More answers to exercises

This version contains more answers to exercises than shown in Appendix A of the book *Modern Compiler Design*.

**Answers for Chapter 1**

1.1 *Advantages:* Assuming the language is still being designed, writing a major piece of software in it is an excellent shake-down for the language design. Compiling the compiler may be a good way to debug the compiler (but there is a problem here: how defensible is debugging by using not well debugged tools?) Any improvement to the compiler benefits the compiler writers themselves, which gives them an incentive to improve the compiler more.

*Disadvantages:* Bootstrapping problems: there is no compiler to compile the first version with. Any changes to the language may necessitate many modifications to the compiler, as both the implementation and the source language change. The compiler may inadvertently be tuned to constructions used specifically in the compiler.

1.2 The front-end (particularly the intermediate-code generator) may want to know if it should optimize for execution speed or code size. In most cases, speed is more important, but if the target machine is an embedded processor in an inexpensive device, memory size may be more important.

The back-end may want to know about restrictions in the source language. For example, are pointers allowed to point to any memory location or only to variables of a certain type? Consider

```c
int x;
float *pf;
x = 12;
*pf = 0.0;
print(x);
```

If the language guarantees that `pf` cannot point to `x`, the last statement can be optimized to `print(12)`.

1.6 The code is basically that of the interpreter of Figure 1.19, except that rather than printing the value, it creates a new `Expression` node with the value.
1.7 The error reporting module reports program errors to the user; it is therefore used as follows. Program text input module: to report missing files. Lexical analysis module: to report lexical errors, for example unterminated strings. Syntax analysis module: to report syntax errors, for example unbalanced parentheses. Context handling module: to report context errors, for example undeclared variables and type mismatches. In principle, none of the other modules should need to report program errors to the user, since the annotated AST as produced by the context handling module should be that of a correct program.

1.9 In data structures outside the while statement, as with any while statement.

1.11 See Figure Answers.1.

SET the flag There is a character _a_ buffered TO False;

PROCEDURE Accept filtered character Ch from previous module:
    IF There is a character _a_ buffered = True:
        // See if this is a second ‘a’;
        IF Ch = ‘a’:
            // We have ‘aa’:
            SET There is a character _a_ buffered TO False;
            Output character ‘b’ to next module;
        ELSE Ch /= ‘a’:
            SET There is a character _a_ buffered TO False;
            Output character ‘a’ to next module;
            Output character Ch to next module;
        ELSE IF Ch = ‘a’:
            SET There is a character _a_ buffered TO True;
        ELSE There is no character ‘a’ buffered AND Ch /= ‘a’:
            Output character Ch to next module;
    PROCEDURE Flush:
        IF There is a character _a_ buffered:
            SET There is a character _a_ buffered TO False;
            Output character ‘a’ to next module;
            Flush next module;

Figure Answers.1 The filter aa → b as a post-main module.

1.13 First a subset is created by taking away some features; the language is then extended by adding new features. An example would be a C compiler which does not implement floating point numbers but does have built-in infinite-length integer arithmetic. The sarcasm comes from the fact that everything is an extended subset of everything else, which makes the term meaningless.

1.14 The grammar is now ambiguous; more in particular $8 - 3 - 5$ will now parse both as $(8 - 3) - 5$ and $8 - (3 - 5)$, with obviously different semantics.

1.15 A possible rewrite is:

\[
\text{parameter_list} \rightarrow \\
\text{in_out_option} \text{ identifier next_identifier_sequence_option}
\]

\[
\text{in_out_option} \rightarrow \text{’IN’ | ’OUT’ | } \text{ε}
\]

\[
\text{next_identifier_sequence_option} \rightarrow \\
\text{’,’ identifier next_identifier_sequence_option | ε}
\]

1.16 (a) left-recursive: B, C; right-recursive: S, C; nullable: S, A; useless: C. (b) The set \{ x, ε \}. (c) Yes, x is produced twice: S → A → B → x and S → B → x.
1.17 For example, variable_identifier, array_identifier, and procedure_identifier may be different in the grammar but may still all be represented by an identifier in an actual program.

