Lecture Notes on Bottom-Up LR Parsing

15-411: Compiler Design
Frank Pfenning*

Lecture 9

1 Introduction

In this lecture we discuss a second parsing algorithm that traverses the input string from left to right. The parsing algorithm LL(1) from the last lecture makes a decision on which grammar production to use based on the first character of the input string. Now we consider LR(1), which can postpone the decision at first by pushing input characters onto a stack and then deciding on the production later, taking into account both the first input character and the stack. There are several LL parser generator tools and several LR parser generator tools. Most hand-written parsers are recursive descent parsers, which follow the LL(1) principles.

Alternative presentations of the material in this lecture can be found in a paper by Shieber et al. [SSP95]. The textbook [App98, Chapter 3] covers parsing. For more detailed background on parsing, also see [WM95].

2 LR(1) Parsing

One difficulty with LL(1) parsing is that it is often difficult or impossible to rewrite a grammar so that 1 token look-ahead during a left-to-right traversal becomes unambiguous. To illustrate this, we return to the earlier exam-

*With edits by André Platzer

ple of simple arithmetic expressions.

$$\begin{array}{lrcl} [\mathsf{plus}] & E & \longrightarrow & E + E \\ [\mathsf{times}] & E & \longrightarrow & E \star E \\ [\mathsf{ident}] & E & \longrightarrow & id \\ [\mathsf{number}] & E & \longrightarrow & num \\ [\mathsf{parens}] & E & \longrightarrow & (\ E\ ) \end{array}$$

If we see a simple expression such as `3 + 4 * 5` (which becomes the token stream num + num $\star$ num), we cannot predict when we see the + symbol which production to use because of the inherent ambiguity of the grammar.

We can rewrite the grammar, at significant expense of readability, or we could just specify that multiplication has higher precedence than addition, $+ < \star$. Obviously, the latter is more convenient, but how can we make it work?

The idea is to put off the decision on which productions to use and just *shift* the input symbols onto a stack until we can finally make the decision! We write

$\gamma \mid w$ parse input w under stack γ

where, as generally in predictive parsing, the rules are interpreted as transitions from the conclusion to the premises. The parsing attempt succeeds if we can consume all of w and and produce the start symbol S on the left-hand side. That is, the deduction representing a successful parse of terminal string w_0 has the form

$$\begin{array}{c} \dfrac{}{S \mid \epsilon}\ R_1 \\ \vdots \\ \epsilon \mid w_0 \end{array}$$

Parsing is defined by the following rules:

$$\frac{}{S \mid \epsilon}\ R_1$$

$$\frac{\gamma\, \mathtt{a} \mid w}{\gamma \mid \mathtt{a}\, w}\ R_2(= \mathsf{shift}) \qquad \frac{\begin{array}{l}[r] X \longrightarrow \beta \\ \gamma\, X \mid w\end{array}}{\gamma\, \beta \mid w}\ R_3(r)(= \mathsf{reduce}(r))$$

We resume the example above, parsing $num + num \star num$. After one step (reading this bottom-up)

$$\begin{array}{rcll} num & | & + \; num \star num & ? \\ \epsilon & | & num + num \star num & \mathsf{shift} \end{array}$$

we already have to make a decision: should we shift + or should we reduce num using rule [number]. In this case the action to reduce is forced, because we will never get another chance to see this num as an E.

$$\begin{array}{rcll} E & | & + \; num \star num & ? \\ num & | & + \; num \star num & \mathsf{reduce(number)} \\ \epsilon & | & num + num \star num & \mathsf{shift} \end{array}$$

At this point we need to shift +; no other action is possible. We take a few steps and arrive at

$$\begin{array}{rcll} E + E & | & \star \; num & \\ E + num & | & \star \; num & \mathsf{reduce(number)} \\ E + & | & num \star num & \mathsf{shift} \\ E & | & + \; num \star num & \mathsf{shift} \\ num & | & + \; num \star num & \mathsf{reduce(number)} \\ \epsilon & | & num + num \star num & \mathsf{shift} \end{array}$$

At this point, we have a real conflict. We can either reduce, viewing $E + E$ as a subexpression, or shift and later consider $E \star E$ as a subexpression. Since the $\star$ has higher precedence than +, we need to shift.

$$\begin{array}{rcll} E & | & \epsilon & R_1 \\ E + E & | & \epsilon & \mathsf{reduce(plus)} \\ E + E \star E & | & \epsilon & \mathsf{reduce(times)} \\ E + E \star num & | & \epsilon & \mathsf{reduce(number)} \\ E + E \star & | & num & \mathsf{shift} \\ E + E & | & \star \; num & \mathsf{shift} \\ E + num & | & \star \; num & \mathsf{reduce(number)} \\ E + & | & num \star num & \mathsf{shift} \\ E & | & + \; num \star num & \mathsf{shift} \\ num & | & + \; num \star num & \mathsf{reduce(number)} \\ \epsilon & | & num + num \star num & \mathsf{shift} \end{array}$$

Since E was the start symbol in this example, this concludes the deduction. If we now read the lines from the top to the bottom, ignoring the separator,

we see that it represents a *rightmost* derivation of the input string. So we have parsed analyzing the string from left to right, constructing a rightmost derivation. This type of parsing algorithms is called LR-parsing, where the L stands for reading the input from left-to-right and the R stands for producing the rightmost derivation.

