Fixing problems with grammars

Informatics 2A: Lecture 13

John Longley

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

18 October, 2011

LL(1) grammars: summary

Given a context-free grammar, the problem of parsing a string can be seen as that of constructing a leftmost derivation, e.g.

$$\begin{aligned}
\mathsf{Exp} &\Rightarrow \mathsf{Exp} + \mathsf{Exp}\\
&\Rightarrow \mathsf{Num} + \mathsf{Exp}\\
&\Rightarrow 1 + \mathsf{Exp}\\
&\Rightarrow 1 + \mathsf{Num}\\
&\Rightarrow 1 + 2
\end{aligned}$$

At each stage, we expand the leftmost nonterminal. In general, it (seemingly) requires magical powers to know which rule to apply.

An LL(1) grammar is one in which the correct rule can always be determined from just the nonterminal to be expanded and the current input symbol (or end-of-input marker).

This leads to the idea of a parse table: a two-dimensional array (indexed by nonterminals and input symbols) in which the appropriate production can be looked up at each stage.

Possible problems with grammars

LL(1) grammars allow for very efficient parsing (time linear in length of input string). Unfortunately, many "natural" grammars aren't LL(1), for various reasons, e.g.

1. They may be ambiguous (bad anyway for computer langs.)
2. They may have rules with shared prefixes: e.g. how would we choose between the following productions?

$$\begin{array}{lcl} \mathsf{Stmt} & \rightarrow & \mathsf{do\ Stmt}\ \mathtt{while}\ \mathsf{Cond} \\ \mathsf{Stmt} & \rightarrow & \mathsf{do\ Stmt}\ \mathtt{until}\ \mathsf{Cond} \end{array}$$

3. They may be left-recursive rules, where the LHS nonterminal appears at the start of the RHS: $\mathsf{Exp} \rightarrow \mathsf{Exp} + \mathsf{Exp}$

Sometimes such problems can be fixed: we can replace our grammar by an equivalent LL(1) one. We'll look at some ways of doing this.

Problem 1: Ambiguity

We've seen many examples of ambiguous grammars. Some kinds of ambiguity are 'needless' and can be easily avoided. E.g. can replace

$$\text{List} \;\rightarrow\; \epsilon \;\mid\; \text{Item} \;\mid\; \text{List List}$$

by

$$\text{List} \;\rightarrow\; \epsilon \;\mid\; \text{Item List}$$

A similar trick works generally for any other kind of 'lists'.
E.g. can replace

$$\text{List1} \;\rightarrow\; \text{Item} \;\mid\; \text{List1} \;;\; \text{List1}$$

by

$$\text{List1} \;\rightarrow\; \text{Item Rest} \qquad \text{Rest} \;\rightarrow\; \epsilon \;\mid\; ;\; \text{Item Rest}$$

Resolving ambiguity with added nonterminals

More serious example of ambiguity:

$$\begin{array}{rcl} \mathsf{Exp} & \rightarrow & \mathsf{Num} \mid \mathsf{Var} \mid (\mathsf{Exp}) \mid -\mathsf{Exp} \\ & \mid & \mathsf{Exp} + \mathsf{Exp} \mid \mathsf{Exp} - \mathsf{Exp} \mid \mathsf{Exp} * \mathsf{Exp} \mid \mathsf{Exp}/\mathsf{Exp} \end{array}$$

We can disambiguate this by adding nonterminals to capture more subtle distinctions between different classes of expressions:

$$\begin{array}{rcl} \mathsf{Exp} & \rightarrow & \mathsf{ExpA} \mid \mathsf{Exp} + \mathsf{ExpA} \mid \mathsf{Exp} - \mathsf{ExpA} \\ \mathsf{ExpA} & \rightarrow & \mathsf{ExpB} \mid \mathsf{ExpA} * \mathsf{ExpB} \mid \mathsf{ExpA} \, / \, \mathsf{ExpB} \\ \mathsf{ExpB} & \rightarrow & \mathsf{ExpC} \mid -\mathsf{ExpC} \\ \mathsf{ExpC} & \rightarrow & \mathsf{Num} \mid \mathsf{Var} \mid (\mathsf{Exp}) \end{array}$$

Note that this builds in certain design decisions concerning what we want the rules of precedence to be — shouldn't entrust this process to a machine!

N.B. our revised grammar is unambiguous, but not yet LL(1) . . .

