Lecture Notes on Predictive Parsing

15-411: Compiler Design
Frank Pfenning*

Lecture 9
September 24, 2013

1 Introduction

In this lecture we discuss two parsing algorithms, both of which traverse the input string from left to right. The first, LL(1), makes a decision on which grammar production to use based on the first character of the input string. If that were ambiguous, the grammar would have to be rewritten to fall into this class, which is not always possible. The second, LR(1), 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. It is variations on the latter which are typically used in modern parser generation tools.

Alternative presentations of the material in this lecture can be found in the textbook [App98, Chapter 3] and a paper by Shieber et al. [SSP95].

2 LL(1) Parsing

We have seen in the previous section, that the general idea of recursive descent parsing without restrictions forces us to non-deterministically choose between several productions which might be applied and potentially backtrack if parsing gets stuck after a choice, or even loop (if the grammar is left-recursive). Backtracking is not only potentially very inefficient, but it makes it difficult to produce good error messages in case the string is not grammatically well-formed. Say we try three different ways to parse a given term and all fail. How could we say which of these is the source of the error? This is compounded because nested choices multiply the

*with contributions by André Platzer

number of possibilities. We therefore have to look for ways to disambiguate the choices.

One way is to *require* of the grammar that at each potential choice point we can look at the next input token and based on that token decide which production to take. This is called *1 token lookahead*, and grammars that satisfy this restriction are called *LL(1)* grammars. Here, the first *L* stands for *L*eft-to-right; the second *L* stands for *L*eftmost parse (which a recursive descent parser generates) and *1* stands for *1 token lookahead*. Potentially, we could also define LL(2), LL(3), etc., but these are of limited practical utility.

Since we are restricting ourselves to parsing by a left-to-right traversal of the input string, we will consider only tails, or postfixes of the input strings, and also of the strings in the grammar, when we restrict our inference rules. For short, we will say γ is a *postfix substring* of the grammar, or w is a postfix substring of the input string w_0. For example, in the grammar

$$\begin{array}{lrcl} [\mathsf{emp}] & S & \longrightarrow & \\ [\mathsf{pars}] & S & \longrightarrow & \texttt{[}\, S\, \texttt{]} \\ [\mathsf{dup}] & S & \longrightarrow & S\, S \end{array}$$

the only postfix substrings are ϵ, $\texttt{[}\, S\, \texttt{]}$, $S\, \texttt{]}$, $\texttt{]}$, S, and $S\, S$, but not $\texttt{[}\, S$.

We begin be defining two kinds of predicates (later we will have occasion to add a third), where β is either a non-terminal or postfix substring of the grammar.

$\mathsf{first}(\beta, \mathtt{a})$	Token $\mathtt{a}$ can be first in string β
$\mathsf{null}(\beta)$	String β can produce the empty string ϵ

These predicates must be computed entirely statically, by an analysis of the grammar before any concrete string is ever parsed. This is because we want to be able to tell if the parser can do its work properly with 1 token look-ahead regardless of the string it has to parse.

We define the relation $\mathsf{first}(\beta, \mathtt{a})$ by the following rules.

$$\frac{}{\mathsf{first}(\mathtt{a}\, \beta, \mathtt{a})}\ F_1$$

This rule *seeds* the first predicate. Then is it propagated to other strings appearing in the grammar by the following three rules.

$$\frac{\mathsf{first}(X, \mathtt{a})}{\mathsf{first}(X\ \beta, \mathtt{a})}\ F_2 \qquad \frac{\mathsf{null}(X) \quad \mathsf{first}(\beta, \mathtt{a})}{\mathsf{first}(X\ \beta, \mathtt{a})}\ F_3 \qquad \frac{\begin{array}{c}[r] X \longrightarrow \gamma \\ \mathsf{first}(\gamma, \mathtt{a})\end{array}}{\mathsf{first}(X, \mathtt{a})}\ F_4(r)$$

Even though ϵ may be technically a postfix substring of every grammar, it can never arise in the first argument of the first predicate. The auxiliary predicate null is also

easily defined.