1.18 The empty representation cannot be recognized by the lexical analyzer, so the token will not reach the parser, which is likely to make parsing (much) harder.

1.19 The grammatical production process stops when the sentential form consists of terminals only; to test this situation, we have to be able to tell terminals and non-terminals apart. Actually, this is not entirely true: we can scan the grammar, declare all symbols in left-hand sides as non-terminals and all other symbols as terminals. So context condition (1) actually provides redundancy that can be checked.

1.20 Suppose there were two different smallest sets of information items, \( S_1 \) and \( S_2 \). Then \( S_1 \) and \( S_2 \) must have the same size (or one would not be the smallest) and each must contain at least one item the other does not contain (or they would not be different). Call one such differing item in \( S_1 \) \( X \). Since both sets started with the same initial items, \( X \) cannot be an initial item but must have been added by some application of an inference rule. This rule clearly did not apply in \( S_2 \), so there must be at least one other item \( Y \) that is present in \( S_1 \) and absent from \( S_2 \). By induction, all items in \( S_1 \) must differ from all items in \( S_2 \), but this is impossible since both started with the same initial items.

Answers for Chapter 2

2.5 (c) \( 0^* (10^*1)^*0^* \)

2.7 They both mean the same as \( a^* \). They are not fundamentally erroneous but may draw a warning from a processor, since they are probably not what the programmer intended. Ambiguity is not a concept in lexical analysis, so they are not ambiguous.

2.9 See Figure Answers.2.

```c
void skip_layout_and_comment(void) {
    while (is_layout(input_char)) {next_char();}
    while (is_comment_starter(input_char)) {
        skip_comment();
        while (is_layout(input_char)) {next_char();}
    }
}

void skip_comment(void) {
    next_char();
    while (!is_comment_stopper(input_char)) {
        if (is_end_of_input(input_char)) return;
        else if (is_comment_starter(input_char)) {
            skip_comment();
        }
        else next_char();
    }
    next_char();
}
```

---

Figure Answers.2 Skipping layout and nested comment.

2.11 Let him/her implement it and then feed an object file or jpeg picture as source file to the compiler; admire the crash. Or, more charitably, explain this intention to the inventor.
2.12 The input can match \( T_1 \rightarrow R_1 \) over more than one length.

2.14 By enumerating all 256 possibilities: the pattern is not regular.

2.15 For example \( T \rightarrow \alpha(R^*)\beta \). The * means that 1 or more R's can be present. The dot after the R shows that one R has already been recognized. Now there are two hypotheses: 1. This was the one and only R; the dot leaves the \( R \) behind and moves in front of the \( \beta \); \( T \rightarrow \alpha(R^*)\beta \). 2. There is another R coming; the dot moves to the position immediately in front of the R; \( T \rightarrow \alpha(R^*)\beta \).

2.16 See Figure Answers.3.

\[
\begin{align*}
T \rightarrow \alpha\bullet(R_1 \& R_2 \& \ldots \& R_n)\beta &\Rightarrow \\
T \rightarrow \alpha\bullet R_1(R_2 \& R_3 \& \ldots \& R_n)\beta \\
T \rightarrow \alpha\bullet R_2(R_1 \& R_3 \& \ldots \& R_n)\beta \\
&\ldots \\
T \rightarrow \alpha\bullet R_n(R_1 \& R_2 \& \ldots \& R_{n-1})\beta
\end{align*}
\]

Figure Answers.3 ε move rule for the composition operator &.

