Appendix E

Answers

Chapter 1—Preliminaries

1. The preceding example has only two unique names x and y. Using colored pens or some other convenient device, identify all of the preactive, active and postactive regions. There are a total of $3 \times 4 = 12$ such regions.

```plaintext
x := 0; ` outer variable (level 0)
be x y. ` level 1 nomenclature for x and y
y := x; ` x on level 0 and y on level 1
be x.
   x, y := y+1, 4; ` level 2 nomenclature for x
   y := y+x;
   eb; ` ditto
   x := y;
   eb ` end of level 2
   x := y; ` x on level 0, y on level 1
   eb ` end of level 1
```

2. Install Hyper.1

*Installation varies from platform to platform. For Unix systems it comes as a “shar” file. On the Macintosh it is a folder. In every case there is a README file on the distribution medium which gives a step-by-step installation procedure. One invariant is the need for a C compiler on the system.*

3. Run each of the examples above in Hyper.

*They are all available in machine-readable form in the distribution package. There are no surprises. Each of the three examples is correct and gives the predicted output.*

4. Write the optimized program (suggested above) to find right triangles.

---

1See distribution package instructions.
APPENDIX E. ANSWERS

\begin{verbatim}
x, y := side1, side2; ` input
if x < y -> x, y := y, x
:: else
fi;

z, result := x - y, 0;
x2, y2, z2 := x^x, y^y, z^z; ` z and z2 vary
x2my2 := x^2 - y^2;
x2py2 := x^2 + y^2;

it
  if x2my2 = z2 /
     x2py2 = z2 -> result := z
  :: else
     fi;

  if z < x + y /
     result = 0 ->
      z2 := z2 + z + z + 1; ` (z+1)*(z+1) = z*z+2*z+1
      z := z + 1
     :: else
     exit
     fi;
ti;

side3 := result ` output
\end{verbatim}

5. Write another interesting program in X. Run it in HYPER.

\begin{verbatim}
x := 2; ` start doubling
x := x*x;
  ` 2^2 = 4
x := x*x;
  ` 2^4 = 16
x := x*x;
  ` 2^8 = 256
x := x*x;
  ` 2^{16} = 65536
x := x*x; ` overflow?
\end{verbatim}

6. [1, 1] Is size(2^A) = 2^{size(A)}?

Yes, for example
size(2^{a,b}) = size({{}, {a}, {b}, {a, b}}) = 4 = 2^2 = 2^{size({a, b})}

7. Is max(A) ≥ choice(A)?

Yes, if A is ordered and nonempty.

8. Is max(A) ≤ max(A ∪ B)?

Yes, if min and choice are defined.
9. Is \((A - B) \cap B = \{\}\)?

\[
(A - B) \cap B = (A \cap B) \cap B = A \cap (B \cap B) = A \cap B = \{\}
\]

10. Is \(A = \{\} \cup \text{choice}(A) \in A\)?

The equivalent \(A \neq \{\} \Rightarrow \text{choice}(A) \in A\) is a more intuitive form.

11. Is \(\text{size}(A) = 0 \lor \text{size}(A) - 1 = \text{size}(A - \{\text{choice}(A)\})\)?

This can stand as an iterative definition of size:

\[
\text{size}(A) \overset{\text{def}}{=} \begin{cases} 
A = \{\} & 0 \\
A \neq \{\} & \text{size}(A - \{\text{choice}(A)\}) + 1
\end{cases}
\]

12. Is \((V^*)^{1/\ast} = V\)?

As an intermediate step, show \(V^1 \subseteq V^*\) and \((V^1)^{1/\ast} = V\).

13. What is \(\{\}\)\(^*\)?

\(\{\lambda\}\)

14. What is \(\{\}\)\(^+\)?

\(\{\}\)

15. What is \(\text{size}(\{a\})\)?

\(\text{size}(\{aaaaaaa\}) = 1\)


\[
\text{length}(s) \overset{\text{def}}{=} \begin{cases} 
s = \lambda & 0 \\
s = \langle x, y \rangle & \text{length}(y) + 1
\end{cases}
\]

17. Let \(P\) be the universe of living people, \(M\) be all men, \(W\) be all women. \(P = M \cup W\). Let \(\text{Married} \subseteq M \times W\).

(a) What is \(\mathcal{D}(\text{Married})\) and \(\mathcal{R}(\text{Married})\)?

\(\text{husbands and wives}\).

(b) What do you call \(x\) where \(\text{size}(\mathcal{R}(\{x\} \triangle \text{Married})) > 1\)?

\(\text{a bigamist}\)

(c) What do you call \(M - \mathcal{D}(\text{Married})\)?

\(\text{bachelors}\)

(d) Suppose \(x \in M\). What is \(\text{Married}(x)\)?

\(\text{if } x \text{ is married, a wife of } x\)

18. Continuing the previous exercise, let \(\text{Mother} \subseteq P \times W\) and \(\text{Father} \subseteq P \times M\).
(a) What is $Mother^{-1} \cup Father^{-1}$?
   the relation Child (of a living mother or father)
(b) What is $\overline{D}(Mother \cup Father)$?
   orphans
(c) What is $Mother \cap Father$?
   \{
   \}
(d) What is $Mother^2$?
   the relation Grandmother


   Adam $\overset{\text{def}}{=} M - D(Father)$
   Eve $\overset{\text{def}}{=} W - D(Mother)$

Chapter 2—Syntax

1. Define terms . . . See the text.

2. In terms of the CFG in Table 2.99:

   (a) Does it make sense to write
   \[
   \Pi = \{r_0, r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9\}
   \]
   Yes. \(\Pi\) is a set of rules; the names can be used for them. It is not unusual for a compiler written in C to have an \texttt{enum} type containing the names of the CFG rules for use in intermodular communication between parser and generator.

   (b) Does it make sense to write \((\text{Complement}, \neg \text{Boolean}) \in \Pi\)?
   Yes. A rule is a pair from $V_N \times V_T^*$. 

