

LL(1) predictive parsing

Informatics 2A: Lecture 10

Alex Simpson

School of Informatics
University of Edinburgh
als@staffmail.ed.ac.uk

8 October, 2013

- 1 LL(1) grammars and parse tables
- 2 Predictive parsing using a parse table
- 3 When is a grammar LL(1)?

LL(1) grammars: the intuition

Think about a (one-state) NPDA derived from a CFG \mathcal{G} .

Intuition: at each step, our NPDA can 'see' two things:

- the current input symbol
- the topmost stack symbol

Roughly speaking, we say \mathcal{G} is an **LL(1) grammar** if, just from this information, it's possible to determine which transition/production to apply next. (**Precise definition later!**) If this is so, parsing can proceed deterministically and efficiently.

Here LL(1) means 'read input from **L**eft, build **L**eftmost derivation, look just **one** symbol ahead'.

Subtle point: we can use the input symbol to help us choose a transition, even if this transition doesn't consume that input symbol. . . hence 'look ahead'.

Parse tables

Saying the current input symbol and stack symbol uniquely determine the production means we can draw up a two-dimensional **parse table** telling us which production to apply in any situation.

Consider e.g. the following grammar for well-bracketed sequences:

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

This has the following parse table:

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

- **Columns** are labelled by **terminals**, which are the input symbols. We include an extra column for an 'end-of-input' marker \$.
- **Rows** are labelled by **nonterminals**. If the current stack symbol is a terminal, just match it against current input symbol.
- **Blank entries** correspond to situations that can never arise in processing a legal input string.

Predictive parsing with parse tables

Given such a parse table, parsing can be done very efficiently using a stack. The stack (reading downwards) records the **predicted sentential form** for the remaining part of the input.

- Begin with just start symbol S on the stack.
- If current input symbol is a (maybe $\$$), and current stack symbol is a non-terminal X , look up rule for a, X in table.
[Error if no rule.] If rule is $X \rightarrow \beta$, pop X and replace with β (pushed right-end-first!)
- If current input symbol is a and stack symbol is a , just pop a from stack and advance input read position.
[Error if stack symbol is any other terminal.]
- Accept if stack empties with $\$$ as input symbol.
[Error if stack empties sooner.]

Example of predictive parsing

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

Let's use this table to parse the input string $((()))$.

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!)

Clicker questions

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

For each of the following two input strings:

) (

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

- ❶ Blank entry in table encountered
- ❷ Input symbol (or end marker) doesn't match expected symbol
- ❸ Stack empties before end of string reached

Further remarks

Slogan: the parse table entry for a, X 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 .

E.g. in the lookups for $), S$ on the previous slide, the S in question turns out to be empty.

Once we've got a parse table for a given grammar \mathcal{G} , we can parse strings of length n in $O(n)$ time (and $O(n)$ space).

Our algorithm is an example of a **top-down predictive parser**: it works by 'predicting' the form of the remainder of the input, and builds syntax trees a top-down way (i.e. starting from the root). There are other parsers (e.g. LR(1)) that work 'bottom up'.

LL(1) grammars: formal definition

Suppose \mathcal{G} is a CFG containing no 'useless' nonterminals, i.e.

- every nonterminal appears in some sentential form derived from the start symbol;
- every nonterminal can be expanded to some (possibly empty) string of terminals.

We say \mathcal{G} is **LL(1)** if for each terminal a and nonterminal X , there is some production $X \rightarrow \alpha$ with the following property:

If $b_1 \dots b_n X \gamma$ is a sentential form appearing in a leftmost derivation of some string $b_1 \dots b_n a c_1 \dots c_m$ ($n, m \geq 0$), the next sentential form appearing in the derivation is necessarily $b_1 \dots b_n \alpha \gamma$.

(Note that if a, X corresponds to a 'blank entry' in the table, *any* production $X \rightarrow \alpha$ will satisfy this property, because a sentential form $b_1 \dots b_n X \gamma$ can't arise.)

Non-LL(1) grammars

Roughly speaking, a grammar is **by definition** LL(1) if and only if there's a parse table for it. Not all CFGs have this property!

Consider e.g. a different grammar for the same language of well-bracketed sequences:

$$S \rightarrow \epsilon \mid (S) \mid SS$$

Suppose we'd set up our initial stack with S , and we see the input symbol $($. What rule should we apply?

- If the input string is $(())$, should apply $S \rightarrow (S)$.
- If the input string is $()()$, should apply $S \rightarrow SS$. We can't tell without looking further ahead.

Put another way: if we tried to build a parse table for this grammar, the two rules $S \rightarrow (S)$ and $S \rightarrow SS$ would be competing for the slot $(, S$. So this grammar is **not** LL(1).

Remaining issues

Easy to see from the definition that any LL(1) grammar will be **unambiguous**: never have two syntax trees for the same string.

- For **computer languages**, this is fine: normally want to avoid ambiguity anyway.
- For **natural languages**, ambiguity is a fact of life! So LL(1) grammars are normally inappropriate.

Two outstanding questions. . .

- How can we tell if a grammar is LL(1) — and if it is, how can we construct a parse table? (See [Lecture 11.](#))
- If a grammar isn't LL(1), is there any hope of replacing it by an equivalent one that is? (See [Lecture 12.](#))

Reading and prospectus

Relevant reading:

- Some lecture notes from a previous year (covering the same material but with different examples) are available via the course website. (See [Readings](#) column of course schedule.)
- See also Aho, Sethi and Ullman, *Compilers: Principles, Techniques, Tools*, Section 4.4.