CS412/413

Introduction to Compilers
Radu Rugina

Lecture 6: Top-Down Parsing
1 Feb 02

Outline

- More on writing CFGs
- Top-down parsing
- LL(1) grammars
- Transforming a grammar into LL form
- Recursive-descent parsing

Where We Are

Review of CFGs

- Context-free grammars can describe
  programming-language syntax
- Power of CFG needed to handle
  common PL constructs (e.g., parens)
- String is in language of a grammar if
  derivation from start symbol to string
- Ambiguous grammars a problem

if-then-else

- How to write a grammar for if stmts?
  \[ S \rightarrow \text{if } (E) \text{ S} \]
  \[ S \rightarrow \text{if } (E) \text{ S else } S \]
  \[ S \rightarrow \text{other} \]

Is this grammar ok?

No—Ambiguous!

- How to parse?
  \[ S \rightarrow \text{if } (E) \text{ S} \]
  \[ S \rightarrow \text{if } (E) \text{ S else } S \]
  \[ S \rightarrow \text{other} \]

Which "if" is the "else" attached to?
Grammar for Closest-if Rule

- Want to rule out if (E) if (E) S else S
- Impose that unmatched "if" statements occur only on the "else" clauses

\[
\begin{align*}
\text{statement} & \rightarrow \text{matched} | \text{unmatched} \\
\text{matched} & \rightarrow \text{if} (E) \text{ matched} \ | \text{else matched} \\
\text{other} & \rightarrow \text{if} (E) \text{ statement} \\
\text{unmatched} & \rightarrow \text{if} (E) \text{ matched else unmatched}
\end{align*}
\]

Top-down Parsing

- Grammars for top-down parsing
- Implementing a top-down parser (recursive descent parser)

Parsing Top-down

\[
\begin{align*}
S & \rightarrow E + S \ | \ E \\
E & \rightarrow \text{num} \ | (S)
\end{align*}
\]

Goal: construct a leftmost derivation of string while reading in token stream

<table>
<thead>
<tr>
<th>Partial derived string</th>
<th>Lookahead</th>
<th>Parse out</th>
<th>Unparsed part</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>()</td>
<td>(1+2(3+4))H5</td>
<td></td>
</tr>
<tr>
<td>( \rightarrow E+S )</td>
<td>(1+2(3+4))H5</td>
<td>1</td>
<td>(1+2(3+4))H5</td>
</tr>
<tr>
<td>( \rightarrow (E+S)+S )</td>
<td>1</td>
<td>2</td>
<td>(1+2(3+4))H5</td>
</tr>
<tr>
<td>( \rightarrow (1+2+E+S) )</td>
<td>2</td>
<td>(1+2(3+4))H5</td>
<td></td>
</tr>
<tr>
<td>( \rightarrow (1+2+S)  )</td>
<td>2</td>
<td>(1+2(3+4))H5</td>
<td></td>
</tr>
<tr>
<td>( \rightarrow (1+2+S)H5 )</td>
<td>(1+2(3+4))H5</td>
<td>1</td>
<td>(1+2(3+4))H5</td>
</tr>
<tr>
<td>( \rightarrow (1+2+E+S)  )</td>
<td>(1+2(3+4))H5</td>
<td>2</td>
<td>(1+2(3+4))H5</td>
</tr>
<tr>
<td>( \rightarrow (1+2+2+S) )</td>
<td>(1+2(3+4))H5</td>
<td>2</td>
<td>(1+2(3+4))H5</td>
</tr>
<tr>
<td>( \rightarrow (1+2+S)H5 )</td>
<td>(1+2(3+4))H5</td>
<td>3</td>
<td>(1+2(3+4))H5</td>
</tr>
<tr>
<td>( \rightarrow (1+2+S)H5 )</td>
<td>(1+2(3+4))H5</td>
<td>3</td>
<td>(1+2(3+4))H5</td>
</tr>
</tbody>
</table>

Problem

\[
\begin{align*}
S & \rightarrow E + S \ | \ E \\
E & \rightarrow \text{num} \ | (S)
\end{align*}
\]

- Want to decide which production to apply based on next symbol

1. \( S \rightarrow E \rightarrow (S) \rightarrow (E) \rightarrow (1) \)
2. \( S \rightarrow E + S \rightarrow (E) + S \rightarrow (1) + E \rightarrow (1) + 2 \)

Why is this hard?