   (c) What are $V_T$, $V_N$, $G$ and $\Pi$?

   \[
   \begin{align*}
   V_T &= \{ \vee, \wedge, \neg, t, f, (, ) \} \\
   V_N &= \{ \text{Proposition}, \text{Disjunction}, \text{Conjunction}, \text{Complement}, \text{Boolean} \} \\
   G &= \text{Proposition} \\
   \Pi &= \{ (\text{Proposition}, \text{Disjunction}), \\
   &\quad (\text{Disjunction}, \text{Disjunction} \lor \text{Conjunction}), \\
   &\quad (\text{Disjunction}, \text{Conjunction}), \\
   &\quad \text{etc.} \}
   \end{align*}
   \]
(d) The text $f \lor t$ is a wff. One can show this fact by applying rules $r_8, r_6, r_4, r_2, r_7, r_6, r_4, r_1$ and $r_0$ in that order. Do it. Give a “rule of thumb” for finding such a sequence of rules for this grammar. Apply it to text $t \land f$. Try numerous other examples.

to be done

(e) What is the effect of replacing rule $r_5$ with

Complement

$\neg$ Complement

to be done

(f) The text “tt” is not a wff. What happens when you try to rewrite it?

to be done

3. Write down an everyday grammar for

$$G = \{\{b, c\}, \{G, A, B\}, G, \{G, A\}, \{A, Bb\}, \{A, Bc\}, \{B, \lambda\}\}$$

G
A
B b
B c
B

Note that the empty right-hand side is not especially noticeable – a definite weakness in this notation.

4. Using the grammar in Table 2.99 as a guide, write a grammar describing instead arithmetic expressions (digits, $+$, $-$, $\times$, $/$).

Expression
  Sum
    Sum + Factor
    Sum - Factor
    Factor
  Factor
    Factor * Primary
    Factor / Primary
    Primary
  Primary
    Digit
    ( Expression )
5. Define terms . . . . See the text.

6. Is $G \in \mathcal{L}(\mathcal{G})$ for any $\mathcal{G}$?
   No, $G$ is nonterminal and $\mathcal{L}(\mathcal{G})$ contains only strings of terminals.

7. Write a grammar $\mathcal{G}$ for which $\lambda \in \mathcal{L}(\mathcal{G})$.
   \[ G \]

8. For several values, $n = 0, 1, 2, \ldots$, write a grammar for which $\text{size}(\mathcal{L}(\mathcal{G})) = n$.
   \[ G1 \]
   \[ G2 \]

9. Solve the previous two problems simultaneously. For example
   \[ G3 \]
   \[ \mathcal{L}(\mathcal{G}) = \{\lambda, 2, 3\} \]

10. Write a grammar for which $\text{size}(\mathcal{L}(\mathcal{G})) = \infty$.
    \[ I \]
    \[ I1 \]
    gives all sequences $\lambda, 1, 11, 111, \ldots$
11. Say how you would write a grammar for which the only valid string is Lincoln’s Gettysburg address.
One rule containing the entire address.
\[ G \]
Four score and seventy years ago our forefathers...

12. Using Table 2.99, give two parses for \( t \lor t \), one canonical and one not. Make up more strings to parse and find the canonical parse for each.
\[ r_7, r_6, r_4, r_2, r_7, r_6, r_4, r_1, r_0 \] is canonical.
\[ r_7, r_7, r_6, r_6, r_4, r_4, r_2, r_1, r_0 \] is not canonical.

13. Using Table 2.99:

(a) On each line of the proof, identify the rule and all of the quantified variables in Equation 2.99. E.g., on the \( r_0 \) line, \( \alpha = \gamma = \lambda \), \( B = \) Proposition, \( \beta = \) Disjunction, \( r = r_0 \). Verify that the parse was canonical.

\[
\begin{array}{cccc}
\text{rule} & \alpha & \beta & \gamma & B \\
\text{r0} & \lambda & \text{Disjunction} & \lambda & \text{Proposition}
\end{array}
\]

(b) Verify that the derivation in Table 2.99 is canonical.
From the exercise above \( \gamma \in V_T^* \) at every step.

(c) Prove that each of \( t \), \( f \), \( (t) \) and \( t \lor (f) \) are Propositions.
\[ t: r_7, r_6, r_4, r_2, r_0 \]
\[ f: r_8, r_6, r_4, r_2, r_0 \]
\[ t \lor (f): \text{tbd} \]

(d) Show that \( f(\lor)t \) cannot be a Proposition.
Assume \( f(\lor)t \) is a Proposition. Only \( r_9 \) can consume the parentheses, therefore \( \lor \) must be a Disjunction. And only \( r_1 \) can consume \( \lor \), therefore Disjunction \( \rightarrow^* \lambda \). But there is no erasure in this grammar; contradiction.

(e) Prove \( SF(\text{Disjunction} \lor t) \)
to be done

14. Write down \( L(G) \) where \( \Pi = \{ \langle G, A \rangle, \langle A, Bb \rangle, \langle A, Bc \rangle, \langle B, \lambda \rangle \} \)
\[ L(G) = \{ b, c \} \]

15. Find two different grammars \( G \) each describing \( L(G) = \{ ab, ba \} \)
\[ G_1 \]
\[ ab \]
\[ ba \]
\[ G_2 \]
16. Find a grammar $G$ describing all sequences of a’s and b’s with an odd number of a’s.

$G$

$B a B$

$G a B a B$

$B$

$B b$

17. Define terms . . . . See the text.

18. Redraw the parse tree replacing the phrase names with the name of the rule that was applied. Since the phrase name is the left side of the rule, no information has been lost. This labeling is less mnemonic, but more useful, than the one using phrase names.

19. What associativity rule is used for exponentiation in PL/1?

Right associative. $2^{*}3^{*}4 = 2^{*81}$. $2^{*}3^{*}4 \Rightarrow (2^{*3})^{*}4 = 2^{*12}$.

20. Using Table 2.99, draw parse trees for $(t)$ and $f \lor t \land (t \lor f)$.
21. Devise a grammar for $+, -, \times, /,$ and $\uparrow$ (meaning the four arithmetic functions plus exponentiation) so that the four arithmetic functions are left associative, exponentiation is right associative and there are three levels of precedence ($\uparrow$ over $\times$ and $/$, and these over $+$ and $-$). Restrict the operands to single digits. Draw parse trees for several expressions. How does the result compare with your answer to Exercise 99.

22. Repeat the previous exercise adding parentheses to the language.

23. Repeat the previous exercise adding unary $'-'$ to the language. Decide what precedence unary $'-'$ should have. Can you justify your choice?

