CS536

MORE Parsing
• CYK
  – Step 1: get a 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 terminal runs
    • …
    • If we can connect the entire tree, rooted at the start symbol, we’ve found a valid parse
Some Interesting properties of CYK

• Very old algorithm
  – Already well known in early 70s

• No problems with ambiguous grammars:
  – Gives a solution for all possible parse tree simultaneously
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 → ,

id ( id , id )
Thinking about Language Design

• Balanced considerations
  – Powerful enough to be useful
  – Simple enough to be parseable

• Syntax need not be complex for complex behaviors
  – Guy Steele’s “Growing a Language”
    https://www.youtube.com/watch?v=_ahvzDzKdB0
Restricting the Grammar

• By restricting our grammars we can
  – Detect ambiguity
  – Build linear-time, $O(n)$ parsers

• LL(1) languages
  – Particularly amenable to parsing
  – Parseable by Predictive (top-down) parsers
    • Sometimes called recursive descent
Top-Down Parsers

• Start at the Start symbol
• “predict” what productions to use
  – Example: if the current token to be parsed is an id, no need to try productions that start with integer literal
  – This might seem simple, but keep in mind multiple levels of productions that have to be used
Predictive Parser Sketch

Parser

Scanner

Col: terminal

Selector table

Row: nonterminal

“Work to do” Stack

Token Stream

a  b  a  a  EOF

current
Algorithm

stack.push(eof)
stack.push(Start non-term)
t = scanner.getToken()
Repeat
  if stack.top is a terminal y
    match y with t
    pop y from the stack
    t = scanner.next_token()
  if stack.top is a nonterminal X
    get table[X,t]
    pop X from the stack
    push production’s RHS (each symbol from Right to Left)
Until one of the following:
  stack is empty accept
  stack.top is a terminal that doesn’t match t
  stack.top is a non-term and parse table entry is empty
  reject
Example

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

```
Stack: ( S ) \varepsilon \{S\} \varepsilon \varepsilon
eof
```

```
current current current current
```

```
{ 
  S
  S
  )
eof
```

```
“Work to do”
Stack
```
Example 2, bad input: You try

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

\[
\begin{array}{cccc}
( & ) & \{ & \} & \text{eof} \\
S & (S) & \epsilon & \{S\} & \epsilon & \epsilon
\end{array}
\]

INPUT

( ( ) } eof
This Parser works great!

• Given a single token we always knew exactly what production it started

\[
S \rightarrow (S) \varepsilon \{S\} \varepsilon \varepsilon
\]

\[
(\quad \quad \{ \quad \quad \text{eof}
\]

\[
S \quad (S) \quad \varepsilon \quad \{S\} \quad \varepsilon \quad \varepsilon
\]
Two Outstanding Issues

1. How do we know if the language is LL(1)
   – Easy to imagine a Grammar where a single token is not enough to select a rule

\[ S \rightarrow (S) \mid \{S\} \mid \varepsilon \mid ( ) \]

1. How do we build the selector table?
   – It turns out that there is one answer to both:

If our selector table has 1 production per cell, then grammar is LL(1)
LL(1) Grammar Transformations

• Necessary (but not sufficient conditions) for LL(1) Parsing:
  – Free of left recursion
    • No nonterminal loops for a production
    • Why? Need to look past list to know when to cap it
  – Left factored
    • No rules with common prefix
    • Why? We’d need to look past the prefix to pick rule
Left-Recursion

• Recall, a grammar such that $X \Rightarrow^* X \alpha$ is left recursive

• A grammar is immediately left recursive if this can happen in one step:

$$A \rightarrow A \alpha | \beta$$

Fortunately, it’s always possible to change the grammar to remove left-recursion without changing the language it recognizes.
Why Left Recursion is a Problem (Blackbox View)

CFG snippet:  \[ XList \rightarrow XList \ x \mid x \]

Current parse tree: \[ XList \]

Current token: \[ x \]

How should we grow the tree top-down?

(OR)

Correct if there are no more \( x \)s

Correct if there are more \( x \)s

We don’t know which without more lookahead
Why Left Recursion is a Problem (Whitebox View)

CFG snippet: $XList \rightarrow XList \ x \mid x$

Current parse tree: $XList$

Parse table: $XList$ $XList \ x$ $\varepsilon$

Current token: $x$

(Stack overflow)

Stack:

- $XList$
- $x$
- $x$
- $x$
- $x$
- $eof$

Current:

- $x$
Removing Left-Recursion

(for a single immediately left-recursive rule)

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

\[\rightarrow\]

\[ A \rightarrow \beta A' \]

\[ A' \rightarrow \alpha A' | \varepsilon \]

Where \( \beta \) does not begin with \( A \)
Example

\[ A \rightarrow A\alpha \mid \beta \quad \text{A} \rightarrow \beta A' \]

\[ A' \rightarrow \alpha A' \mid \varepsilon \]

\[ \text{Exp} \rightarrow \text{Exp} - \text{Factor} \quad \text{Exp} \rightarrow \text{Factor Exp'} \]

\[ \mid \text{Factor} \quad \text{Exp'} \rightarrow - \text{Factor Exp'} \]

\[ \mid \text{intlit} \mid (\text{Exp}) \quad \mid \varepsilon \]

\[ \text{Factor} \rightarrow \text{intlit} \mid (\text{Exp}) \]
Let’s check in on the Parse Tree...

Exp \rightarrow Exp - Factor  \\
|   Factor  \\
Factor \rightarrow intlit | ( Exp )

Exp \rightarrow Factor Exp'    \\
Exp' \rightarrow - Factor Exp'  \\
| \varepsilon  \\
Factor \rightarrow intlit | ( Exp )
... We’ll fix that later
General Rule for Removing Immediate Left-Recursion

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

\[ A \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_n A' \]

\[ A' \rightarrow \beta_1 A' | \beta_2 A' | \ldots | \beta_m A' | \varepsilon \]
Left Factored Grammars

• If a nonterminal has two productions whose RHS has a common prefix it is not left factored and not LL(1)

\[ Exp \rightarrow ( \text{Exp} ) \mid ( ) \]

Not left factored
Left Factoring

• Given productions of the form

\[ A \rightarrow \alpha \beta_1 | \alpha \beta_2 \]

\[ A \rightarrow \alpha A' \]

\[ A' \rightarrow \beta_1 | \beta_2 \]
Combined Example

\[
\begin{align*}
\text{Exp} & \rightarrow ( \text{Exp} ) \mid \text{Exp} \text{Exp} \mid ( ) \\
\text{Exp} & \rightarrow ( \text{Exp} ) \text{Exp}' \mid ( ) \text{Exp}' \\
\text{Exp}' & \rightarrow \text{Exp} \text{Exp}' \mid \varepsilon \\
\text{Exp}' & \rightarrow ( \text{Exp}'') \\
\text{Exp}'' & \rightarrow \text{Exp} ) \text{Exp}' \mid ) \text{Exp}' \\
\text{Exp}' & \rightarrow \text{expr} \text{expr}' \mid \varepsilon
\end{align*}
\]
Where are we at?

• We’ve set ourselves up for success in building the selection table
  – Two things that prevent a grammar from being LL(1) were identified and avoided
    • Not Left-Factored grammars
    • Left-recursive grammars
  – Next time
    • Build two data structures that combine to yield a selector table:
      – FIRST set
      – FOLLOW set