CS412/413

Introduction to
Compilers and Translators
Spring '00

Lecture 5: Bottom-up parsing

Outline

• More tips for LL(1) grammars
• Bottom-up parsing
• LR(0) parser construction

Administrivia

• Programming Assignment 1 due next class (Friday)
  – should be well under way -- leave time for testing, documentation
  – do not need to construct DFA!
• All group assignments should have settled out
• Homework 2 due next Friday
• Reading: finish Chapter 3 of Appel

Grammars

• Have been using grammar for language of “sums with parentheses” $(1+(3+4))+5$
• Simple grammar w/ left associativity:
  \[
  S \rightarrow S + E \mid E \\
  E \rightarrow \text{number} \mid (S)
  \]
• LL(1) grammar for same language:
  \[
  S \rightarrow ES' \\
  S' \rightarrow \varepsilon \mid +S \\
  E \rightarrow \text{number} \mid (S)
  \]

Review

• Can make recursive descent parsers for LL(1) grammars

  "Natural" language grammar
  \[
  S \rightarrow ES' \\
  S' \rightarrow \varepsilon \mid +S \\
  E \rightarrow \text{number} \mid (S)
  \]

  How to perform this step?

  LL(1) grammar
  \[
  S \rightarrow E + S \\
  E \rightarrow \text{number} \mid (S)
  \]

  Predictive parse table
  \[
  S \rightarrow E + S \\
  E \rightarrow \text{number} \mid (S)
  \]

  Recursive-descent parser

  Recursive-descent parser w/ AST generation

Left vs. Right Recursion

Right recursion : right-associative

\[
S \rightarrow ES' \\
S' \rightarrow \varepsilon \mid +S \\
E \rightarrow \text{number} \mid (S)
\]

Left recursion : left-associative

\[
S \rightarrow S + E \\
S \rightarrow E
\]
Left-recursive vs Right-recursive

- Left-recursive grammars don’t work with top-down parsing: arbitrary amount of lookahead needed

<table>
<thead>
<tr>
<th>derived string</th>
<th>lookahead</th>
<th>read/unread</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ S \rightarrow S + E ]</td>
<td>1</td>
<td>1 + 2 + 3 + 4</td>
</tr>
<tr>
<td>[ S + E \rightarrow S + E + E ]</td>
<td>1</td>
<td>1 + 2 + 3 + 4</td>
</tr>
<tr>
<td>[ S + E \rightarrow E ]</td>
<td>1</td>
<td>1 + 2 + 3 + 4</td>
</tr>
<tr>
<td>[ E \rightarrow + E + E ]</td>
<td>1</td>
<td>1 + 2 + 3 + 4</td>
</tr>
<tr>
<td>[ 1 + E + E \rightarrow 1 + 2 + 3 + 4 ]</td>
<td>2</td>
<td>1 + 2 + 3 + 4</td>
</tr>
<tr>
<td>[ 1 + 2 + E + E \rightarrow 1 + 2 + 3 + 4 ]</td>
<td>3</td>
<td>1 + 2 + 3 + 4</td>
</tr>
<tr>
<td>[ 1 + 2 + 3 + 4 \rightarrow 1 + 2 + 3 + 4 ]</td>
<td>4</td>
<td>1 + 2 + 3 + 4</td>
</tr>
</tbody>
</table>

How to create an LL(1) grammar

- Write a right-recursive grammar

\[ S \rightarrow E + S \]
\[ S \rightarrow E \]

- Left-factor common prefixes, place suffix in new non-terminal

\[ S \rightarrow ES' \]
\[ S' \rightarrow \epsilon \]
\[ S' \rightarrow + S \]

EBNF

- Extended Backus-Naur Form: allows some regular expression syntax on RHS
  - \( ^{*} \), \(+\), \((\)\) operators (Iota spec: \( ^{?} \))
  - BNF: operator at top level

\[ \begin{align*}
S & \rightarrow ES' \\
S' & \rightarrow \epsilon | + S \\
S & \rightarrow E (+ E')
\end{align*} \]

- EBNF version: no position on + associativity

Top-down parsing EBNF

- Recursive-descent code can directly implement the EBNF grammar:

\[ S \rightarrow E (+ E') \]

\[ \begin{align*}
void parse_S () { & // parses sequence of E + E + E ... \\
parse_E (); & } \\
while (true) { & switch (token) { \\
case '+': & token = input.read(); parse_E (); \\
break; & } \\
case ')': & case EOF: return; \\
default: & throw new ParseError(); \\
} \\
} \\
\end{align*} \]

Building a left-associative AST

Expr parse_S () {
  Expr result = parse_E ();
  while (true) {
    switch (token) {
      case '+':
        token = input.read ();
        result = new Add(result, parse_E ());
        break;
      case ')':
        case EOF: return result;
        break;
      default:
        throw new ParseError();
    }
  }
}

