Introduction to Algorithms and Data Structures

Lecture 24: LL(1) predictive parsing

John Longley

School of Informatics
University of Edinburgh

11 February 2025

Efficient parsing for artificial languages

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

$$
\begin{array}{rcl}
\text{stmt} & \rightarrow & \text{if-stmt} \mid \text{while-stmt} \mid \text{begin-stmt} \mid \text{assg-stmt} \\
\text{if-stmt} & \rightarrow & \textbf{if} \text{ bool-expr } \textbf{then} \text{ stmt } \textbf{else} \text{ stmt} \\
\text{while-stmt} & \rightarrow & \textbf{while} \text{ bool-expr } \textbf{do} \text{ stmt} \\
\text{begin-stmt} & \rightarrow & \textbf{begin} \text{ stmt-list } \textbf{end} \\
\text{stmt-list} & \rightarrow & \text{stmt} \mid \text{stmt ; stmt-list} \\
\text{assg-stmt} & \rightarrow & \text{var} := \text{arith-expr} \\
\text{bool-expr} & \rightarrow & \text{arith-expr compare-op arith-expr} \\
\text{compare-op} & \rightarrow & < \mid > \mid <= \mid >= \mid == \mid !=
\end{array}
$$

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

$$\begin{aligned} \text{stmt} &\rightarrow \text{begin-stmt} \\ &\rightarrow \textbf{begin}\ \text{stmt-list}\ \textbf{end} \end{aligned}$$

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 $\rightarrow$ stmt   or   stmt-list $\rightarrow$ stmt ; stmt-list   ?

No way to tell without further lookahead!

Fixing a grammar

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

$$\begin{aligned} \text{stmt-list} &\rightarrow \text{stmt stmt-tail} \\ \text{stmt-tail} &\rightarrow \epsilon \mid \text{; stmt stmt-tail} \end{aligned}$$

From the **if**, now see that the next two rules must be:

$$\begin{aligned} \text{stmt-list} &\rightarrow \text{stmt stmt-tail} \\ \text{stmt} &\rightarrow \text{if-stmt} \\ \text{if-stmt} &\rightarrow \textbf{if} \text{ bool-expr } \textbf{then} \text{ stmt } \textbf{else} \text{ stmt} \end{aligned}$$

The whole derivation so far:

$$\begin{aligned} \text{stmt} &\rightarrow \text{begin-stmt} \\ &\rightarrow \textbf{begin} \text{ stmt-list } \textbf{end} \\ &\rightarrow \textbf{begin} \text{ stmt stmt-tail } \textbf{end} \\ &\rightarrow \textbf{begin} \text{ if-stmt stmt-tail } \textbf{end} \\ &\rightarrow \textbf{begin if} \text{ bool-expr } \textbf{then} \text{ stmt } \textbf{else} \text{ stmt stmt-tail } \textbf{end} \end{aligned}$$

This accounts for the first two tokens, **begin if**. Predicted form for the rest is bool-expr **then** stmt **else** stmt stmt-tail **end**.

Parse tables

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

$$S \rightarrow \epsilon \mid TS \qquad T \rightarrow (S)$$

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:

	(	)	\$
S	$S \rightarrow TS$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$
T	$T \rightarrow (S)$		

- 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

	(	)	$
S	$S \to TS$	$S \to \epsilon$	$S \to \epsilon$
T	$T \to (S)$		

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

Operation	Remaining input	Stack state
	(())$	S
Lookup (, S	(())$	TS
Lookup (, T	(())$	$(S)S$
Match (	())$	$S)S$
Lookup (, S	())$	$TS)S$
Lookup (, T	())$	$(S)S)S$
Match (	))$	$S)S)S$
Lookup), S	))$	$)S)S$
Match)	)$	$S)S$
Lookup), S	)$	$)S$
Match)	$	S
Lookup $, S	$	empty stack

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

Short exercise

	(	)	\$
S	$S \to TS$	$S \to \epsilon$	$S \to \epsilon$
T	$T \to (S)$		

For each of the following two input strings:

)   (

what will go wrong when we try to apply our parsing algorithm?

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 ϵ. 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

```
LL1_Parse (table,S,input)
    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 → β:
                    stack.pop()
                    push symbols of β onto stack
                        (backwards!)
        else                          # Match case
            if x = input[pos]
                stack.pop()
                pos += 1
            else error
    if input[pos] = $
        return Success
    else error
```

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 ϵ, and the a belongs to whatever follows the X.

	(	)	\$
S	$S \to TS$	$S \to \epsilon$	$S \to \epsilon$
T	$T \to (S)$		

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`)

$\Downarrow$ lexing

Stream of tokens (e.g. `x2, =, -, x1`)

$\Downarrow$ parsing

Syntax tree

$\Downarrow$ typechecking etc.

Annotated syntax tree

$\Downarrow$ compiling, optimization

Java bytecode

$\Downarrow$ linking

JVM state

$\Downarrow$ running

Program behaviour

The language processing pipeline (NL version)

Broadly similar pipeline e.g. for spoken English:

Raw soundwaves

⇓ phonetics

Phones (e.g. $[\mathrm{p}^{h}]$–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.