24. Verify that a postorder walk of the tree in Figure 2.99 gives the canonical parse for $f \lor t \land \lnot = f$.

25. Prove that a postorder walk of a parse tree always gives the canonical parse.

26. Describe an algorithm to build the parse tree from the canonical parse and the grammar.

27. Show that for any CFG each subtree of a parse tree corresponds to a contiguous subsequence of the canonical parse.

28. Extend the previous exercise to show that any rule $r$ appearing in a canonical parse is immediately preceded by $m$ contiguous subsequences, one for each nonterminal on the right-hand of the rule.

29. Using the results of the previous exercise, design a space-efficient computer representation of parse trees.

   *The tree needs access functions. Suppose one chooses the pair of functions below as adequate for tree walking.*

   ```
   int Rule(Node N)
   int Branches(int R)
   Node SubTree(Node N, int k)
   ```

   *where $\text{Rule}(N)$ is the short name of the rule applied to build node $N$, $\text{Branches}(\text{Rule}(N)) = 0$ when $N$ is a leaf, otherwise $\text{SubTree}(N, k)$ is defined for $0 \leq k < \text{Branches}(\text{Rule}(N))$.*

30. Compute $\text{PFN}$ for $((t))$ and $t \land (t \lor f)$ from the canonical parse.

   $t$

   $ttf \lor \land$
31. Compute \( \text{PFN} \) for \( \text{tft} \lor \neg\text{f} \land \text{f} \) using Table 2.99. Draw the parse tree. What is an equivalent expression? Is it unique?

32. Use a stack to evaluate \( \text{tft} \lor \neg\text{f} \land \text{f} \).

33. Show, by induction on the \( \text{CFG} \) for \( \text{PFN} \), that the number of infix operators is always one less than the number of \( \text{t/f} \) operands.

34. Use the result of the previous exercise to prove that exactly one value remains on the stack after evaluation, thereby insuring the integrity of the stack and the uniqueness of the value of a \( \text{PFN} \) expression.

35. Add a nonterminal ‘Conditional’ for if-then-else to propositions. It should describe the proposition
   \[
   \text{if if } t \text{ then } f \text{ else } t \text{ then } f \text{ else } t
   \]
   among others. Add an \( \text{Op3} \) operator ‘ifthenelse’ to the \( \text{PFN} \) \( \text{CFG} \). Make up some examples to turn into \( \text{PFN} \)and execute.

36. Remove operands \( t \) and \( f \) from the \( \text{CFG} \) for \( \text{PFN} \) and add a phrase name \( \text{Op0} \) to the \( \text{CFG} \) with two zero-ary operators \( t \) and \( f \). As the phrase name indicates, these new \( \text{PFN} \) operators do not pop anything from the stack. Like all \( \text{PFN} \) operators they push a single result on the stack. Now repeat Exercise 2.32 and the one following it. Does it change anything to have only operators in \( \text{PFN} \)? Extend the stack-integrity proof to \( \text{PFN} \) containing operators \( \text{OpN} \) for \( N = 0 \) up to some arbitrary value.

37. Substitute 1 for \( t \) and 0 for \( f \) in both the \( \text{PFN} \) and Proposition \( \text{CFGs} \). This changes nothing. Now generalize 0 and 1 to binary integers and the operators to act bitwise. That is the relation between the two languages and machines?
   The latter handles boolean vectors, the former boolean scalars.

38. Continuing the previous exercise, add a postfix operator ‘.’ to propositions and a corresponding \( \text{Op1} \) operator to the \( \text{PFN} \). Add a conventional memory to the stack machine and interpret the ‘.’ operator as ‘fetch from memory’. The effect is to add variables to propositions.

39. Define terms . . . . See the text.

40. Note that the grammar for propositions is left recursive. Devise an algorithm to detect left recursion in a \( \text{CFG} \).

41. Write a circular \( \text{CFG} \).

42. Devise an algorithm to detect circularity in a \( \text{CFG} \). What is the best diagnostic you are able to provide?

43. Devise an algorithm to remove erasure from any \( \text{CFG} \) \( G \) such that \( \lambda \not\in L(G) \). Show that applying your algorithm leaves size(\( \Pi \rightarrow \{ \lambda \} \)) = 1 if \( \lambda \in L(G) \).
44. Write a CFG with an unreachable nonterminal.

45. Devise an algorithm to detect unreachable nonterminals. What is the best
diagnostic you are able to provide?

46. Write a CFG with a nonterminating nonterminal.

47. Devise an algorithm to detect nonterminating nonterminals. What is the
best diagnostic you are able to provide?

48. Show that the following CFG is ambiguous.

\[
G \\
S \\
i S t S e S \\
i S t S \\
x
\]

49. Write an ambiguous CFG shorter that the one in the previous exercise.

\[
G \\
GG \\
x
\]

50. Write an ambiguous CFG with size(\(\Pi\)) = size(\(\mathcal{R}(\varnothing)\)).

See previous answer.

51. Show that you cannot write an ambiguous CFG with size(\(\Pi\)) = size(\(V_N\)).

Each nonterminal is on the left of a rule, thus no nonterminal has more than
one defining rule. Starting from G, each replacement is predetermined,
therefore size(\(\mathcal{L}(G)\)) = \(\infty\) and there is just one canonical parse for it.

52. Show that any CFG with a nonterminal A for which both \(A \rightarrow^+ A\alpha\) and
\(A \rightarrow^+ \beta A\alpha\) (left and right recursive in A) is ambiguous.

The partial parse \(A \rightarrow^+ \beta A\alpha\) contains the ambiguity because it can be
either \(A \rightarrow^+ \beta A \rightarrow^+ \beta A\alpha\) or \(A \rightarrow^+ A\alpha \rightarrow^+ \beta A\alpha\). Having arrived at A in
any parse, there are two canonical ways to get to \(\beta A\alpha\).

53. Devise an algorithm that, given a CFG, either correctly reports whether it
is ambiguous or unambiguous, or reports it failed to make the distinction.

If the grammar is left and right recursive in some symbol, report ambigu-
ous. There are classes of grammars for which automatic parsers can be
built (see Chapter 5). Report unambiguous for each such grammar. For
all others report failure to determine. One could do better.
54. Extend the previous exercise with the additional constraint that the algorithm never reports a failure.
   Instead of reporting failure, loop.

