Lecture 8:

- Quiz 2
- $LL(k)$ Grammars and Parsing

Announcements:

- HW-1 due
- HW-2 out (will cover basics Wed.)

$LL(k)$ Parsing

We will implement an $LL(k)$ parser

- read from left-to-right (1st L), performing a left-most derivation (2nd L)
- parses top down (parse tree built root down)
- at most k look ahead symbols (more later)
- ... some work is usually required to ensure a grammar is $LL(k)$!

Consider this statement rule: ... let <var> be a VAR literal

```
<stmt> ::= <var> '=' <expr>
```

Assuming the parser *knows* <stmt> is to be applied ...

(1) calls lexer's `nextToken()` and checks that it is a `VAR` token

(2) calls lexer's `nextToken()` and checks that it is an `ASSIGN` token

(3) and so on until it finishes the <stmt> rule

Produces an error if it finds a token it isn't expecting

Tips for $LL(k)$ Grammars

Watch out for left recursion!

Example: $(1)\ E \rightarrow N \quad (2)\ E \rightarrow E + N$ ... assume N is an integer literal

To parse "5 + 4 + 3", need to determine which E rule to apply ...

Q: How far do we need to look ahead in the string to pick the rule?

- we have to go to the end of the expression ...
  (1) Looking at 5, we don't know whether to apply 1 or 2
  (2) To pick 2, need to know if the string *ends* in "+ N"
  (3) Means we have to read the entire string to know which rule to apply

If the string is longer than k, then we are stuck!

- This grammar is *not* $LL(k)$ since has no fixed size k

Tips for $LL(k)$ Grammars

One solution

$E \rightarrow N + E \mid N$

Q: How many look aheads needed? ... 2 (see "left factoring")

General approach to rewriting left recursion to be in $LL(k)$ ...

$E \rightarrow N\ E'$

$E' \rightarrow +\ N\ E' \mid \varepsilon$

Q: how far do we need to look ahead for "5 + 4 + 3" now?

- just 1 token ... this is now an $LL(1)$ grammar

Tips for $LL(k)$ Grammars

A grammar can also have indirect left recursion

$S \rightarrow T$ a | a

$T \rightarrow S$ b | b

- allows derivations: $S \Rightarrow T$ a $\Rightarrow S$ b a
- having strings of the form: a, ba, aba, baba, ababa, ...

Example rewriting for this grammar

By replacing RHS of T in S, we get:

$S \rightarrow S$ b a | b a | a

Now we can remove the direct left recursion ...

$S \rightarrow$ a S' | ba S'

$S' \rightarrow$ ba S' | ε

Tips for $LL(k)$ Grammars

Watch out for grammars that are *not* left-factored! ... common prefixes

Example: $E \rightarrow$ `if` B `then` S $E \rightarrow$ `if` B `then` S `else` S

- both E rules have a **common prefix**
- this means more look-ahead tokens than needed
- here, since B and S can be any length strings, there is no fixed k!

To left-factor: Given rule $S \rightarrow X_1 \ldots X_n \, \alpha$ $S \rightarrow X_1 \ldots X_n \, \beta$

Rewrite to: $S \rightarrow X_1 \ldots X_n \, S'$ $S' \rightarrow \alpha \mid \beta$

Example: After left factoring ... now it is $LL(1)$!

$E \rightarrow$ `if` B `then` S E'

$E' \rightarrow$ `else` S | ε

Tips for $LL(k)$ Grammars

Watch out for <u>ambiguous grammars</u>! ... problematic for $LL(k)$ parsers

Example: $E \rightarrow \texttt{ID} \mid P \qquad P \rightarrow \texttt{[ID]} \mid \texttt{ID}$

... multiple (left-most) ways to generate an `ID`

$E \Rightarrow \texttt{ID}$

$E \Rightarrow P \Rightarrow \texttt{ID}$

These produce different parse trees!

- implying potentially different language interpretations (more later)
- also unclear how to choose between these (parsers will only do one)