Introduction to Algorithms and Data Structures

Lecture 23: LL(1) predictive parsing

John Longley

School of Informatics
University of Edinburgh

9 February 2024
Efficient parsing for artificial languages

Consider how we’d like to parse a program in the little programming language from Lecture 20.

\[
\begin{align*}
\text{stmt} & \rightarrow \text{if-stmt} \mid \text{while-stmt} \mid \text{begin-stmt} \mid \text{assg-stmt} \\
\text{if-stmt} & \rightarrow \text{if bool-expr then stmt else stmt} \\
\text{while-stmt} & \rightarrow \text{while bool-expr do stmt} \\
\text{begin-stmt} & \rightarrow \text{begin stmt-list end} \\
\text{stmt-list} & \rightarrow \text{stmt} \mid \text{stmt ; stmt-list} \\
\text{assg-stmt} & \rightarrow \text{var := arith-expr} \\
\text{bool-expr} & \rightarrow \text{arith-expr compare-op arith-expr} \\
\text{compare-op} & \rightarrow < \mid > \mid <= \mid >= \mid == \mid !=
\end{align*}
\]

We’d like to read the program from left to right, processing each token (i.e. terminal symbol occurrence) only once — hoping for \(O(n)\) runtime.
Predictive parsing: the idea

Start symbol is `stmt`.

Want to construct a leftmost derivation of our program starting from this (i.e. expanding the leftmost non-terminal at each step).

Let’s suppose the first token in the program is `begin`.

From this alone, we can tell that the first two steps must be

\[
\begin{align*}
\text{stmt} & \rightarrow \text{begin-stmt} \\
& \rightarrow \text{begin stmt-list end}
\end{align*}
\]

So we have to parse the complete program as `begin stmt-list end`.

Can now step over `begin`, and proceed to parse the remaining input as `stmt-list end`.

We think of this as the predicted form for the remaining input.
LL(1) predictive parsing: intuition

In each of these two steps, the correct production to apply has been determined from just two pieces of information:

- the current token (e.g. `begin`).
- the nonterminal to be expanded (e.g. `stmt`, `begin-stmt`).

If it’s always possible to determine the next production from just this information, then the grammar is said to be LL(1).
(Meaning: read input from Left; build Leftmost derivation; look just 1 token ahead.) In this case, parsing can be very efficient.

Unfortunately, our example grammar isn’t quite LL(1), and the very next step illustrates this.

We now have to expand `stmt-list`. Suppose second input token is `if`. Which rule should we apply?

```
stmt-list → stmt  or  stmt-list → stmt ; stmt-list
```

No way to tell without further lookahead!
Fixing a grammar

In this case, we can recast the rules for \texttt{stmt-list} to fix the problem:

\[
\text{stmt-list} \rightarrow \text{stmt} \text{ stmt-tail} \\
\text{stmt-tail} \rightarrow \epsilon \mid ; \text{ stmt} \text{ stmt-tail}
\]

From the \texttt{if}, now see that the next two rules must be:

\[
\text{stmt-list} \rightarrow \text{stmt} \text{ stmt-tail} \\
\text{stmt} \rightarrow \text{if-stmt} \\
\text{if-stmt} \rightarrow \texttt{if bool-expr then stmt else stmt}
\]

The whole derivation so far:

\[
\text{stmt} \rightarrow \text{begin-stmt} \\
\rightarrow \texttt{begin stmt-list end} \\
\rightarrow \texttt{begin stmt stmt-tail end} \\
\rightarrow \texttt{begin if-stmt stmt-tail end} \\
\rightarrow \texttt{begin if bool-expr then stmt else stmt stmt-tail end}
\]

This accounts for the first two tokens, \texttt{begin if}. Predicted form for the rest is \texttt{bool-expr then stmt else stmt stmt-tail end}.
Parse tables

Consider the following grammar for bracket sequences, e.g. (()))()

\[
S \rightarrow \epsilon \mid TS \\
T \rightarrow (S)
\]

This is LL(1): can always tell from the ‘current token’ and ‘current non-terminal’ which rule to apply. Take on trust for now.

Idea: That means we can draw up a 2-dim parse table, telling us which rule to apply in any situation. In this case:

<table>
<thead>
<tr>
<th></th>
<th>(   )</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$S \rightarrow TS$</td>
<td>$S \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$T$</td>
<td>$T \rightarrow (S)$</td>
<td>$S \rightarrow \epsilon$</td>
</tr>
</tbody>
</table>

- Columns are labelled by terminals plus ‘end-of-input’ marker $.$.
- Rows are labelled by non-terminals.
- Entry in column $a$ and row $X$ tells us the rule to apply if we have $a$ in the input and $X$ is predicted.
- Blank entries: situations that never arise for a legal input.