55. Extend the previous exercise with the additional constraint that the algorithm always terminates. This is the same as devising an algorithm to detect all ambiguous CFGs.
   This is undecidable — cannot do it.

56. Show that for any probabilistic grammar,
   \[
   \sum_{r \in \Pi} p(r) = \text{size}(V_N)
   \]
   \[
   \sum_{r \in \Pi} p(r) = \sum_{B \in D(\diamond)} \sum_{r \in \{B\} \cap \Pi} p(r) = \sum_{B \in D(\diamond)} 1.0 = \text{size}(V_N)
   \]

57. Add, change or remove some rules and reassign probabilities in the grammar in Table 2.99. What are your \(V_N\) and \(V_T\)? Generate some “English” sentences using a random number generator (10-sided dice are convenient).
   To what extent do the sentences satisfy your instincts for English style and content? What is the expected length in sentences of a “Story”?

58. Devise a probabilistic grammar that could generate “Jack and Jill.” What is the probability that it would?
   
   \textit{Jack and Jill went up the hill to fetch a pail of water.}
   \textit{Jack fell down and broke his crown and Jill came tumbling after.}
   25 words and a period, 22 unique symbols.

   Table 2.99 is an interesting starting point. A new nonterminal Runon-Sentence connecting sentences by conjunctions, prepositional phrases and objects need to be added. With optimal probabilities this gives a more credible solution than the following suggestion:
   Replace Sentence with Word and give a rule defining Word to be each of the words in the story. That is:

   \begin{tabular}{|l|c|}
   \hline
   Story & 1.0 \\
   Words & 0.97 \\
   Words Word & 0.03 \\
   Word & .05 \\
   Jack & 0.05 \\
   and & 0.05 \\
   ... & ... \\
   after & 0.04 \\
   \hline
   \end{tabular}
The expected length (50%) is about 23; the probability that a story of the right length (25) is in fact the limerick is about \((1/20)^2\).

59. Pick a child’s first reader and devise a probabilistic CFG that can generate all of the sentences in the reader.

   After doing Jack and Jill, it seems that there is a near zero probability of generating anything interesting for long texts. Bad assignment.

60. Devise an interesting probabilistic CFG that can generate Lincoln’s Gettysburg address. The CFG that generates only the LGA is not interesting; neither is the CFG that generates all addresses. What is the probability of generating the LGA with your CFG?

   The LGA is even harder than the reader because the syntax is more sophisticated and requires a more elaborate grammar. Gedanken problem only.

61. Devise a probabilistic CFG to generate interesting programs in C (or some other programming language). The measure of success is the proportion that compile and run. Another measure of success is the proportion of all legal programs that could be generated. A perfect generator is capable of generating every runnable program and never generates one that will fail to compile or will abort during execution. Could one invent a programming language for which a perfect generator is possible?

   There already is a CFG for C so one might as well use it. That guarantees no syntax errors. It also forces the choice of identifiers and constants to conform to the semantics of the whole language (harder than picking a subset). A few of each is enough. There are a number of generatable C programs of length 3. For example

   ```
   int i;
   ```

   Nearly all of these compile. Many generatable programs of length 4 violate constraints of the language. For example

   ```
   short float sf;
   ```

   It seems that there might be value in such a generator for testing.

   Suppose a language consisted only of a sequence of expressions of constants. Then, except for arithmetic anomalies, every program is runnable. A desk calculator is an example of a computer executing such a language.

62. Pick a specific probabilistic CFG and write a program to randomly generate the strings it describes. Hint: Write one procedure per nonterminal and call the print statement for each terminal.
Sentence();
    if (random() < 0.1) break;
}

void Sentence(void) {
    Subject();
    Predicate();
    printf(".ln");
}

63. Write a program that accepts any probabilistic CFG as input and generates syntactically correct strings according to the assigned probabilities.

This assignment requires considerable machinery. One must be able to read a grammar and build a data structure from which it can be applied. Much of Chapter 5 also requires such machinery. Then the generating program starts with a list of length 1 containing the goal symbol, and repeatedly locates a nonterminal in the list, rolls the dice, rewrites it and continues until there are no more nonterminals.

64. Write a program to accept a sequence of strings and compute an interesting probabilistic grammar that generates them with approximately the observed probabilities. The length of the computed grammar should not grow linearly with the length of the sequence.

This problem is algorithmically straightforward and computationally awful. The two bounding grammars (one generates all strings in the manner of the second Jack and Jill solution; the other generates just the input strings) are not interesting and forbidden as solutions. The question of how the mind formulates grammars in the face of such problems is worth contemplating.

65. Devise a test program generator with three parameters:

(a) LENGTH: approximate length of generated program in tokens (0 to 10000).
(b) CONSTRUCT: principal construct to be tested (Expression, IfThenElse, Declaration...).
(c) LANGUAGE: programming language (X, C, Pascal, Ada...).

Test it and save it for the exercises on compiler performance in Chapter 4.

66. One could build a parser for any CFG by assigning probabilities to its rules and keeping track of the generated strings. To parse some string \( \alpha \), the generator would be started. When \( \alpha \) finally showed up, the record of how
it was generated would be the parse. How long would you expect such a parser to run as a function of the length of its input?

The first problem is to generate a string of the right length. Whatever probabilities are assigned, a string of length $N$ appearing is less likely than $1/N$. The parse is longer than the input by some constant factor $C$. Each decision is taken with some probability $1/n$ for small $n$ (say 2 or 3). All this leads to an approximate chance of $(1/N)(1/n)^C$ of generating the exact text needing parsing. One could wait a long time.

67. What is $V_T$ in the Grammar-grammar in Table 99?

$$V_T = \{=, ;, ', A, B, \ldots z\}$$

68. Use the cFG in Table 2.99 to parse itself.

For each nonterminal $X$, name its rules $X_1$, $X_2$, $X_3$. The first two rules are named $Grammar1$ and $Grammar2$. The next rule is named $Rule1$, and so on.

