Top-down parsing
Last Time

- **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

- **Scanner**
- **Token Stream**: a b a a EOF
- **Current**
- **Selector table**: Row: nonterminal, Col: terminal
- **“Work to do” Stack**
- **Parser**
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
  stack.top is a terminal that doesn’t match t
  stack.top is a non-term and parse table entry is empty
accept
reject
Example

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

```
S
(S)  ε  {S}  ε  ε
```

```
stack
{S}
S
S
)
```

```
"Work to do"
```

```
Stack
```

```
(eof)
```

```
(current)
```

```
(current)
```

```
(current)
```

```
(current)
```

```
 eof
```

```
 eof
```

```
 eof
```

```
 eof
```

```
 eof
```

```
 eof
```

```
 eof
```

```
 eof
```

```
 eof
```

```
 eof
```

```
 eof
```
Example 2, bad input: You try

$$S \rightarrow (S) | \{ S \} | \varepsilon$$

```
(   )   {   }   eof
S  (S)  ε  {S}  ε  ε
```

INPUT

```
( ( ) )  eof
```
This Parser works great!

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

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

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

$$S \rightarrow (S) \mid \{S\} \mid \varepsilon \mid ()$$

2. 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 $X \Rightarrow + \overleftarrow{X} \alpha$ is left recursive

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

\[ A \rightarrow A \alpha \mid \beta \]

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: \( XList \rightarrow XList \ x \ | \ x \)

Current parse tree: \( XList \)  \hspace{1cm} \text{Current token: } x

How should we grow the tree top-down?

Correct if there are no more xs
Correct if there are more xs

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: \( XList \begin{array}{c|c} XList \, x & \varepsilon \end{array} \)

Current token: \( x \)

(Stack overflow)
Removing Left-Recursion

(for a single immediately left-recursive rule)

\[ A \rightarrow A \alpha \mid \beta \]

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

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

\[ A \rightarrow A\alpha \mid \beta \]

\[ A \rightarrow \beta A' \]
\[ A' \rightarrow \alpha A' \mid \epsilon \]

\[ Exp \rightarrow Exp - Factor \mid Factor \]
\[ Factor \rightarrow \text{intlit} \mid (Exp) \]

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

\[
\begin{align*}
\text{Exp} & \rightarrow \text{Exp} - \text{Factor} \\
& \quad | \quad \text{Factor} \\
\text{Factor} & \rightarrow \text{intlit} | (\, \text{Exp} \,)
\end{align*}
\]

\[
\begin{align*}
\text{Exp} & \rightarrow \text{Factor} \, \text{Exp'} \\
\text{Exp'} & \rightarrow - \text{Factor} \, \text{Exp'} \\
& \quad | \quad \epsilon \\
\text{Factor} & \rightarrow \text{intlit} | (\, \text{Exp} \,)
\end{align*}
\]
... We’ll fix that later
General Rule for Removing Immediate Left-Recursion

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

\[ A \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \\
A' \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A' \mid \varepsilon \]
Left Factored Grammars

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

\[ Exp \to ( Exp ) \mid ( ) \]

Not left factored
Left Factoring

- Given productions of the form

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

\[ A \rightarrow \alpha A' \]

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

\[ Exp \rightarrow (Exp) \mid Exp\ Exp \mid () \]

Remove
Immediate left-recursion

\[ Exp \rightarrow (Exp)\ Exp' \mid ()\ Exp' \]
\[ Exp' \rightarrow Exp\ Exp' \mid \varepsilon \]

Left-factoring

\[ Exp' \rightarrow Exp\ Exp' \mid \varepsilon \]

25
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