2.17 Each round adds at least one dotted item to Closure set and there is only a finite number of dotted items.

2.18 There are 6 items that have the dot before a basic pattern or at the end:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>integral_number ( \rightarrow ) ((\bullet [0-9])^+)</td>
<td>(\lessssss\text{recognized})</td>
</tr>
<tr>
<td>integral_number ( \rightarrow ) (([0-9])^+)</td>
<td>(\lessssss\text{recognized})</td>
</tr>
<tr>
<td>fixed_point_number ( \rightarrow ) ((\bullet [0-9])^* \cdot . \cdot ([0-9])^+)</td>
<td>(\lessssss\text{recognized})</td>
</tr>
<tr>
<td>fixed_point_number ( \rightarrow ) ((([0-9])^* \cdot . \cdot ([0-9])^+)</td>
<td>(\lessssss\text{recognized})</td>
</tr>
<tr>
<td>fixed_point_number ( \rightarrow ) ((([0-9])^* \cdot . \cdot ([0-9])^+))</td>
<td>(\lessssss\text{recognized})</td>
</tr>
</tbody>
</table>

and so there are \(2^6=64\) subsets of 6 items.

2.20 See Figure Answers.4. Unfortunately, it appears that this is not in any way better than marking by character; on the contrary, the state array is probably larger.

\[
\begin{array}{c|c|c}
0 & (1, 0) & (2, 0) \\
1 & (1, 1) & (2, 1) \\
2 & (3, 2) & - \\
3 & (3, 3) & - \\
\end{array}
\]

Figure Answers.4 Fitting the strips with entries marked by state.

2.22 See Figure Answers.5. We first copy strings and characters; this avoids recognizing the StartComment inside them. Next we break up the string into safe chunks and keep track of where we are in a start condition <Comment>. Comments can contain * but not *:/; so we consume the *s one by one, and then match the final /'. This resets the start condition to INITIAL.

2.23 Close cooperation between lexical and syntax analyzer is required. As a kludge, preliminary skipping of the dynamic expression based on counting nested parentheses could be considered. Error recovery is a night-
Figure Answers.5 Lex filter for removing comments from C program text.

% Start Comment
Layout { [ \t] }
AnyQuoted { \\ }
StringChar { [\"\n\]\] }{AnyQuoted}
CharChar { [\"\n\]\] }{AnyQuoted}
StartComment { /* */ }
EndComment { /** */ }
SafeCommentChar { [\"\n] }
UnsafeCommentChar { /** */ }
%%
"{StringChar}\" { printf("%s", yytext); } /* string */
\'{CharChar}\' { printf("%s", yytext); } /* character */
{Layout}*{StartComment} { BEGIN Comment; }
<Comment>{SafeCommentChar}+ {} /* safe comment chunk */
<Comment>{UnsafeCommentChar} {} /* unsafe char, read one by one */
<Comment>\n* {} /* to break up long comments */
<Comment>{EndComment} { BEGIN INITIAL; }

2.25 Initially use a hash table of size \( N \), as usual, and implement it in an extensible array. When the table gets crowded, for example when there are more than \( N \) identifiers, extend the table to twice its size and adapt the hash function. Go through the occupied half of the table, and for each chain recompute the hash values of its elements. Since \( k \text{ MOD} (2N) \) is either \( k \text{ MOD} N \) or \( k \text{ MOD} N + N \) (prove!), the chain splits into two chains, one that stays where it was and one that will be located in the newly allocated part, at an address that is \( N \) higher. Both will be equally long on the average; this distributes the identifiers evenly. This process can be repeated as often as required, memory permitting.

2.26 It isn’t as simple as that. It depends on the amount of interaction of the macro processing with the lexical analysis of larger units, for example strings and comments. In C the scheme is hare-brained since it would require the macro processor to do almost full lexical analysis, to avoid substituting inside strings and comments. But in a language in which macro names have an easily recognizable form (for example in PL/I, in which macro names start with a \%, there is no such interference, and a better structuring of the compiler is obtained by a separate phase. But the loss in speed and the large memory requirements remain. Also, with full macro processing preceding compilation, it is very difficult to reconstruct the source text as the compiler user sees it.