The canonical parse starts with rule $Grammar2$ making nonterminal $Grammar$ out of nothing (empty grammar). Then the parse contains $Letter7$ (for $G$), $Nonterminal1$, then 6 $LetterN$-$Nonterminal2$ pairs reducing the identifier ‘Grammar’ to Nonterminal. Rule $Phrase2$ starts nonterminal $Phrase$ out of nothing (empty phrase). The the sequence beginning with $Letter7$ is repeated reduce the second ‘Grammar’ to Nonterminal. Rules $Symbol2$ and $Phrase1$ follow. After a few more $Letter$-$Nonterminal$ pairs, $Symbol2$, $Phrase1$, $Rule1$ and $Grammar1$ the first line is parsed. And so on.

69. Write a self-describing grammar-grammar that neither uses nor allows erasure (second and fifth rules in Table 2.99).

```plaintext
Grammar = Grammar Rule;
Grammar = Rule;
Rule = Nonterminal '=' Phrase ';;';
Phrase = Phrase Symbol;
Phrase = Symbol;
Symbol = Terminal;
Symbol = Nonterminal;
Terminal = ''' ' = '' ' ;
Terminal = ''' '' ;
Terminal = ''' ;
Terminal = ''' ;
Terminal = ''' ;
Terminal = ''' ;
Terminal = ''' ;
Nonterminal = Letter;
Nonterminal = Nonterminal Letter;
Letter = 'A';
Letter = 'B';
...;
Letter = 'z';
```
70. Use BNF to unambiguously describe itself.

The intent of BNF was that whitespace could be ignored (and therefore not used to disambiguate). The assignment is impossible. One can get a good start however:

\[
\begin{align*}
\text{<Bnf>} & ::= \text{<Rule>} \\
& \mid \text{<Bnf> <Rule>}
\\text{<Rule>} & ::= \text{<Nonterminal>} ::= \text{<Phrase>} \\
& \mid \text{<Rule>} \mid \text{<Phrase>}
\\text{<Phrase>} & ::= \\
& \mid \text{<Phrase> <Symbol>}
\\text{<Symbol>} & ::= \text{<Terminal>} \\
& \mid \text{<Nonterminal>}
\\text{<Terminal>} & ::= :: = | < | > | | | <Letter>
\\text{<Nonterminal>} & ::= < <Identifier> >
\\text{<Identifier>} & ::= <Letter>
\& \mid <Identifier> <Letter>
\\text{<Letter>} & ::= A | B | C | \cdots z
\end{align*}
\]

The inability to distinguish between metasymbols and terminals is the first cause of ambiguity. For example, one cannot tell whether there is one, two or three rules defining \text{<Rule>}. There are several other ambiguities of this kind. The definition of \text{<Nonterminal>} itself is ambiguous; it could require \text{< and >} and 10 specific letters inside \text{< and >}.

71. Noting that the grammar-grammar used in the example is not left recursive, change the two right recursions into left recursions and repeat the derivation. You will have to become clairvoyant on some decisions.

72. Duplicate the proof in Table 2.99 by deriving

\[TD(Proposition, f \lor t \land \neg f) \text{ from } TD(\lambda, \lambda)\]

73. Show \(SF(\alpha) \Leftrightarrow TD(G, \alpha)\).

74. Show that \text{Scan.g} is ambiguous (ignore the output vocabulary for this exercise).

75. Parse \(x\) and \(x + x\) using \text{Sum.g}, keeping track of the output.

76. Translate the grammar for Proposition (Table 2.99) into the executable notation. Add emitters for the parse sequence. Parse \(f \lor t \land \neg f\) by the new grammar, duplicating the information in Table 2.99. (Note, the clumsier executable notation will cause you to add some rules for which you will not produce output.)

77. Find the shortest \text{Short.g} such that

\[Gem(Short.g, Short.g) = Short.g\]
78. Implement GEM. Invert the parser \texttt{Sum.g}. Call the result \texttt{UnSum.g}. Parse an expression and use \texttt{UnSum.g} to recover the original input. (Note: grammars must start with a gratuitous ‘;’ for GEM to work. It is easiest to just leave room in the code array and stick a ‘;’ in upon program initialization.)

79. How does GEM respond to a missing right quote (‘ or ”) in the grammar? Can you suggest an improvement?

80. How does GEM respond to errors in the input (see page 99)? What can be done to make GEM easier to use?

81. Write a grammar to take deblanked grammars and restores them to readable form. Call it \texttt{Pretty.g}; it is a kind of inverse to \texttt{Scan.g}. Write a formula describing the relation between \texttt{Scan.g} and \texttt{Pretty.g}.

82. Extend \texttt{Scan.g} so that it also removes comments in the style of those in \texttt{x}.

83. Change the expression-to-PFN translator \texttt{Expr.g} to process Propositions instead.

84. Try the grammar for PFN in Table 2.99 on GEM and explain the behavior. Then invent and implement a grammar that does accept PFN as input.

85. Extend the expression-to-PFN translator \texttt{Expr.g}. Add right associative exponentiation. Without using \texttt{Scan.g}, allow blanks in the expression input. Allow multicharacter identifiers and integers. Separate all PFN output by single blanks.

86. Speed up GEM by making L, D and C reserved names, implementing them directly in the “hardware” of GEM.

87. Speed up GEM \texttt{SEARCH} mode by building a table of starting points for the search, and links to next-rule rather than rummaging past rule after rule.

88. What are $\mathcal{L}(\sim())$, $\mathcal{L}(a|b)$, $\mathcal{L}(a|a)$, $\mathcal{L}(a\&b)$, $\mathcal{L}(a\&a)$, $\mathcal{L}(a^+)$, $\mathcal{L}(a^9)$, $\mathcal{L}(a^{b3})$, and $\mathcal{L}(cr^t(a\&b))$?

89. Where is $((H|n)a)^2(ln)^2 - ((\sim)H|n)^3()$?

90. Describe the following programming language constructs as regular expressions:

(a) identifier
(b) real constant
(c) C reserved word

\footnote{This is the GEM that is in the distribution package.}
91. Give a regular expression for change for a dime.