The decisions above are based on the suffix (i.e., top) of the stack on the left-hand side and the first token lookahead on the right-hand side. Here, the suffix of the stack on the left-hand side must be a *prefix substring* of a grammar production. If not, it would be impossible to complete it in such a way that a future grammar production can be applied in a reduction step: the parse attempt is doomed to failure.

3 LR(1) Parsing Tables

We could now define again a slightly different version of $\mathsf{follow}(\gamma, \mathtt{a})$, where γ is a prefix substring of the grammar or a non-terminal, and then specialize the rules. An alternative, often used to describe parser generators, is to construct a *parsing table*. For an LR(1) grammar, this table contains an entry for every prefix substring of the grammar and every token seen on the input. An entry describes whether to shift, reduce (and by which rule), or to signal an error. If the action is ambiguous, the given grammar is not LR(1), and either an error message is issued, or some default rule comes into effect that choses between the option.

We now construct the parsing table, assuming $+ < *$, that is, multiplication binds more tightly than addition. Moreover, we specify that both addition and multiplication are *left associative* so that, for example, `3 + 4 + 5` should be parsed as `(3 + 4) + 5`. We have removed id since it behaves identically to num.

$$
\begin{array}{lrcl}
[\mathsf{plus}] & E & \longrightarrow & E \ \mathtt{+} \ E \\
[\mathsf{times}] & E & \longrightarrow & E \ \mathtt{*} \ E \\
[\mathsf{number}] & E & \longrightarrow & num \\
[\mathsf{parens}] & E & \longrightarrow & \mathtt{(} \ E \ \mathtt{)}
\end{array}
$$

As before, we assume that a special end-of-file token `$` has been added to the end of the input string. When the parsing goal has the form $\gamma\,\beta \mid \mathtt{a}\,w$ where β is a prefix substring of the grammar, we look up β in the left-most column and `a` in the top row to find the action to take. The non-terminal $\epsilon\,E$ in the last line is a special case in that E must be the only thing on the

stack. In that case we can accept if the next token is `$` because we know that `$` can only be the last token of the input string.

$\gamma\,\beta \mid \mathtt{a}\,w$	+	$\star$	num	(	)	$
E + E	reduce(plus) (+ left assoc.)	shift ($+ < \star$)	error	error	reduce(plus)	reduce(plus)
$E \star E$	reduce(times) ($+ < \star$)	reduce(times) ($\star$ left assoc.)	error	error	reduce(times)	reduce(times)
num	reduce(number)	reduce(number)	error	error	reduce(number)	reduce(number)
(E)	reduce(parens)	reduce(parens)	error	error	reduce(parens)	reduce(parens)
E +	error	error	shift	shift	error	error
$E \star$	error	error	shift	shift	error	error
(E	shift	shift	error	error	shift	error
(	error	error	shift	shift	error	error
$\epsilon\,E$	shift	shift	error	error	error	accept(E)

We can see that the bare grammar has four shift/reduce conflicts, while all other actions (including errors) are uniquely determined. These conficts arise when E + E or $E \star E$ is on the stack and either + or $\star$ is the first character in the remaining input string. It is called a shift/reduce conflict, because either a shift action or a reduce action could lead to a valid parse. Here, we have decided to resolve the conflicts by giving a precedence to the operators and declaring both of them to be left-associative.

It is also possible to have reduce/reduce conflicts, if more than one reduction could be applied in a given situation, but it does not happen in this grammar. A simple illustration where reduce/reduce conflicts can possibly show is when you have add a test statement with natural equation notation in your language.

$$\begin{array}{lcl} S & \longrightarrow & A; \\ S & \longrightarrow & T? \\ A & \longrightarrow & id = E \\ T & \longrightarrow & id = E \end{array}$$

In this grammar, `id=E` can be reduced to an assignment (A) or to a test statement (T). Both reductions are possible. Here, it's easy to distinguish between both cases based on taking the follow sets into account. A can only be followed by ";" and test can only be followed by "?", which makes it unambiguous at the time of reduction here with a 1 token lookahead.

Parser generators will generally issue an error or warning when they detect a shift/reduce or reduce/reduce conflict. For many parser generators, the default behavior of a shift/reduce conflict is to shift, and for a

reduce/reduce conflict to apply the textually first production in the grammar. Particularly the latter is almost never what is desired, so we strongly recommend rewriting the grammar to eliminate any conflicts in an LR(1) parser.