$$\frac{}{\mathsf{null}(\epsilon)}\ N_1 \qquad \frac{\mathsf{null}(X) \quad \mathsf{null}(\gamma)}{\mathsf{null}(X\,\gamma)}\ N_2 \qquad \frac{\begin{array}{l}[r]X \longrightarrow \gamma \\ \mathsf{null}(\gamma)\end{array}}{\mathsf{null}(X)}\ N_3$$

We can run these rules to saturation because there are only $O(|G|)$ possible strings in the first argument to both of these predicates, and at most the number of possible terminal symbols in the grammar, $O(|\Sigma|)$, in the second argument. Naive counting the number of prefix firings (see [GM02]) gives a complexity bound of $O(|G| \times |\Xi| \times |\Sigma|)$ where $|\Xi|$ is the number of non-terminals in the grammar. Since usually the number of symbols is a small constant, this is roughly equivalent to $O(|G|)$ and so is reasonably efficient. Moreover, it only happens once, before any parsing takes place.

Next, we modify the rules for recursive descent parsing from the last lecture to take these restrictions into account. The first two stay the same.

$$\frac{}{\epsilon : \epsilon}\ R_1 \qquad \frac{w : \gamma}{\mathtt{a}\,w : \mathtt{a}\,\gamma}\ R_2$$

The third,

$$\frac{\begin{array}{l}[r]X \longrightarrow \beta \\ w : \beta\,\gamma\end{array}}{w : X\,\gamma}\ R_3(r)$$

is split into two, each of which has an additional precondition.

$$\frac{\begin{array}{l}[r]X \longrightarrow \beta \\ \mathsf{first}(\beta, \mathtt{a}) \\ \mathtt{a}\,w : \beta\,\gamma\end{array}}{\mathtt{a}\,w : X\,\gamma}\ R_3' \qquad \frac{\begin{array}{l}[r]X \longrightarrow \beta \\ \mathsf{null}(\beta) \\ w : \beta\,\gamma\end{array}}{w : X\,\gamma}\ R_3''?$$

We would like to say that a grammer is LL(1) if the additional preconditions in these last two rules make all choices unambiguous when an arbitrary non-terminal X is matched against a string starting with an arbitrary terminal $\mathtt{a}$. Unfortunately, this does not quite work yet in the presence non-terminals that can rewrite to ϵ, because the second rule above does not even look at the input string.

To further refine this we need one additional predicate, again on postfix strings in the grammar and non-terminals.

$\mathsf{follow}(\beta, \mathtt{a})$ Token $\mathtt{a}$ can follow string β in a valid string

We seed this relation with the rules

$$\frac{\begin{array}{l} X\,\gamma\ \mathit{postfix} \\ \mathsf{first}(\gamma, \mathtt{a}) \end{array}}{\mathsf{follow}(X, \mathtt{a})}\ W_1$$

Here, $X\,\gamma$ *postfix* means that the string $X\,\gamma$ appears as a postfix substring on the right-hand side of a production. We then propagate this information applying the following rules from premises to conclusion until saturation is reached.

$$\frac{\mathsf{follow}(\mathtt{b}\,\gamma, \mathtt{a})}{\mathsf{follow}(\gamma, \mathtt{a})}\ W_2 \qquad \frac{\mathsf{follow}(X\,\gamma, \mathtt{a})}{\mathsf{follow}(\gamma, \mathtt{a})}\ W_3 \qquad \frac{\begin{array}{l} \mathsf{follow}(X\,\gamma, \mathtt{a}) \\ \mathsf{null}(\gamma) \end{array}}{\mathsf{follow}(X, \mathtt{a})}\ W_4 \qquad \frac{\begin{array}{l} [r]X \longrightarrow \gamma \\ \mathsf{follow}(X, \mathtt{a}) \end{array}}{\mathsf{follow}(\gamma, \mathtt{a})}\ W_5$$

The first argument here should remain a non-empty postfix or a non-terminal. Now we can refine the proposed R_3'' rule from above into one which is much less likely to be ambiguous.