92. Give a regular expression for regular expressions.

93. Elaborate on the following “regular schedule” by adding rules to describe more symbols. How far does it pay to carry such a description? Would \( V_G = \{ \text{His, Hers} \} \) be a good idea?

\[
\text{DailyGrind} = (\text{Weekday}^5 \text{ Saturday Sunday})^*; \\
\text{Weekday} = \text{Alarm AMchores Drive Work Drive PMchores Sleep}; \\
\text{Saturday} = \text{Alarm Sleep AMchores (Shopping | Lawns) PMchores Sleep}; \\
\text{Sunday} = \text{AMchores Church (TV | Outing) PMchores Sleep};
\]

94. The essence of grammar is phrase structure; things defined in terms of sequences of things, to any depth. Describe something else (other than DailyGrind) with a REG. Extra credit for creativity.

95. The grammar-grammar in Table 2.99 can also be rewritten as a REG, and then rewritten once again to be self describing. Do both, including only meta operators ‘*’, ‘?’, parentheses, and ‘|’. You may assume an ASCII form of input.

96. Repeat the previous exercise but use all of the regular expression meta operators. You must do some language design to decide how to represent the subscripts and superscripts within the limits of ASCII.

97. Consider a program with the following properties:

(a) The input consists of letters and blanks. A word is any sequence or 12 or fewer letters. A telegram is any sequence of words (but not XXXX), separated by one or more blanks, terminated by word XXXX. A batch is a sequence of telegrams, also terminated by word XXXX. Use a REG to describe a batch. (Hint: Use operator ‘−’ and integer superscripts.)

(b) The output consists of a sequence of 40-character lines. Each line starts with a sequence of words separated by single blanks, and is padded on the right with zero or more blanks. The output words are the same as the input words, and in the same order, except that word XXXX does not appear and telegrams are separated by a blank line. Use a REG to describe output resulting from an acceptable batch (hint: use operator ‘&c’). Is your grammar strict enough to make the output well defined?

(c) Combine your grammars by letting \( V_G = \{ \text{input, output} \} \). This is an example of a nonstandard grammar.

(d) How would you send this exercise as a telegram?
The last set of exercises in Section 2.99 should be reread and perhaps reworked using the regular expression operator \*.

Express the grammar in Table 2.99 as an executable REG and use it to test the extended version of GEM.

Given the rules
Conjunction = Complement (′∧′ Complement)∗
Complement = ′¬′ Boolean
write procedures Conjunction and Complement to generate rules r3, r4, r5, r6 so that
- All matches are positive (use ‘==’).
- After every match on ch, call getchar. There should be exactly one “discard” for every terminal in the REG.
- There is one call to generator Rule for each rule of the CFG.

Put the procedures all together, compile them, and parse Propositions. The implementation of Rule can just contain a print statement.

The calls to the error routine could pass a string parameter giving a diagnostic. Write down the best diagnostic you can think of for each of the three calls to Error. What other information would you like the error routine to have to enhance the quality of its diagnostics?

It would seem that since there are so many things that are not syntactically a Proposition that there would be more calls to Error. Why aren’t there?

Suppose that instead of aborting the run, Error returns to continue translation after issuing a diagnostic. What must you do to insure that the parser does not get into a loop?

Extend the lexical structure of X to allow simple character constants.

Extend the lexical structure of X to allow underbar in identifiers.

Extend the lexical structure of X to allow underbar inside identifiers (but not starting or ending them).

Write an unambiguous REG for the lexical structure of X (that is, one that does not depend on longest-match). (Hint: have several kinds of lexeme lists, one ending in Identifier, one ending in Integer, one ending in WhiteSpace, and so on.)

The real number 12.31 can be processed by turning ′.’ into a digit, as the cost of 1.2.3.1 also being accepted by the lexer. This usually does not cause any trouble because the mistake will be caught during numeric conversions. One can also look at 12.31 as three lexemes, ‘12’, ′.’ and
APPENDIX E. ANSWERS

‘31’. The disadvantage is that now 12 . 31, with imbedded blanks, is also acceptable. Would the lexical REG have to be modified to allow the second alternative?

110. In C the assignment *x++<<= -++++y; is allowed. What are the tradeoffs between this kind of lexical structure and the one proposed for X? Why is the blank after the = necessary? Or is it? Or do you care to know?

111. Explain the function Lex.x in Table E.0 line by line.

112. Extend the solution in Table 2.99 to handle the backslash escape in C strings. You must be able to accept "\n" as well as "\007". It is convenient to allow "\x" to stand for \x wherever x does not have some other interpretation.

113. In C division by a dereferenced value caused a translation diagnostic.

    *ThisIsAPointer = ThisIsANumerator/*ThisIsAPointer;

What is the problem? What is the cure? (Turning the clock back is not an acceptable answer.)

114. In C all of the following: "\n", 1.2e-3, /* This is a Comment */, a***b, as well as a convention for hiding linebreaks, defeat the relatively simple lexer algorithm just presented. Other conventional languages have similarly complex lexical constructs. It is not hard to process such things but it does complicate the lexer. There is a single loop in the one presented here; there may be a loop per kind of lexeme in a more complex lexer. The multiple loop lexer will look like the program in Table E.0. Even though it is not consistent with other simplifications in the presentation you are enjoying here, implement a lexer for C.

115. A particularly straightforward way to implement Scan() is to switch on every character, without regard for classes such as Letter and Digit, until the token is isolated. Such a scanner is big but easy to understand and also fast. The isolation of lexemes and the identification of tokens is all mixed together. A fragment of such a scanner written in a C-like notation follows:

```c
    Scan() {
        switch (ch) {
            case 'i':
                ch = getchar();
                switch (ch) {
                    case 'f':
                        ch = getchar();
```

³This can be done in about 500 lines of C code.
lexeme = "if";
if (ch∈Letter∪Digit) {
    return MakeToken(lexeme, ifTOKEN);
} else {
    while (ch∈Letter∪Digit) {
        lexeme = strcatchar(lexeme, ch);
        ch = getchar();
    }
    return MakeToken(lexeme, Identifier);
} etc. etc.
}

Implement a scanner for X this way. Compare the result with that of the previous exercise. How deeply would switches nest for C? For Ada?

116. Revisit the exercises (Exercises 99 ff) to extend the lexical reg. Extend the lexicon instead.

117. Try writing Statement right now, before reading about the standard solutions. If you cannot do it, then read the first solution below and return to do this exercise.

118. Suppose that Variable were arbitrarily complex, containing subscripts, structure qualification, indirection or anything else that makes trying to look ahead futile. Now repeat the previous exercise. If you cannot do it, then read the second solution below and return to this exercise.