One interesting special case is the situation in a language where the else-clause of a conditional is optional. For example, one might write (among other productions)

$$\begin{array}{lcl} S & \longrightarrow & IS \\ S & \longrightarrow & E \\ E & \longrightarrow & \texttt{ID} \\ IS & \longrightarrow & \textsf{if } E \textsf{ then } S \\ IS & \longrightarrow & \textsf{if } E \textsf{ then } S \textsf{ else } S \end{array}$$

Now a statement

```
if (b) then if (c) then x else y
```

is ambiguous because it could be read as

```
if (b) then (if (c) then x) else y
```

or

```
if (b) then (if (c) then x else y)
```

In fact, LR parsers will report a shift/reduce conflict for this grammar.

In a shift/reduce parser, typically the default action for a shift/reduce conflict is to shift. This means that the above grammar in a tool such as ML-Lex will parse the ambiguous statement into the second form, that is, the `else` is match with the most recent unmatched `if`. This is consistent with language such as C (or the language used in this course), so we can tolerate the above shift/reduce conflict, if you wish, instead of rewriting the grammar to make it unambiguous. Yet, this is a dangerous business, because we depend on the mercy of the LR parser generator to hopefully choose the right conflict resolutions for us.

```
state 7 contains 1 shift/reduce conflict.
...
state 7
  SI -> if E then S . (rule 4)
  IS -> if E then S . else S (rule 5)
  else       shift, and go to state 8
  else       [reduce using rule 4 (IS)]
  $default reduce using rule 4 (IS)
```

How else can we fix the conflict? The easiest and best fix is to change the language and require some form of an `end if` token. This solution will also make the subtlety transparent to the user, because there can no longer be surprises in how the ambiguity is resolved. The problem is that this changes the language. The other fix is to disambiguate the grammar by factorization.

$$
\begin{array}{lcl}
S & \longrightarrow & IS \\
S & \longrightarrow & E \\
E & \longrightarrow & \texttt{ID} \\
IS & \longrightarrow & \textsf{if } E \textsf{ then } S \\
IS & \longrightarrow & \textsf{if } E \textsf{ then } ITE \textsf{ else } S \\
ITE & \longrightarrow & \textsf{if } E \textsf{ then } ITE \textsf{ else } ITE \\
ITE & \longrightarrow & E
\end{array}
$$

The effect of this factorization is that the then part of an if-then-else statement can no longer be a single if-then statement. It has to be a full if-then-else or not an if-statement at all.

4 Implementations of LR Parser Generators

LR parser generators are quite tricky to implement. A full coverage of how they work would need several lectures. The principle behind actual implementations of LR parser generators is to define a nondeterministic (pushdown) automaton called characteristic automaton that follows all possible parse positions. This characteristic automaton is then turned into an actual implementation by converting it into a deterministic (pushdown) automaton using the subset construction. This automaton will follow all possible analysis situations until a reduce becomes evident. There are several similarities between lexer generators (previous lectures) and LR parser generators. In fact, lexers can actually be generated using a special case of LR grammars called LALR(0), where we leave out the stack so that every shifted character has to be reduced immediately.

Full LR(1) parsers also have too many states. That is why subclasses of LR(1) are often used for the parsing situations, with a set of states restricted to those of LR(0).

Quiz

1. What happens if we remove the ϵ from the last entry in the LR parser table? Aren't ϵ's irrelevant and can always be removed?

2. What makes `x*y;` difficult to parse in C and C0? Should this be handled in the lexing? How can this be handled in the parser? How can this be handled in later stages after parsing?

3. Give a very simple example of a grammar with a shift/reduce conflict.

4. Give an example of a grammar with a shift/reduce conflict that occurs in programming language parsing and is not easily resolved using associativity/precedence of arithmetic operators.

5. Give a very simple example of a grammar with a reduce/reduce conflict.

6. Give an example of a grammar with a reduce/reduce conflict that occurs in programming language parsing and is not easily resolved.

7. Suppose you want to add a short hand notation for asserts into your programming language. Should a programming language have a test statement of the form `?E;` or of the form `E?;` and which one is better for which purpose?

8. In the reduce rule, we have used a number of symbols on the top of the stack and the lookahead to decide what to do. But isn't a stack something where we can only read one symbol off of the top? Does it make a difference in expressive power if we allow decisions to depend on 1 or on 10 symbols on the top of the stack? Does it make a difference in expressive power if we allow 1 or arbitrarily many symbols from the top of the stack for the decision?

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[SSP95] Stuart M. Shieber, Yves Schabes, and Fernando C. N. Pereira. Principles and implementation of deductive parsing. *J. Log. Program.*, 24(1&2):3–36, 1995.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.