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: Given CFG G
  \[ G \rightarrow \alpha \in L(G) \]
- 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 \( w \)

\[ \text{passed on to next phase of our compiler} \]
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

**Cons:** \(O(n^3)\) time complexity \(\leq\) too slow!

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

\[\text{LL(1)} \& \ LALR(1)\]

Classes of grammars

**LL(1)**
- Scan from L to R
- 1 token lookahead
- Leftmost derivation

**LALR(1)**
- Look ahead
- Technical detail of LR parsers
- Rightmost derivation (in reverse)
- Scan L to R

Both are accepted by parser generators

**LALR(1)**
- parsed by bottom-up parsers
- harder to understand

**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

Example

CFG: \[ s \rightarrow ( s ) | \{ s \} | \varepsilon \]

Parse table:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>EOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>(s)</td>
<td>3</td>
<td>3 → s3</td>
</tr>
</tbody>
</table>

Input: ( { } ) EOF
Predictive parser algorithm

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

repeat

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

    if stack.top is nonterminal
        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: \[ s \Rightarrow ( s ) | \{ s \} | \varepsilon \]

Parse table:

<table>
<thead>
<tr>
<th></th>
<th>( )</th>
<th>{ }</th>
<th>EOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>(s)</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Input: ( ( } EOF

```
```
Consider

CFG: \[ s \rightarrow (s) \mid \{s\} \mid (\) \mid () \mid \{\} \mid \epsilon \]

Parse table:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>EOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(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: \( xlist \rightarrow xlist \ ID \mid ID \)

Current parse tree: \( xlist \)

Current token: \( ID \)

How to grow tree (top-down)?

- \( xlist \)
- \( ID \)

Use if no more IDs after this

Use if there are more IDs

Don't know without lookahead
Removing left-recursion

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

Consider: \[ A \rightarrow A \beta \]
\[ | \alpha \]

doesn't start with nonterm A

Solution: introduce new nonterminal \( A' \) and new productions:

\[ A \rightarrow \alpha \ A' \]
\[ A' \rightarrow \beta \ A' \ some \ production \]

messes up associativity of parse tree

(we'll fix this when we build the AST)

More generally,

\[ A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n | A \beta_1 | A \beta_2 | \ldots | A \beta_p \]

transforms to

\[ A \rightarrow \alpha_1 \ A' | \alpha_2 \ A' | \ldots | \alpha_n \ A' \]
\[ A' \rightarrow \beta_1 \ A' | \beta_2 \ A' | \ldots | \beta_p \ A' | \epsilon \]
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: \( S \to ( S ) \mid ( ) \)  

Given: \( A \to \alpha \beta_1 \mid \alpha \beta_2 \)

transform it to

\[
\begin{align*}
A & \to \alpha A' \\
A' & \to \beta_1 \mid \beta_2
\end{align*}
\]

More generally,

\( A \to \alpha \beta_1 \mid \alpha \beta_2 \mid \ldots \mid \alpha \beta_n \mid \delta_1 \mid \delta_2 \mid \ldots \mid \delta_p \)

transforms to

\[
\begin{align*}
A & \to \alpha A' \mid \delta_1 \mid \delta_2 \mid \ldots \mid \delta_p \\
A' & \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n
\end{align*}
\]

Combined example

\[
\begin{align*}
\text{exp} & \to ( \text{exp} ) \\
& \mid \text{exp} \text{exp} \\
& \mid ( )
\end{align*}
\]