119. Rework the parser for Assignment, this time allowing Assignment instead of Disjunction inside the parentheses of Boolean.

120. Devise a cfg for arithmetic assignments analogous to the cfg for boolean assignments presented in this section. Implement a recursive descent parser for it.

121. Implement a recursive descent parser for English, using the cfg in Table 99 or your own extension of it.

122. Conventional programming languages present a fair challenge to the writer of a recursive descent parser. The following grammar is abstracted from the ANSII C REG. Write function AbstractDeclarator. You see that 9 functions are required according to the REG. Since a Pointer always starts with a ‘*’ and neither AbstractDeclarator nor AbstractQualifier do, the problem of the optional Pointers is resolvable. The next problem arises in ParameterDeclaration: when does one call Declarator and when does one call AbstractDeclarator? It turns out they are distinguished, in general,
only when the Identifier of the Declarator shows (or fails to show) up. That is:

\[
\text{Declarator} \rightarrow^* \ast \ast \ast \{\} \\
\text{AbstractDeclarator} \rightarrow^* \ast \ast \{\}
\]

Try your hand at a solution and then read the following material.

123. Implement a parser for arithmetic assignments in C.
124. Implement a parser for AbstractDeclarator.
125. Confirm that the CFG and REG for X in Tables 99 and 99 and 99 describe the same language.
126. Write a parser for X.
127. needed

Chapter 3—Semantics

1. By the statement of the problem, one can write \( V\text{ar} = \{x\} \cup R \) and \( V\text{ar} \times V\text{al} = \{x\} \times V\text{al} \cup R \times V\text{al} \) where \( R \) is the rest of the variables. Suppose \( s = \text{choice}(\Sigma_x) \). That is, \( s \in \Sigma \land s \not\in \Sigma_x \). From the definition, \( s \subseteq \{x\} \times V\text{al} \cup R \times V\text{al} \land s \not\subseteq R \times V\text{al} \).

2. 2

3. The question reduces to \( \text{choice}(Y) \in Y \) where \( Y = R(\{x\} \triangleleft s) \).

4. Substituting for the definition, \( s(\{x\}) \leftarrow \{v\} = (\{x\} \triangleleft s) \cup (\{x\} \times \{v\}) = (\{x\} \triangleleft s) \cup \{x, v\} = s(x) \leftarrow v \)

5. Only if \( s \in \Sigma_x \)
6. yes
7.

\[
\Sigma_X \triangleleft \overset{\text{def}}{=} 2^{V\text{ar} \times V\text{al}} \\
\Sigma_X \overset{\text{def}}{=} \Sigma - \Sigma_X
\]

8. Substituting for the definition, \( s(X) \leftarrow s(X) = (X \triangleleft s) \cup (X \times R(X \triangleleft s)) \). The result is \( s \) only if all the variables \( X \) already each have exactly the same values. One such circumstance is where \( X = \{x\} \) and \( s \) is deterministic in \( x \).

9. This formula specifies all the variables \( X \) are to become undefined. Substituting for the definition, \( s(X) \leftarrow \{\} = (X \triangleleft s) \cup (X \times \{\}) = (V\text{ar} - X) \triangleleft s. \) But \( (V\text{ar} - X) \triangleleft s \in \Sigma_X \). This state is \( s \) only when \( D(s) \cap X = \{\}, \) that is, when all of \( X \) is already undefined.
10. Substituting for the definition, \( s(\{\}) \leftarrow s(X) = (\overline{\text{Var}} s) \cup (\{\} \times s(X)) = \text{Var} s = s. \)

11. Substituting for the definition, \( s(\{x\}) \leftarrow \{\} = (\overline{\{x\}} s) \cup (\{x\} \times \{\}). \)

Chapter 4 — Performance

1. tba

Chapter 5 — Advanced Topics

1. [1,1] Draw a diagram for a DFA that recognizes positive integers.\(^4\)

2. [1,1] Draw a diagram for a DFA that recognizes rounded values of \(1/7\).

3. [1,1] Suppose that you have a DFA that recognizes truncated representations of fraction \(1/n\). How can you transform it into a DFA that recognizes rounded representations of \(1/n\)? (Hint: does your solution work for \(1/101\)?).

4. [1,1] Draw a diagram for a DFA that recognizes any sequence of nickels and dimes (N and D) that adds up to a quarter. (Hint: let state \(k\) represent an accumulation of \(5k\) cents.)

5. [1,1] Draw a diagram for an automaton that recognizes a sequence of zero or more a’s, followed by a sequence of zero or more b’s, followed by one c. (Hint: there is a 3-state NFA solution.).

6. [1,1] Use a regular expression to describe the strings accepted by each of the automata in this list of exercises.

7. [1,1] Write down the grammars defining the automata derived in the previous set of exercises.

8. [1,1] Write a program to execute the DFA in Figure E.1.

9. [1,1] Write a program to execute any DFA. (Hint: represent \(\Pi\) as a 2-dimensional matrix. The matrix defines a function mapping each state-symbol pair into a state and \(V_N\) as a vector recording which states are final and which are not.)

10. [1,1] Write a program to execute any NFA. (Hint: provide backtracking.)

11. [1,1] Show how to derive a grammar \(A'\) for the sequence of states passed to accept a string from the grammar \(A\) defining the FA. (Hint \(V_F' = V_N\)).

12. [1,1] Show how to derive a grammar \(A'\) for the sequence of transitions applied to accept a string from the grammar \(A\) defining the FA. (Hint: \(V_F' = \Pi\)).

\(^4\)The statement of a recognition condition for an automaton implies in addition “and rejects anything else.”
APPENDIX E. ANSWERS

13. [1,1] Show $A \in C(\{A\})$.

14. [1,1] Show $A' \subseteq C(A')$.

15. [1,1] Show that the error state is not a final state.

16. [1,1] Show $\{\} \in V'_N$ iff $L(A') \subset V'_T$.

17. [1,1] Show that for any constructed DFA $A'$

$$\text{size}(\Pi') = \text{size}(V'_N) \times \text{size}(V'_T) + \text{size}(V'_F)$$

18. [1,1] Suppose the NFA–DFA transformation were applied to a DFA. Would the input and output of the transformation necessarily be the same?

