Class 9: Recursive descent and table-driven top-down parsing

SI 413 - Programming Languages and Implementation

Dr. Daniel S. Roche

United States Naval Academy

Fall 2011

Top-down parsing

```
1 Initialize the stack with S, the start symbol.;
2 while stack and input are both not empty do
3     if top of stack is a terminal then
4         Match terminal to next token
5     else
6         Pop nonterminal and replace with
          r.h.s. from a derivation rule
7 Accept iff stack and input are both empty
```

Make choice on Step 6 by "peeking" ahead in the token stream.

LL(1) Grammars

A grammar is LL(1) if it can be parsed top-down with just 1 token's worth of look-ahead.

Example grammar

$S \rightarrow T\ T$
$T \rightarrow \texttt{ab}$
$T \rightarrow \texttt{aa}$

Is this grammar LL(1)?

Common prefixes

The *common prefix* in the previous grammar causes a problem.

In this case, we can "factor out" the prefix:

LL(1) Grammar

$S \rightarrow T\ T$
$T \rightarrow \texttt{a}\ X$
$X \rightarrow \texttt{b}$
$X \rightarrow \texttt{a}$

Left recursion

The other enemy of LL(1) is *left recursion*:

$$S \rightarrow exp$$
$$exp \rightarrow \texttt{exp + NUM}$$
$$exp \rightarrow \texttt{NUM}$$

- Why isn't this LL(1)?
- How could we "fix" it?

Making grammars LL using tail rules

To make LL grammars, we usually end up adding extra "tail rules" for list-like non-terminals.

For instance, the previous grammar can be rewritten as

$$
\begin{aligned}
S &\rightarrow \mathit{exp} \\
\mathit{exp} &\rightarrow \texttt{NUM}\ \mathit{exptail} \\
\mathit{exptail} &\rightarrow \epsilon \;\big|\; \texttt{+}\ \texttt{NUM}\ \mathit{exptail}
\end{aligned}
$$

This is now LL(1).

(Remember that ϵ is the empty string in this class.)

Recall: Calculator language scanner

Token name	Regular expression
NUM	[0-9]+
OPA	[+-]
OPM	[*/]
LP	(
RP	)
STOP	;

LL(1) grammar for calculator language

$$
\begin{aligned}
S &\rightarrow \textit{exp}\ \texttt{STOP} \\
\textit{exp} &\rightarrow \textit{term}\ \textit{exptail} \\
\textit{exptail} &\rightarrow \epsilon \mid \texttt{OPA}\ \textit{term}\ \textit{exptail} \\
\textit{term} &\rightarrow \textit{sfactor}\ \textit{termtail} \\
\textit{termtail} &\rightarrow \epsilon \mid \texttt{OPM}\ \textit{factor}\ \textit{termtail} \\
\textit{sfactor} &\rightarrow \texttt{OPA}\ \textit{factor} \mid \textit{factor} \\
\textit{factor} &\rightarrow \texttt{NUM} \mid \texttt{LP}\ \textit{exp}\ \texttt{RP}
\end{aligned}
$$

How do we know this is LL(1)?

Recursive Descent Parsers

A recursive descent top-down parser uses *recursive functions* for parsing every non-terminal, and uses the function call stack implicitly instead of an explicit stack of terminals and non-terminals.

If we also want the parser to *do something*, then these recursive functions will return values. They will also sometimes take values as parameters.

(See posted examples.)

Table-driven parsing

Auto-generated top-down parsers are usually *table-driven*.

The program stores an *explicit* stack of expected symbols, and applies rules using a nonterminal-token table.

Using the expected non-terminal and the next token, the table tells which production rule in the grammar to apply.

Applying a production rule means pushing some symbols on the stack.

(See posted example.)

Automatic top-down parser generation

In table-driven parsing, the code is always the same;
only the table is different depending on the language.

Top-down parser generators first generate two sets for each non-terminal:

- FIRST: Which tokens can appear at the beginning of a non-terminal
- FOLLOW: Which non-terminals can come after this non-terminal

There are simple rules for generating FIRST and FOLLOW, and then for generating the parsing table using these sets.