CS 536 Announcements for Thursday, March 10, 2022

Last Time
- bottom-up parsing
- CYK algorithm
- Chomsky normal form
- removing useless nonterminals

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: $G \rightarrow L(G)$
- 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 $w$

[Expresseed as 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 < too slow!

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

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

**Classes of grammars**

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

**LALR(1)**
- Look ahead
- 1 token look ahead
- Rightmost derivation (in reverse)
- Scan from 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

JavaCup generates a LALR(1) parser

*Regular grammars* $\rightarrow$ *languages* that can be recognized by DFA
Top-down parsers

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

Predictive parser overview

- The parser is indexed by [nonterm, token].
- Each table entry for row X is either empty (bad input) or contains RHS for grammar rule for X (i.e., \( X \to \text{RHS} \)).
- Contains tokens & nonterms = leaves of current parse tree.

Diagram:
- Scanner
- Token stream: \( a \ b \ a \ a \) EOF
- Current token
- Work to do: Stack
- Selector/parse table
- Parser: able ... EOF
- Indexed by [nonterm, token]
Example

CFG: \[ S \rightarrow (S) | \{S\} | \epsilon \]

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
Predictive parser algorithm

```java
stack.push(EOF)  // initial stack
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  // accept input
    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>|</td>
<td>|</td>
</tr>
</tbody>
</table>

Input:  ( ( } EOF

```
Consider

CFG:

\[
S \rightarrow (S)|\{S|()|\{|\} | \epsilon
\]

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 we could look ahead 2 tokens then 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/selector table & each entry has only 1 production, 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 $X$ if $X \Rightarrow^* \alpha X \beta$
- A grammar $G$ is **left recursive** in nonterm $X$ if $X \Rightarrow^* X \beta$
- A grammar $G$ is **immediately left recursive** in $X$ if $X \Rightarrow X \beta$

**Why left-recursion is a problem**

```
xlist  \Rightarrow  xlist  ID  |  10
```

curr parse tree
curr token: ID

How to grow tree top down?

```
xlist  \Rightarrow  xlist  ID
```

use if no more
10s after this

```
xlist  \Rightarrow  xlist
```

use if there are more 10s

Don't know without lookahead

```
xlist  \Rightarrow  \emptyset
```

stack overflow

Parse table

```
xlist  10
xlist  xlist  10
```

```
Removing left-recursion

Can remove "immediate left-recursion" without "changing" grammar

Consider \( A \rightarrow A \beta \)

\[ \alpha \]

doesn't start with \( A \)

Introduce new non-term \( A' \)

& 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

\[
A \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n \mid A \beta_1 \mid A \beta_2 \mid ... \mid A \beta_p
\]

transform to

\[
A \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid ... \mid \alpha_n A' \\
A' \rightarrow \beta_1 A' \mid \beta_2 A' \mid ... \mid \beta_p A' \mid \varepsilon
\]
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:

Given $A \rightarrow \alpha \beta_1 | \alpha \beta_2$

Transform to $A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 | \beta_2$

More generally:

$A \rightarrow \alpha \beta_1 | \alpha \beta_2 | \ldots | \alpha \beta_n | S_1 S_2 \ldots S_p$

transforms to

$A \rightarrow \alpha A' | S_1 | S_2 | \ldots | S_p$

$A' \rightarrow \beta_1 | \beta_2 | \ldots | \beta_n$

gets collapsed when AST is built.
Combined example

\[
\text{exp} \rightarrow (\text{exp}) \\
| \text{exp} \ \text{exp} \\
| ()
\]

remove immediate left-recursion

\[
expr \rightarrow (expr) \ expr' \ | \ ()\ expr'
expr' \rightarrow expr \ expr' \ | \ \epsilon
\]

left-factor