19. [1,1] Suppose the NFA–DFA transformation were applied twice, to a NFA and then to the resulting DFA. Would the input and output of the second transformation necessarily be the same?

20. [1,1] Verify the NFA–DFA transformation.

21. [1,1] Verify the formula in Exercise 17.

22. [1,1] Label the states of the DFA for $a^*b^*c$ and draw the diagram for it.

23. [1,1] Repeat the previous three exercises for $a^*b^*c^*$.

24. [1,1] Show that the set of parse stacks $\rho$ is a FA language.

25. [1,1] Show that for an LR(0) automaton, $V'_F = \Pi$.

26. [1,1] Verify the construction of the example NFA.

27. [1,1] Verify the construction of the example DFA.

28. [1,1] LALR(1) — shift-reduce conflict for LR(0).

\begin{align*}
G & \rightarrow \ E \perp \\
E & \rightarrow \ T \\
T & \rightarrow \ T \ x \\
T & \rightarrow \ x
\end{align*}

29. [1,1] LALR(1) — reduce-reduce conflict for LR(0).

\begin{align*}
G & \rightarrow \ E \perp \\
E & \rightarrow \ S \ x \\
E & \rightarrow \ T \ z \\
S & \rightarrow \ a \\
T & \rightarrow \ a
\end{align*}
30. [1,1] \textbf{LALR}(1) — \textit{reduce-reduce} conflict for \textit{SLR}(1)

\[ \begin{align*}
\text{G} &\rightarrow \text{E} \perp \\
\text{E} &\rightarrow \text{a T a} \\
\text{E} &\rightarrow \text{b T b} \\
\text{E} &\rightarrow \text{a x b} \\
\text{T} &\rightarrow \text{x}
\end{align*} \]

31. [1,1] \textbf{LALR}(1) — erasure in the lookahead.

\[ \begin{align*}
\text{G} &\rightarrow \text{E} \perp \\
\text{E} &\rightarrow \text{S x} \\
\text{E} &\rightarrow \text{T U y} \\
\text{S} &\rightarrow \text{a} \\
\text{T} &\rightarrow \text{a} \\
\text{U} &\rightarrow \lambda
\end{align*} \]

32. [1,1] \textbf{LALR}(1) — defeats simple lookahead analysis of \textit{LR}(0) \textit{NFA}— a NQLR example.

\[ \begin{align*}
\text{G} &\rightarrow \text{E} \perp \\
\text{E} &\rightarrow \text{b A d} \\
\text{E} &\rightarrow \text{a A c} \\
\text{E} &\rightarrow \text{b g c} \\
\text{E} &\rightarrow \text{a g d} \\
\text{A} &\rightarrow \text{B} \\
\text{B} &\rightarrow \text{g}
\end{align*} \]

33. [1,1] not \textbf{LALR}(1)

\[ \begin{align*}
\text{G} &\rightarrow \text{E} \perp \\
\text{E} &\rightarrow \text{S x y} \\
\text{E} &\rightarrow \text{T x z} \\
\text{S} &\rightarrow \text{a} \\
\text{T} &\rightarrow \text{a}
\end{align*} \]

34. [1,1] Simple ambiguous grammar. Parse xxx⊥ two ways.

\[ \begin{align*}
\text{G} &\rightarrow \text{E} \perp \\
\text{E} &\rightarrow \text{E E} \\
\text{E} &\rightarrow \text{x}
\end{align*} \]


\[ \begin{align*}
\text{G} &\rightarrow \text{E} \perp \\
\text{E} &\rightarrow \text{i E} \\
\text{E} &\rightarrow \text{i E t E} \\
\text{E} &\rightarrow \text{x}
\end{align*} \]

36. [1,1] Verify the invariant \( G \rightarrow^* \rho \delta \) where \( G = P \) for the parse of \((f \lor f)\perp\) shown in the previous example.
37. [1,1] Apply the LR(0) machine to strings \( f \perp, (f) \perp, f \lor f \perp, (f \lor f) \perp \). What is the canonical parse in each case?

38. [1,1] Apply the LR(0) machine to string \( ff \perp \). What kind of diagnostic can be generated in this case? In general?

39. [1,1] Invent a hack to avoid the repetitious transitions across the parse stack \( \rho \) after a substitution. Hint: if \( p \) is the length of the canonical parse, and \( i \) is the length of the input excluding \( \perp \), the number DFA steps (shift or reduce) should be only \( 2p + i \).

40. [1,1] Once again apply the LR(0) machine to strings \( f \perp, (f) \perp, f \lor f \perp, (f \lor f) \perp \), but in this case use the parse state stack \( \sigma \) instead of the parse stack \( \rho \).

41. [1,1] Does using \( \sigma \) affect the quality of the diagnostics that can be generated?

42. [1,1] Given the LR(0) DFA and some parse state stack \( \sigma \), show how to compute the corresponding parse stack \( \rho \).

43. [1,1] Show that merely letting shift take precedence over reduce is the correct solution for the LR(0) machine in Figure 5.3.

44. [1,1] Each of the grammars in the set starting with Exercise 99 fails to be LR(0). Identify the failure(s). See if the LR(0) machine contains the resolution to the problem(s) as in the example worked above.

45. [1,1] Compute the relations \( <, \leq, >, \leq^*, >^*, FB \) for each of the CFGs starting with Exercise 99.

46. [1,1] Compute the functions \( shift \) for each of the CFGs starting with Exercise 99.

47. [1,1] Compute the SLR(1) functions \( reduce \) for each of the CFGs starting with Exercise 99.

48. [1,1] Considering the results of the previous two exercises, which of the CFGs are not SLR(1) and why not?

49. [1,1] What difficulties might arise from the storage efficiency hack of combining the implementing arrays for \( shift \) and \( reduce \)?

50. [1,1] Verify the computation of the LALR(1) lookahead for the example.

51. [1,1] Compute the relations \( <, \leq, >, \leq^*, >^*, FB \) for the lookahead grammars associated with each of the CFGs starting with Exercise 99.

52. [1,1] Compute the LALR(1) functions \( reduce \) for each of the CFGs starting with Exercise 99.
53. [1,1] Considering the results of the previous two exercises, which of the CFGs are not \textsc{LALR}(1) and why not?