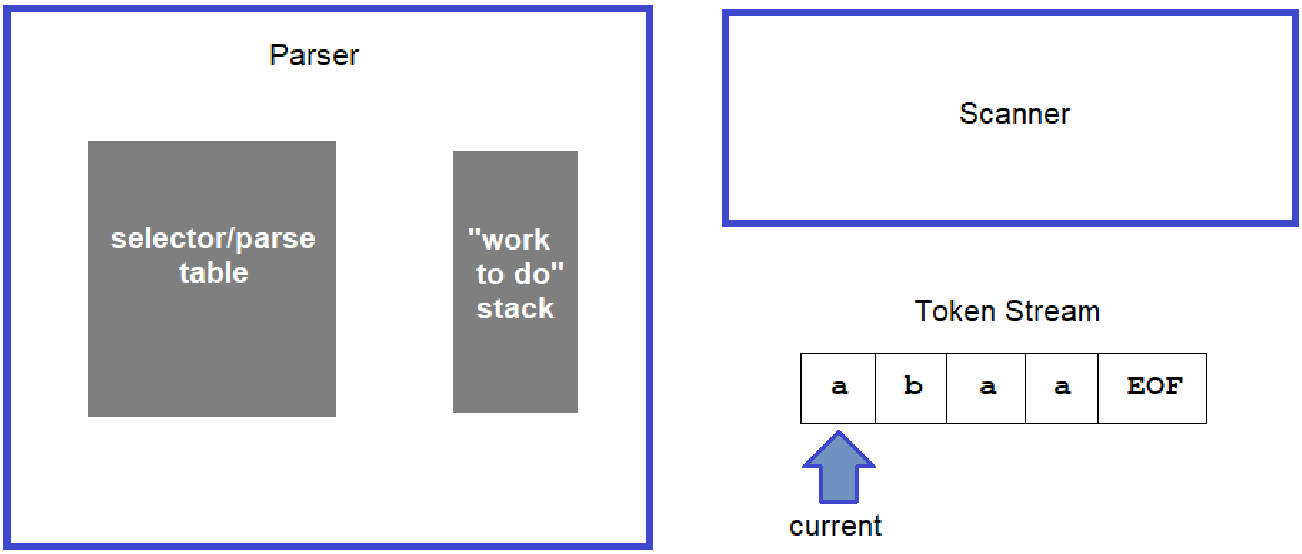
LL(1)

- parsed by top-down parsers

Top-down parsers

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

Predictive parser overview



Example

CFG: s → (s) | { s } | ε

Parse table:

	(	)	{	}	EOF
s					

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
```

Example

CFG: $s \rightarrow (s) \mid \{ s \} \mid \varepsilon$

Parse table:

	(	)	{	}	EOF
s					

Input: ((} EOF

Consider

CFG: $s \rightarrow (s) \mid \{ s \} \mid ( ) \mid \{ \} \mid \varepsilon$

Parse table:

	(	)	{	}	EOF
S					

Two issues

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

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

Removing left-recursion

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

Consider: $A \rightarrow A\ \beta$
$\quad\quad | \ \alpha$

Solution: introduce new nonterminal A'
and new productions:

More generally,

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

transforms to

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 → (s) | ( )

Given: $A \rightarrow \alpha\, \beta_1 \mid \alpha\, \beta_2$

transform it to

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

Combined example

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