LL Parsers and LL(k) Grammars

Spring 2011
Instructor: Mike Wilson
Parsers and Recognizers

Both are concerned with the same job:

Is this string a member of the language of the grammar?

Recognizer
- Does purely membership test.
- No parse tree is produced, so ambiguities don't matter.

Parser
- Produces an actual parse tree
LL(k) Grammars

- **LL(k)**
  - Produces a derivation by left-to-right symbol scan
  - Produces a leftmost derivation
  - Uses no more than \( k \) symbols of lookahead

- **LL(k) parsers**
  - Produces a parser tree corresponding to a leftmost derivation
  - *Predictive* parsers: work by predicting productions based on symbol lookahead
LL(k) Parsing Techniques

- **Recursive Descent**
  - A parse procedure is written for every nonterminal
  - Each parse procedure knows how to handle all productions from that nonterminal

- **Table-Driven**
  - LL parsing can be automated in a table-driven, generalized LL Parse procedure
  - Most automatically generated LL-parsers use tables
LL(1) Prediction

- LL parsers need to predict each production based on symbols in lookahead
  - Use Derives-λ, FIRST, FOLLOW, and PREDICT

- Most LL parsers are limited to LL(1)
  - Potential for exponential growth in parser size
    Parser size is $O(|\Sigma|^k)$

- PREDICT sets can be formed mechanically
Derives-\(\lambda\), FIRST, FOLLOW, PREDICT

- **Review:**

- **Derives-\(\lambda\)**
  - Set of all nonterminals which can derive \(\lambda\)

- **FIRST(\(N\))**
  - All terminals which can begin a string derived from nonterminal \(N\)

- **FOLLOW(\(N\))**
  - All terminals which can follow nonterminal \(N\) in some sentential form

- **PREDICT(P)**
  - All terminals which predict using production \(P\) when deriving from LHS of \(P\)
Non-LL(1) Grammars

Grammars are not LL(1) when they:
- Have overlapping PREDICT sets for the same nonterminal
- Can never advance the input

This happens when:
- Some productions have common prefixes
- Some productions use left-recursion
- The grammar is ambiguous
Non-LL(1) Grammars

- A Language may be LL(1) while the grammar is not
  - We can attempt to fix the grammar

- We can remove common prefixes

- We can remove left-recursion
Factoring Common Prefixes

- Given productions of the form
  \[ A \rightarrow BC \]
  \[ A \rightarrow BD \]

- We can "factor" the common prefix into a single production
  - Add additional productions for the tails
    - \[ A \rightarrow BX \]
    - \[ X \rightarrow C \]
    - \[ X \rightarrow D \]

- Algorithm in text applies this repeatedly to the longest common prefix until none remain
Removing Left-Recursion

- Given productions of the form
  \[ A \rightarrow Aa \]
  \[ \mid b \]

- We can remove the left-recursion by transforming to right-recursion

  \[ A \rightarrow XY \]
  \[ X \rightarrow b \]
  \[ Y \rightarrow aY \]
  \[ \mid \lambda \]

- Algorithm in text applies this repeatedly until none remains
Can everything be LL(1)?

- No. Classic example is if-then-else
  - Reduced to the Dangling Bracket Language, DBL
    \{[i]^k \mid i \geq k \geq 0\}
  - This *language* can never be made LL(k)
    - Can remove common prefixes and left-recursion
    - Resulting grammar is ambiguous

- Our infix arithmetic with correct precedence has a related problem
  - No LL grammar exists which provides left-to-right order of evaluation
Parsing non-LL languages

- We can still parse non-LL languages
- LR(k) parsing is strictly more powerful: can handle these grammars naturally
- Alternative: Modify our resulting LL parser to "break ties" by precedence

```
S → A $ 
A → [ A CL | \lambda 
CL → ] | \lambda 

CL() 
if peek() = ']' 
  match(']') 
else if peek() = $ 
  return
```
Useful Properties of LL(1) Grammars

- Can always get correct leftmost derivation
  - Recall that obtaining derivations for some grammars is undecideable

- All LL(1) grammars are unambiguous

- Parsing is linear time and space complexity in input

- Error handling is easy!
Error Handling

- LL parsing is *predictive*: we always know what symbols to expect.

- We can recognize errors when we don't see expected symbol.

- Easy to give informative error messages:
  - Line 15, character 12: expected constant or variable, got "<"
Error Handling

- Good parsers give as much information as possible
  - What about subsequent errors in the input?

- Error Recovery
  - Return our parser and input to a functional state

- Easiest method: Panic Mode
  - Skip tokens until we can resynchronize parser and input
Error Handling

- Classical method

  - Add a termset to all recursive calls
  
  - On error, procedure can skip tokens until lookahead matches symbol from termset
  
  - Caller claims it can continue parsing from that symbol
Error Handling

- Classical method
  - Add a termset to all recursive calls
  - On error, procedure can skip tokens until lookahead matches symbol from termset
  - Caller claims it can continue parsing from that symbol