Problem 2: Shared prefixes

Consider the two productions

$$\begin{aligned} \text{Stmt} &\rightarrow \text{do Stmt } \texttt{while} \text{ Cond} \\ \text{Stmt} &\rightarrow \text{do Stmt } \texttt{until} \text{ Cond} \end{aligned}$$

On encountering the nonterminal Stmt and the terminal `do`, an LL(1) parser would have no way of choosing between these two rules.

Solution: factor out the common part of these rules, so 'delaying' the decision until the relevant information becomes available:

$$\begin{aligned} \text{Stmt} &\rightarrow \text{do Stmt Test} \\ \text{Test} &\rightarrow \texttt{while} \text{ Cond} \mid \texttt{until} \text{ Cond} \end{aligned}$$

This simple trick is known as left factoring.

Problem 3: Left recursion

Suppose our grammar contains a rule like

$$\text{Exp} \;\rightarrow\; \text{Exp} + \text{ExpA}$$

Problem: whatever terminals Exp could begin with, Exp + ExpA could also begin with. So there's a danger our parser would apply this rule indefinitely:

$$\text{Exp} \;\Rightarrow\; \text{Exp} + \text{ExpA} \;\Rightarrow\; \text{Exp} + \text{ExpA} + \text{ExpA} \;\Rightarrow\; \cdots$$

(In practice, we wouldn't even get this far: there'd be a clash in the parse table, e.g. at Num, Exp.)

So left recursion makes a grammar non-LL(1).

Eliminating left recursion

Consider e.g. the rules

$$\mathrm{Exp} \;\rightarrow\; \mathrm{ExpA} \;\mid\; \mathrm{Exp} + \mathrm{ExpA} \;\mid\; \mathrm{Exp} - \mathrm{ExpA}$$

Taken together, these say that Exp can consist of ExpA followed by zero or more suffixes $+$ExpA or $-$ExpA.

So why not just say this?!

$$\mathrm{Exp} \;\rightarrow\; \mathrm{ExpA\ OpsA} \qquad \mathrm{OpsA} \;\rightarrow\; \epsilon \;\mid\; +\mathrm{ExpA\ OpsA} \;\mid\; -\mathrm{ExpA\ OpsA}$$

(Reminiscent of Arden's rule.) Likewise:

$$\mathrm{ExpA} \;\rightarrow\; \mathrm{ExpB\ OpsB} \qquad \mathrm{OpsB} \;\rightarrow\; \epsilon \;\mid\; +\mathrm{ExpB\ OpsB} \;\mid\; -\mathrm{ExpB\ OpsB}$$

Together with the earlier rules for ExpB and ExpC, these give an LL(1) version of our grammar for arithmetic expressions.

Indirect left recursion

Left recursion can also arise in a more indirect way. E.g.

$$A \rightarrow a \mid Bc \qquad B \rightarrow b \mid Ad$$

By considering the combined effect of these rules, can see that they are equivalent to

$$\begin{array}{lllll} A & \rightarrow & aE \mid bcE & \quad B & \rightarrow \; bF \mid adF \\ E & \rightarrow & \epsilon \mid dcE & \quad F & \rightarrow \; \epsilon \mid cdF \end{array}$$

(Won't go into the systematic method here.)

LL(1) grammars: summary

- Often (not always), a “natural” grammar for some language of interest can be massaged into an LL(1) grammar. This allows for very efficient parsing.
- Knowing a grammar is LL(1) also assures us that it is unambiguous — often non-trivial! By the same token, LL(1) grammars are poorly suited to natural languages.
- However, an LL(1) grammar may be less readable and intuitive than the original. It may also appear to mutilate the ‘natural’ structure of phrases. We must take care not to mutilate it so much that we can no longer ‘execute’ the phrase as intended.
- One can design realistic computer languages with LL(1) grammars. For less cumbersome syntax that ‘flows’ better, one might want to go a bit beyond LL(1) (e.g. to LR(1)), but the principles remain the same.

Example of an LL(1) grammar

Here is a minor modification of the programming language grammar from Lecture 9. Combining it with our revised grammar for arithmetic expressions, we get an LL(1) grammar for a respectable programming language.

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

Addendum: Chomsky Normal Form

Whilst on the subject of ‘transforming grammars into equivalent ones of some special kind’ . . .