$$\frac{\begin{array}{l} [r]X \longrightarrow \beta \\ \mathsf{first}(\beta, \mathtt{a}) \\ \mathtt{a}\,w : \beta\,\gamma \end{array}}{\mathtt{a}\,w : X\,\gamma}\ R_3' \qquad \frac{\begin{array}{l} [r]X \longrightarrow \beta \\ \mathsf{null}(\beta) \\ \mathsf{follow}(X, \mathtt{a}) \\ \mathtt{a}\,w : \beta\,\gamma \end{array}}{\mathtt{a}\,w : X\,\gamma}\ R_3''$$

We avoid creating an explicit rule to treat the empty input string by appending a special $ symbol at the end before starting the parsing process. We repeat the remaining rules for completeness.

$$\frac{}{\epsilon : \epsilon}\ R_1 \qquad \frac{w : \gamma}{\mathtt{a}\,w : \mathtt{a}\,\gamma}\ R_2$$

These rules are interpreted as a parser by proof search, applying them from the conclusion to the premise. We say the grammar is LL(1) if for any goal $w : \gamma$ at most one rule applies. If X cannot derive ϵ, this amounts to checking that there is at most one production $X \longrightarrow \beta$ such that $\mathsf{first}(\beta, \mathtt{a})$. For nullable non-terminals the condition is slightly more complicated, but can still easily be read off from the rules.

We now use a very simple grammar to illustrate these rules. We have transformed it in the way indicated above, by assuming a special token $ to indicate the end of the input string.

$$\begin{array}{lcll} [\mathsf{start}] & S & \longrightarrow & S'\,\$ \\ [\mathsf{emp}] & S' & \longrightarrow & \epsilon \\ [\mathsf{pars}] & S' & \longrightarrow & [\,S'\,] \end{array}$$

This generates all string starting with an arbitrary number of opening parentheses followed by the same number of closing parentheses and an end-of-string marker.

We have:

$\mathsf{null}(\epsilon)$	N_1
$\mathsf{null}(S')$	N_3
$\mathsf{first}(\texttt{[}\, S' \,\texttt{]}, \texttt{[})$	F_1
$\mathsf{first}(\texttt{]}, \texttt{]})$	F_1
$\mathsf{first}(S' \,\texttt{]}, \texttt{]})$	F_3
$\mathsf{first}(S', \texttt{[})$	F_4 [pars]
$\mathsf{first}(S' \,\texttt{]}, \texttt{[})$	F_2
$\mathsf{first}(\$, \$)$	F_1
$\mathsf{first}(S' \,\$, \$)$	F_3
$\mathsf{first}(S' \,\$, \texttt{[})$	F_2
$\mathsf{first}(S, \$)$	F_4 [start]
$\mathsf{first}(S, \texttt{[})$	F_4 [start]
$\mathsf{follow}(S', \$)$	W_1
$\mathsf{follow}(S', \texttt{]})$	W_1
$\mathsf{follow}(\texttt{[}\, S' \,\texttt{]}, \$)$	W_5
$\mathsf{follow}(\texttt{[}\, S' \,\texttt{]}, \texttt{]})$	W_5
$\mathsf{follow}(S' \,\texttt{]}, \$)$	W_3
$\mathsf{follow}(S' \,\texttt{]}, \texttt{]})$	W_3
$\mathsf{follow}(\texttt{]}, \$)$	W_4
$\mathsf{follow}(\texttt{]}, \texttt{]})$	W_4

3 Parser Generation

Parser generation is now a very simple process. Once we have computed the null, first, and follow predicates by saturation from a given grammar, we specialize the inference rules $R_3'(r)$ and $R_3''(r)$ by matching the first two and three premises against grammar productions and saturated database. In this case, this leads to the following specialized rules (repeating once again the two initial rules).