Summary

- Now have complete recipe for building a parser
Bottom-up parsing

- A more powerful parsing technology
- LR grammars -- more expressive than LL
  - can handle left-recursive grammars, virtually all programming languages
  - More natural expression of programming language syntax
- Shift-reduce parsers
  - automatic parser generators (e.g. yacc, CUP)
  - detect errors as soon as possible
  - allows better error recovery

Shift-reduce parsers

Top-down parsing

S → S + E | E
E → number | (S)

Bottom-up parsing

S → S + E | E
E → number | (S)

Right-most derivation -- backward
- Start with the tokens
- End with the start symbol

(1+2+(3+4))+5 ← (1+2+(3+4))+5
← (E+2+(3+4))+5
← (S+2+(3+4))+5
← (S+(3+4))+5
← (S+(E+4))+5 ← (S+(S+4))+5
← (S+(S+E))+5 ← (S+(S))+5 ← (S+E)+5 ← S

Progress of bottom-up parsing

S → S + E | E
E → number | (S)

Top-down vs. Bottom-up

Bottom-up: Don’t need to figure out as much of the parse tree for a given amount of input

Advantage of bottom-up parsing: can select productions based on more information
Shift-reduce parsing

• Parsing is a sequence of shift and reduce operations
• Parser state is a stack of terminals and non-terminals (grows to the right)
• Unconsumed input is a string of terminals
• Current derivation step is always stack+input

<table>
<thead>
<tr>
<th>Derivation step</th>
<th>stack</th>
<th>unconsumed input</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1+2+(3+4))+5</td>
<td>(1+2+(3+4))+5</td>
<td></td>
</tr>
<tr>
<td>(E+2+(3+4))+5</td>
<td>(E+2+(3+4))+5</td>
<td></td>
</tr>
<tr>
<td>(S+2+(3+4))+5</td>
<td>(S+2+(3+4))+5</td>
<td></td>
</tr>
<tr>
<td>(S+E+(3+4))+5</td>
<td>(S+E+(3+4))+5</td>
<td></td>
</tr>
</tbody>
</table>

Shift-reduce parsing

• Parsing is a sequence of shifts and reduces
• Shift -- move look-ahead token to stack
  - shift
• Reduce -- Replace symbols γ in top of stack with non-terminal symbol X, corresponding to production X → γ (pop γ, push X)
  - reduce S → S+E

Problem

• How do we know which action to take -- whether to shift or reduce, and which production?
  - Sometimes can reduce but shouldn’t -- e.g., X → e can always be reduced
  - Sometimes can reduce in different ways

Action Selection Problem

• Given stack σ and look-ahead symbol b, should we
  – shift b onto the stack (making it σb)
  – reduce some production X → γ assuming that stack has the form αγ (making it αX)
• If stack has form αγ, should apply reduction X → γ depending on what stack prefix α is -- but α is different for different possible reductions, since γ’s have different length.

Parser States

• Goal: know what reductions are legal at any given point
  • Idea: summarize all possible stack prefixes α as a parser state
  • Parser state is defined by a DFA that reads in the stack α
  • Accept states of DFA: unique reduction!
  • Summarizing discards information
    – affects what grammars parser handles
    – affects size of DFA (number of states)
**LR(0) parser**

- Left-to-right scanning, Right-most derivation, "zero" look-ahead characters
- Too weak to handle most language grammars (including this one)
- But will help us understand how to build better parsers

**An LR(0) grammar: non-empty lists**

\[
S \rightarrow (L) \\
S \rightarrow id \\
L \rightarrow S \\
L \rightarrow L , S \\
x (x,y) (x, (y,z), w) \\
(((x))) (x, (y, (z, w)))
\]

**LR(0) states**

- A state is a set of items
- An LR(o) item is a production from the language with a separator ",," somewhere in the RHS of the production
- Stuff before ",," already on stack (beginnings of possible \( \gamma \)'s to be reduced)
- Stuff after ",," : what we might see next
- The prefixes \( \alpha \) represented by state itself

**Start State & Closure**

- First step: augment grammar with prod'n \( S' \rightarrow S \) $
- Start state of DFA: empty stack = \( S' \rightarrow . \) S $
- DFA start state closure

\[
S' \rightarrow . S $ \\
S \rightarrow . (L) \\
S \rightarrow . id \\
S \rightarrow (L) \mid id \\
S \rightarrow (L), S \\
S \rightarrow . (L) \mid id \\
S \rightarrow (L), S
\]

**Applying symbols**

- In new state, include all items that have appropriate input symbol just after dot, and advance dot in those items (and take closure.)

**Applying reduce actions**

- Pop RHS from stack, replace with LHS X \( (X \rightarrow \gamma) \), rerun DFA (e.g. (x))
### Bottom-up parsing

- Grammars can be parsed bottom-up using a DFA + stack
- State construction converts grammar into states that capture information needed to know what action to take
- **Next time:** shift-reduce parsing tables
  SLR, LR(1) parsers, automatic parser generators