A context-free grammar $\mathcal{G} = (N, \Sigma, P, S)$ is in Chomsky normal form (CNF) if all productions are of the form

$$A \;\rightarrow\; BC \quad \text{or} \quad A \;\rightarrow\; a \qquad (A, B, C \in N,\;\; a \in \Sigma)$$

Theorem: Disregarding the empty string, every CFG $\mathcal{G}$ is equivalent to a grammar $\mathcal{G}'$ in Chomsky normal form. $(\mathcal{L}(\mathcal{G}') = \mathcal{L}(\mathcal{G}) - \{\epsilon\})$

This is useful, because certain general parsing algorithms (e.g. the CYK algorithm, see Lecture 17) work best for grammars in CNF.

Converting to Chomsky Normal Form

Consider for example the grammar

$$S \;\to\; TT \;\mid\; [S] \qquad\qquad T \;\to\; \epsilon \;\mid\; (T)$$

Step 1: remove all ϵ-productions, and for each rule $X \to \alpha Y \beta$, add a new rule $X \to \alpha\beta$ whenever Y 'can be empty'.

$$S \;\to\; TT \;\mid\; T \;\mid\; [S] \;\mid\; [\,] \qquad\qquad T \;\to\; (T) \;\mid\; ()$$

Step 2: remove 'unit productions' $X \to Y$.

$$S \;\to\; TT \;\mid\; (T) \;\mid\; () \;\mid\; [S] \;\mid\; [\,] \qquad\qquad T \;\to\; (T) \;\mid\; ()$$

Now all productions are of form $X \to a$ or $X \to x_1 \ldots x_k$ $(k \geq 2)$.

Converting to Chomsky Normal Form, ctd.

$$S \;\rightarrow\; TT \mid (T) \mid () \mid [S] \mid [\,] \qquad\qquad T \;\rightarrow\; (T) \mid ()$$

Step 3: For each terminal a, add a nonterminal Z_a and a production $Z_a \rightarrow a$. In all rules $X \rightarrow x_1 \ldots x_k$ ($k \geq 2$), replace each a by Z_a.

$$S \;\rightarrow\; TT \mid Z_(T Z_) \mid Z_(Z_) \mid Z_[S Z_] \mid Z_[Z_]$$

$$T \;\rightarrow\; Z_(T Z_) \mid Z_(Z_) \qquad Z_(\rightarrow (\quad Z_) \rightarrow) \quad Z_[\rightarrow [\quad Z_] \rightarrow]$$

Step 4: For every production $X \rightarrow Y_1 \ldots Y_n$ with $n \geq 3$, add new symbols $W_2, \ldots, W_{n-1}$ and replace the production with $X \rightarrow Y_1 W_2, \quad W_2 \rightarrow Y_2 W_3, \quad \ldots, \quad W_{n-1} \rightarrow Y_{n-1} Y_n$.

E.g. $S \rightarrow Z_(T Z_) \mid Z_[S Z_]$ become

$$S \rightarrow Z_(W \qquad W \rightarrow T Z_) \qquad S \rightarrow Z_[V \qquad V \rightarrow S Z_]$$

The resulting grammar is now in Chomsky Normal Form.

Reading

- Making grammars LL(1): former lecture notes available via the Course Schedule webpage.
- Chomsky Normal Form: Kozen chapter 21, Jurafsky & Martin section 12.5.

Some light relief: Palindromic sentences

Not too hard to construct 'nonsense' palindromic sentences or phrases in English. (Ignore whitespace, punctuation, case distinctions.) E.g.

I made reviled tubs repel; no, it is opposition, lepers, but delivered am I.

Much more satisfying to find examples that are coherent or interesting in some other way. E.g.

never odd or even

"Norma is as selfless as I am, Ron."

Live dirt up a side-track carted is a putrid evil.

Famous example

A man, a plan, a canal — Panama!

. . . which some smartaleck noticed could be tweaked to . . .

A dog, a plan, a canal — Pagoda!

But for me there's nothing to equal . . .

Best palindrome in the world?

(From Guy Steele, *Common Lisp Reference Manual*, 1983.)

> *A man, a plan, a canoe, pasta, heros, rajahs, a coloratura, maps, snipe, percale, macaroni, a gag, a banana bag, a tan, a tag, a banana bag again (or a camel), a crepe, pins, Spam, a rut, a Rolo, cash, a jar, sore hats, a peon, a canal — Panama!*