Parsing is now easy: at each step, just do what the table tells us!
Example of LL(1) parsing

\[
\begin{array}{c|ccc}
S & S \rightarrow TS & S \rightarrow \epsilon & S \rightarrow \epsilon \\
\hline
T & T \rightarrow (S) & & \\
\end{array}
\]

Let’s use this table to parse the input string \((())\).
A stack keeps track of the predicted form for remaining input.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Remaining input</th>
<th>Stack state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lookup (, S)</td>
<td>((()))$</td>
<td>(S)</td>
</tr>
<tr>
<td>Lookup (, T)</td>
<td>((()))$</td>
<td>(TS)</td>
</tr>
<tr>
<td>Match ()</td>
<td>((()))$</td>
<td>((S)S)</td>
</tr>
<tr>
<td>Lookup (, S)</td>
<td>((()))$</td>
<td>(TS)S)</td>
</tr>
<tr>
<td>Lookup (, T)</td>
<td>((()))$</td>
<td>((S)S)S)</td>
</tr>
<tr>
<td>Match ()</td>
<td>((()))$</td>
<td>((S)S)S)</td>
</tr>
<tr>
<td>Lookup (, S)</td>
<td>((()))$</td>
<td>(S)S)</td>
</tr>
<tr>
<td>Match ()</td>
<td>((()))$</td>
<td>(S)S)</td>
</tr>
<tr>
<td>Lookup (, S)</td>
<td>((()))$</td>
<td>(S)</td>
</tr>
<tr>
<td>Match ()</td>
<td>((()))$</td>
<td>(S)</td>
</tr>
<tr>
<td>Lookup $ S)</td>
<td>((()))$</td>
<td>empty stack</td>
</tr>
</tbody>
</table>

(Also easy to build a syntax tree as we go along!)
Short exercise

\[
\begin{array}{c|cc|c}
  \text{S} & ( & \text{S} \to TS & \text{S} \to \epsilon & \text{S} \to \epsilon \\
  \text{T} & \text{T} \to (S) & & \\
\end{array}
\]

For each of the following two input strings:

1. Blank entry in table encountered
2. Input symbol (or end marker) doesn’t match predicted symbol
3. Stack empties before end of string reached

**Answer:** For ), we start by expanding S to \( \epsilon \). But this empties the stack, whereas we haven’t consumed any input yet. So 3.

For (, we get to a point where we’ve reached the end marker $ in the input, which doesn't match the predicted symbol ‘)’ on the stack. So 2.
LL(1) parsing: the algorithm

\textbf{LL1\_Parse} (table,S,input)

\begin{verbatim}
pos = 0
initialize stack with single entry S
while stack not empty
  x = stack.peek()
  if x is non-terminal \# Lookup case
    case table[x,input[pos]] of
      blank: error
      rule x \rightarrow \beta:
        stack.pop()
        push symbols of \beta onto stack
        (backwards!)
    else \# Match case
      if x = input[pos]
        stack.pop()
        pos += 1
      else error
  else if input[pos] = $
    return Success
  else error
\end{verbatim}
Parse table revisited

**Remember:** The parse table entry for $X, a$ tells us which rule to apply if we’re expecting an $X$ and see an $a$.

► Often, the $a$ will be simply the first symbol of the $X$-subphrase in question.

► But not always: maybe the $X$-subphrase in question is $\epsilon$, and the $a$ belongs to whatever follows the $X$.

<table>
<thead>
<tr>
<th></th>
<th>$(\quad)$</th>
<th>$$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$S \rightarrow TS$</td>
<td>$S \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$T$</td>
<td>$T \rightarrow (S)$</td>
<td></td>
</tr>
</tbody>
</table>

In this simple case, not too hard to see by ad-hoc reasoning that the parse table is correct.

For a large grammar, this might be hard!

However, there’s an **algorithm** that takes a grammar and constructs the parse table (or else detects the grammar isn’t LL(1)). Involves **First** and **Follow** sets — but won’t pursue this here.
Further remarks

- LL(1) is an example of a top-down parser: builds syntax trees from the root. Contrast with CYK which is bottom-up.
- For any ‘naturally arising’ LL(1) grammar, easy to see that our parser runs in time $\Theta(n)$ (with small hidden constants).
- Not every CF language has an LL(1) grammar. But if we’re designing a language, we can try to ensure that it does!
- LL(1) is nice for simple ‘command languages’ — and the ‘lightweight’ parsing algorithm is a plus.
- For large-scale languages, may want a bit more flexibility. Common choice is LR(1) parsing (more complex than LL(1)).
- In the real world, no one implements parsers for large languages by hand! We just write a CFG, then run a parser generator which creates one automatically — typically by constructing a parse table.
Putting it in context: The language processing pipeline

Think about the phases in which e.g. a Java program is processed:

Raw source text (e.g. x2=−x1) ⇒ lexing
Stream of tokens (e.g. x2, =, −, x1) ⇒ parsing
Syntax tree ⇒ typechecking etc.
Annotated syntax tree ⇒ compiling, optimization
Java bytecode ⇒ linking
JVM state ⇒ running
Program behaviour

IADS Lecture 23 Slide 12
The language processing pipeline (NL version)

Broadly similar pipeline e.g. for spoken English:

Raw soundwaves

⇓

Phonetics

Phones (e.g. [pʰ]–pot, [p]–spot)

⇓

Phonology

Phonemes (e.g. /p/, /b/)

⇓

Segmentation, tagging

Words, morphemes, part-of-speech info

⇓

Parsing

Syntax tree

⇓

Agreement checking etc.

Annotated syntax tree

⇓

Semantics

Logical form or ‘meaning’

⇓

... 

Though with ambiguity at all stages, and much ‘feedback’ from later stages to earlier ones.
Reading

- Appel and Ginsburg, *Modern Compiler Implementation in C*, Sections 3.1 and 3.2. Online access via UoE library. More detail than we need: covers the algorithm for constructing the parse table. Equivalent books exist for ML and Java (but no online library access for the latter).

- Aho, Sethi and Ullman, *Compilers: Principles, Techniques, Tools*, Section 4.4. Close to our treatment, but may be hard to find online.