$$\frac{}{\epsilon : \epsilon}\ R_1 \qquad \frac{w : \gamma}{\mathtt{a}\, w : \mathtt{a}\, \gamma}\ R_2$$

$$\frac{\texttt{[}\, w : S' \,\$\, \gamma}{\texttt{[}\, w : S\, \gamma}\ R_3'(\mathsf{start}) \qquad \frac{\$\, w : S' \,\$\, \gamma}{\$\, w : S\, \gamma}\ R_3'(\mathsf{start})$$

$$\frac{\texttt{[}\, w : \texttt{[}\, S' \,\texttt{]}\, \gamma}{\texttt{[}\, w : S'\, \gamma}\ R_3'(\mathsf{pars}) \qquad \frac{\texttt{]}\, w : \gamma}{\texttt{]}\, w : S'\, \gamma}\ R_3''(\mathsf{emp}) \qquad \frac{\$\, w : \gamma}{\$\, w : S'\, \gamma}\ R_3''(\mathsf{emp})$$

Recall that these rules are applied from the bottom-up, starting with the goal $w_0\,\texttt{\$} : S$, where w_0 is the input string. It is easy to observe by pattern matching that each of these rules are mutually exclusive: if one of the applies, none of the other rules applies. Moreover, each rule except for R_1 (which accepts) has exactly one premise, so the input string is traversed linearly from left-to-right, without backtracking. When none of the rules applies, then the input string is not in the language defined by the grammar. This proves that our simple language $\texttt{(}^n\texttt{)}^n$ is LL(1).

Besides efficiency, an effect of this approach to parser generation is that it supports good error messages in the case of failure. For example, if we see the parsing goal $\texttt{(}\,w\,:\,\texttt{)}\,\gamma$ we can state: *Found '(' while expecting ')'.*, and similarly for other cases that match none of the conclusions of the rules.

4 Removing Ambiguities

One standard way to deal with ambiguities in grammars is to rewrite them, but under the constraint that they accept the same strings. When designing our own programming language, we sometimes have the immense luxury to actually change the syntax to make it easier to parse (and, hopefully, also easier to read and understand).

As an example, we use the following simple grammar for expressions.

[plus]	E	$\longrightarrow$	E + E
[times]	E	$\longrightarrow$	E * E
[ident]	E	$\longrightarrow$	id
[number]	E	$\longrightarrow$	num
[parens]	E	$\longrightarrow$	(E)

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

In order the rewrite it to make the parse tree unambiguous we have to analyze how to *rule out* the unintended parse tree. In the expression `3 + 4 * 5` we have to all the parse equivalent to `3 + (4 * 5)` but we have to *rule out* the parse equivalent to `(3 + 4) * 5`. In other words, the left-hand side of a product is *not allowed to be a sum* (unless it is explicitly parenthesized).

Backing up one step, how about `3 + 4 + 5`? We want addition to be *left associative*, so this should parse as `(3 + 4) + 5`. In other words, we have to *rule out* the parse `3 + (4 + 5)`. Instead of

$$E \longrightarrow E + E$$

we want

$$E \longrightarrow E + P$$

where P is a new nonterminal that does not allow a sum. Continuing the above thought, P *is* allowed to be a product, so we proceed

$$P \longrightarrow P * A$$

Since multiplication is also left-associative, we have made up a new symbol A which cannot be a product. In fact, in our language A can only be an identifier, a number, or a parenthesized (arbitrary) expression.

[plus]	E	$\longrightarrow$	E + P
[times]	P	$\longrightarrow$	P * A
[ident]	A	$\longrightarrow$	id
[number]	A	$\longrightarrow$	num
[parens]	A	$\longrightarrow$	(E)

This is not yet complete, because it is in fact empty: it claims an expression must always be a sum. But it could also just be a product. Similarly, products P may just consist of an atom A. This yields:

[plus]	E	$\longrightarrow$	E + P
[e/p]	E	$\longrightarrow$	P
[times]	P	$\longrightarrow$	P * A
[p/a]	P	$\longrightarrow$	A
[ident]	A	$\longrightarrow$	id
[number]	A	$\longrightarrow$	num
[parens]	A	$\longrightarrow$	(E)

You should convince yourself that this grammar is now unambiguous. Unfortunately, it is not LL(1): from the first token we cannot tell which grammar production to choose. In fact any token can start *any production*!