Grammar is Problem

- This grammar cannot be parsed top-down with only a single look-ahead symbol
- Not LL(1) = Left-to-right-scanning, Left-most derivation, 1 look-ahead symbol
- Is it LL(k) for some \( k \)?
- Can rewrite grammar to allow top-down parsing; create LL(1) grammar for same language

Making a grammar LL(1)

\[
\begin{align*}
S & \rightarrow E + S \\
E & \rightarrow \text{num} \\
S & \rightarrow (S)
\end{align*}
\]

- Problem: can't decide which S production to apply until we see symbol after first expression
- Left-factoring: Factor common S prefix, add new non-terminal S' at decision point. S' derives \((+E)\)
- Also: convert left-recursion to right-recursion
Parsing with new grammar

\[
S \rightarrow ES' \\
S' \rightarrow \epsilon | + S \\
E \rightarrow \text{num} \ (S)
\]

Predictive Parsing

- LL(1) grammar:
  - for a given non-terminal, the look-ahead symbol uniquely determines the production to apply
  - top-down parsing = predictive parsing
- Driven by predictive parsing table of non-terminals \times terminals \rightarrow productions

Using Table

\[
\begin{array}{c|c|c}
S & ES' & + S \\
\hline
S & (1+2+3+4)+S & 1 \\
E & \text{num} & 3 \\
\end{array}
\]

How to Implement?

- Table can be converted easily into a recursive-descent parser

Recursive-Descent Parser

```c
void parse_S() {
    bokahead token
    switch (token) {
        case '+': parse_E(); parse_S(); return;
        case '(': parse_E(); parse_S(); return;
        default: throw new ParseError();
    }
}
```

Recursive-Descent Parser

```c
void parse_S() {
    bokahead token
    switch (token) {
        case '+': token = input.read(); parse_S(); return;
        case '(': parse_E(); return;
        case EOP: return;
        default: throw new ParseError();
    }
}
```
Recursive-Descent Parser

```c
void parse_E()
{
    switch (token) {
    case number: token = input.read(); return;
    case ":" token = input.read(); parse_S();
        if (token == ")") throw new ParseError();
        token = input.read(); return;
    default: throw new ParseError();
    }
}
```

<table>
<thead>
<tr>
<th>$S'$</th>
<th>$S$</th>
<th>$E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rightarrow$</td>
<td>$+$</td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
<td>$\rightarrow$</td>
</tr>
</tbody>
</table>

Call Tree = Parse Tree

```
(1 + 2 + (3 + 4) + 5)
```

How to Construct Parsing Tables

- Needed: algorithm for automatically generating a predictive parse table from a grammar

```
S \rightarrow ES \\
E \rightarrow \text{number} | S (S) |
```

Constructing Parse Tables

- Can construct predictive parser if:
  1. For every non-terminal, every look-ahead symbol can be handled by at most one production
  2. FIRST($\gamma$) for arbitrary string of terminals and non-terminals $\gamma$'s:
     - set of symbols that might begin the fully expanded version of $\gamma$
  3. FOLLOW($X$) for a non-terminal $X$'s:
     - set of symbols that might follow the derivation of $X$ in the input stream

Parse Table Entries

- Consider a production $X \rightarrow \gamma$
- Add $\gamma$ to the X row for each symbol in FIRST($\gamma$)

```
S \rightarrow \epsilon | +S | ES | E | number |
```

Computing nullable, FIRST

- $X$ is nullable if it can derive the empty string:
  - if it derives $\epsilon$ directly $X \rightarrow \epsilon$
  - if it has a production $X \rightarrow Y \gamma$ where all RHS symbols ($\gamma$, $Z$) are nullable
- Algorithm: assume all non-terminals non-nullable apply rules repeatedly until no change

```
- FIRST($X$) = FIRST($\gamma$) if $X \rightarrow \gamma$
- FIRST($\epsilon$) = $\epsilon$
- FIRST($X \gamma$) $= \text{FIRST}($ if $X \rightarrow \gamma$
- FIRST($X \gamma$) $= \text{FIRST}($ if $X$ is nullable
```

Grammar is LL(1) if no conflicting entries
Computing FOLLOW

- Compute FOLLOW(X):
  - FOLLOW(S) = { $ } (1)
  - FOLLOW(X) = FIRST(Y)
  - If X → yi, FOLLOW(Y) = FOLLOW(Y)
  - If X → yj and $ is nullable (or non-existent), FOLLOW(Y) = FOLLOW(X)

- Algorithm: Assume FOLLOW(X) = {} for all X, apply rules repeatedly to build FOLLOW sets

- Common theme: iterative analysis. Start with initial assignment, apply rules until no change

Example

- nullable
  - only S is nullable

- FIRST
  - FIRST(E $) = (num, ( )
  - FIRST( + ) = ( )
  - FIRST(num) = (num)
  - FIRST(S$) = ( )

FOLLOW

- FOLLOW(S) = { $, }
- FOLLOW(S$) = { $, }
- FOLLOW(E$) = { +, $, $, $}
- FOLLOW(E) = { +, $, $, $}

Ambiguous grammars

- Construction of predictive parse table for ambiguous grammar results in conflicts

S → S + S | S * S | num

FIRST(S + S) = FIRST(S * S) = FIRST(num) = { num }

```
  num   +   *
S → num, →S + S, →S * S
```

Summary

- LL(k) grammar
  - left-to-right scanning
  - leftmost derivation
  - can determine what production to apply from the next k symbols
  - Can automatically build predictive parsing tables

- Predictive parsers
  - Can be easily built for LL(k) grammars from the parsing tables
  - Also called recursive-descent, or top-down parsers