Computer Science 131, Spring 2002
Assignment 2: Syntax
(Sample Solution)

Chris Stone

Monday, February 11

1. *In class you were given a copy of the grammar for the concrete syntax of the C language, taken from K&R. Is this grammar LL(1)? Explain why or why not.*

   This grammar is not LL(1), for numerous reasons. Not only are many rules left-recursive, but the grammar is not even left-factored. For example, the *selection-statement* nonterminal has two production rules that start with the token `if`, the *relational-expression* nonterminal has four rules that each begin with the same nonterminal *relational-expression*, *parameter-specifiers* has two rules starting with the nonterminal *declaration-specifiers*, and so on. In any of these cases (and all the others) multiple right-hand sides can therefore start with the same terminal.

   (However, this grammar is nearly LALR(1) and hence would be a reasonable starting point for some automatic parser-generating tools such as Yacc.)

2. *Consider the following grammar for regular expressions:*

$$\begin{array}{rl} R \rightarrow & \epsilon \\ | & \mathtt{a} \mid \mathtt{b} \mid \mathtt{c} \mid \ldots \\ | & R\,R \\ | & R + R \\ | & R^* \end{array}$$

   As shown, derivations in the this grammar might be a reasonable representation of an abstract syntax for regular expressions (i.e., trees in this grammar can be used to represent regular expressions). However, this grammar is less satisfactory as a definition for a concrete syntax of regular expressions.

   (a) *Show that the above grammar is ambiguous by providing a string which can be parsed in more than one way.*

   The regular expression $\mathtt{a} + \mathtt{b}^*$ can be parsed as meaning $(\mathtt{a} + \mathtt{b})^*$ or as meaning $\mathtt{a} + (\mathtt{b}^*)$. Similarly, the regular expression `abc` can be parsed as meaning `(ab)c` or as meaning `a(bc)`. There are many other examples.

(b) *Give an unambiguous grammar for a concrete syntax of regular expressions, subject to the following constraints:*

- *The concrete syntax should permit parenthesized regular expressions*
- *Star should bind more tightly than concatenation, which should bind more tightly than +.*
- *Concatenation should be left-associative, but + should be right-associative. (For regular expressions the choice of associativity doesn't affect the meaning at all, but we need to make* some *choice in order for the grammar to be unambiguous.)*

Your grammar need not be LL(1).

We can construct such a grammar by having a nonterminal for each level of precedence in the language, as was done in the C grammar and the unambiguous grammars for arithmetic discussed in class the day that grammers were first introduced.

$$
\begin{array}{rll}
R \rightarrow & T & \text{// A regexp is a sequence of } T\text{'s connected by } + \\
\mid & T + R & \text{//} \quad \text{(grouped to the right)} \\
T \rightarrow & V & \text{// A } T \text{ is a sequence of } V\text{'s} \\
\mid & T\ V & \text{//} \quad \text{(grouped to the left)} \\
V \rightarrow & W & \text{// Each } V \text{ is a } W \text{ with zero or more stars} \\
\mid & V^* & \\
W \rightarrow & \epsilon & \text{// And } W\text{'s are ``atomic'' regular expressions} \\
\mid & \texttt{a} \mid \ldots & \\
\mid & (R) &
\end{array}
$$

3. (a) *Consider the grammar*

$$
\begin{array}{rll}
S \rightarrow & E\ \$ & \\
E \rightarrow & T & \\
\mid & E + T & \\
T \rightarrow & n & \textit{(where } n \textit{ represents any integer constant)}
\end{array}
$$

This grammar is not LL(1). Is it LL(k) for any integer k? Carefully justify your answer.

Since the grammars is left-recursive, it can't be LL(k). But given any sequence in this language, we can tell whether it's a T or not with two tokens of lookahead (i.e., by looking to see if the second character is a minus sign). But all this tells us is that a predictive parser can always make its *first* descision correctly with two tokens of lookahead; each successive decision will require more lookahead.

Suppose the input were $0-0-\cdots-0$ where there are n subtractions. A predictive parser would have to start by making n predictions before the first match: $E \rightarrow E - T \rightarrow E - T - T \rightarrow \ldots \rightarrow T - T - \cdots T$ (where the last prediction has n subtractions). At this point the parser could start predicting the T's and

matching the numerals and subtraction signs. However, getting this far required us to know that we needed to make n predictions of subtraction, and there's no way to figure out how many subtractions appear in the input with a constant amount of lookahead.

Or, consider the recursive descent implementation of this grammar. The function for E would check the input, decide whether it's a single T, and if not, take this same input and recursively look for an E, then look for a subtraction sign, and then look for a T. Clearly this will go into an infinite loop, as the E function does nothing but call itself with the same input sequence of tokens.

(b) *Consider the grammar*

$$\begin{array}{lll} S \rightarrow & E\ \$ & \\ E \rightarrow & T & \\ \quad | & E + T & \\ \quad | & E - T & \\ T \rightarrow & n & \textit{(where } n \textit{ represents any integer constant)} \end{array}$$

This grammar is also not LL(1). Is it LL(k) for any integer k? Carefully justify your answer.

This is more obviously non-LL(1); the same arguments apply as in the previous case, but we can also observe that if the input is not a T then we must decide whether it's an expression ending in a subtraction and a T or ending in an addition and a T. Since any expression can end with a subtraction or an addition, it's impossible to figure out how the input is going to end with just a constant number of tokens at the beginning.

4. *Several designers of the programming language Dylan came from the Lisp community, and thus Dylan started out life with a concrete syntax very similar to that of Lisp and Scheme (using S-expressions and prefix notation), e.g.,*

```
(define-method double (x)
   (* 2 x))
```

Later an alternate concrete *"infix" syntax was developed, closer to that of most imperative programming languages:*

```
define method double(x)
  2 * x;
end method;
```

The plan was that the language would have a single abstract syntax, but two different concrete syntaxes. Programmers could choose whether to write files of code in prefix-style or infix-style; the compiler wouldn't care, since source programs in either syntax

could be mapped the same set of internal abstract syntax trees. If desired, automated tools could easily convert code from one syntax to the another.

Should programming languages support more than one concrete syntax? *Answer this question in three paragraphs. In the first paragraph, argue as strongly as possible why one might want to have a language with multiple concrete syntaxes; in the second paragraph, argue as strongly as possible why it might be a bad idea. Finally, explain which side you personally find most convincing.*

This will be discussed in class, but here are just a few possibilities:

- Pro:
  - Having multiple syntaxes could make it easier to learn the language for people of differening backgrounds.
  - Different programmers could code in their preferred style while still using the same language (and hence having code guaranteed to interoperate with code written by other people with different preferences).
- Con:
  - Can be harder to read or modify other people's code; to be able to read code in this language must either "know" all the syntaxes or else have a tool that converts syntax automatically.
  - Tools (compilers, syntax highlighters, etc.) may have to know about all the different syntaxes.
  - Macros might not be portable across syntaxes, since they often involve "pieces" of concrete syntax that might not make sense in isolation.

Since the last part asks for an opinion, obviously there's no right answer.