5 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. The example in the previous section illustrate this: it was relatively easy to rewrite the grammar to be unambiguous, but we need much more work to make it LL(1).

We can react by rewriting the grammar, at significant expense of readability, or we could just specify that (a) addition and multiplication are left-associative, and (b) multiplication has higher precedence than addition, + $<$ *. 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 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

$$\frac{}{S \mid \epsilon}\, R_1$$
$$\vdots$$
$$\epsilon \mid w_0$$

Parsing is defined by the following rules:

$$\frac{}{S \mid \epsilon}\, R_1(= \mathsf{accept})$$

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

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

num	$\mid$	+ num * num	?
ϵ	$\mid$	num + num * num	shift

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.

E	$\mid$	+ num * num	?
num	$\mid$	+ num * num	reduce(number)
ϵ	$\mid$	num + num * num	shift

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

E + E	$\mid$	* num	
E +num	$\mid$	* num	reduce(number)
E +	$\mid$	num * num	shift
E	$\mid$	+ num * num	shift
num	$\mid$	+ num * num	reduce(number)
ϵ	$\mid$	num + num * num	shift

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

E	ϵ	accept
E + E	ϵ	reduce(plus)
E + E * E	ϵ	reduce(times)
E + E * num	ϵ	reduce(number)
E + E *	num	shift
E + E	* num	shift
E +num	* num	reduce(number)
E +	num * num	shift
E	+ num * num	shift
num	+ num * num	reduce(number)
ϵ	num + num * num	shift

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 left-to-right and the R stands for rightmost.

The decisions above are based on the postfix of the stack on the left-hand side and the first token on the right-hand side. Here, the postfix 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.

6 LR(1) Parsing Tables

We could now define again a slightly different version of follow(γ, 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 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 chooses between the options.

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.

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

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 $\gamma\,\beta \mid \texttt{a}\,w$ 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.

$\beta \setminus$ a	+	*	num	(	)	$
E + E	reduce(plus) (+ left assoc.)	shift (+ $<$ *)	error	error	reduce(plus)	reduce(plus)
E * E	reduce(times) (+ $<$ *)	reduce(times) (* 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 *	error	error	shift	shift	error	error
(E	shift	shift	error	error	shift	error
(	error	error	shift	shift	error	error
ϵ	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 * E is on the stack and either + or * 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.

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 rarely 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)

$$\begin{array}{lcl} E & \longrightarrow & \mathsf{if}\ E\ \mathsf{then}\ E \\ E & \longrightarrow & \mathsf{if}\ E\ \mathsf{then}\ E\ \mathsf{else}\ E \end{array}$$

Now a statement

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

is ambiguous because it would be read as

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

or

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

In a shift/reduce parser, typically the default action for a shift/reduce conflict is to shift to extend the current parse as much as possible. This means that the above grammar in a tool such as ML-Yacc 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 C0, 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.

We can also think about how to rewrite the grammar so it is unambiguous. What we have to so is rule out the parse

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

In other words, the *then* clause of a conditional should be balanced in terms of if-then-else and not have something that is just an if-then without an *else* clause.

$$\begin{array}{lcl} E & \longrightarrow & \mathsf{if}\ E\ \mathsf{then}\ E \\ E & \longrightarrow & \mathsf{if}\ E\ \mathsf{then}\ E'\ \mathsf{else}\ E \\ \\ E' & \longrightarrow & \mathsf{if}\ E\ \mathsf{then}\ E'\ \mathsf{else}\ E' \\ E' & \longrightarrow & \ldots \end{array}$$

We would also have to repeat all the other clauses for E, or refactor the grammar so the other productions of E can be shared with E'.

Questions

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? Discuss some possible solutions, once you have identified a problem?
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 or 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. 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?
8. What's wrong with this grammar that was meant to define a program P as a sequence of statements S by $P \to S \mid P; P$

References

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

[GM02] Harald Ganzinger and David A. McAllester. Logical algorithms. In P.Stuckey, editor, *Proceedings of the 18th International Conference on Logic Programming*, pages 209–223, Copenhagen, Denmark, July 2002. Springer-Verlag LNCS 2401.

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