2.28 Answer for \( N=3 \) in Figure Answers.6.

2.29 FigureAnswers.7 shows an LALR(1) suffix grammar for the grammar of Figure 2.84.

2.32 Addition is commutative, \( a + b = b + a \), but subtraction is not, \( a - b \neq b - a \). So, when \( 9 + 3 + 1 \) is incorrectly interpreted as \( 9 + (3 + 1) \), no great harm is done, but when \( 9 - 3 - 1 \) is incorrectly interpreted as \( 9 - (3 - 1) \), an incorrect answer results.

2.33 See Hanson (1985).

2.35 Collect invariants that hold when each routine is called and propagate them to the routines themselves.

2.37 (a) LL(1) and \( \epsilon \)-free. (b) Predictive is still more efficient.
GENERIC PROCEDURE F1(Type)(parameters to F1):
SET the Type variable Var TO ...;
// some code using Type and Var

GENERIC PROCEDURE F2(Type)(parameters to F2):
FUNCTION F1_1(parameters to F1_1):
INSTANTIATE F1(Integer);
FUNCTION F1_2(parameters to F1_2):
INSTANTIATE F1(Real);
SET the Type variable Var TO ...;
// some code using F1_1, F1_2, Type and Var

GENERIC PROCEDURE F3(Type)(parameters to F3):
FUNCTION F2_1(parameters to F2_1):
INSTANTIATE F2(Integer);
FUNCTION F2_2(parameters to F2_2):
INSTANTIATE F2(Real);
SET the Type variable Var TO ...;
// some code using F2_1, F2_2, Type and Var

Figure Answers.6 An example of exponential generics by macro expansion.

%token IDENTIFIER
%token EOF

input_suffix :
expression_suffix EOF | EOF ;
expression :
term | expression '+' term ;
expression_suffix :
term_suffix | expression_suffix '+' term | '+' term ;
term :
IDENTIFIER | '(' expression ')' ;
term_suffix :
expression ')' | expression_suffix ')' | ')' ;

Figure Answers.7 An LALR(1) suffix grammar for the grammar of Figure 2.84.

2.41 The recursion stack consists of a list of activation records, each of which defines an active routine; only the top one is running. Each activation record contains a continuation address (often called return address) telling where the routine should continue when it becomes the top node. The code from the continuation address to the end of the routine consists of zero or more routine calls. These calls represent what is being predicted and the corresponding grammar symbols are part of the prediction stack. Thus each activation record represents part of the prediction stack; the total prediction stack is the concatenation of all these parts, in the order of the activation records. Additional exercise: draw a picture that illustrates the above explanation in a clear way.

2.42 (a) S → aNb | Nc; N → ε. FOLLOW(N) = {b, c}, but initially, only c can follow N and after an a has been seen, only b can follow N.
(b) We predict a nullable alternative of N only if the input token is in the actual follow set; only then do we have the guarantee that the next input token will be matched eventually.
For \( \beta \rightarrow \beta_1 \ldots \beta_n \), stack \( n \) pairs \((\beta_i, \kappa)\), where \( \kappa = \text{FIRST}(-1 \ldots \beta_i) \) if \( \beta_{i+1} \ldots \beta_n \) does not produce \( \varepsilon \), and \( \kappa = \text{FIRST}(-1 \ldots \beta_i) \cup \sigma \) if \( \beta_{i+1} \ldots \beta_n \) does produce \( \varepsilon \).

(d) On input \( b \), the strong-LL(1) parser predicts \( N \rightarrow \varepsilon \), since \( b \) is in \( \text{FOLLOW}(N) \), but the full-LL(1) parser detects the error since \( b \) is not in \( \{a, \varepsilon\} \). Something similar happens on the input \( ac \) with the token \( c \).

(e) Suppose the grammar has a non-terminal \( N \) with a FIRST/FOLLOW conflict; then \( N \) has a nullable alternative, say \( A_j \), and there is a token, say \( t \), that is both in a FIRST set of an alternative, say \( A_k \), and in \( \text{FOLLOW}(N) \). Since \( t \) is in \( \text{FOLLOW}(N) \), there must be a full-LL(1) pair \((N, \sigma)\), with \( t \) in \( \sigma \). That pair has a full-LL(1) conflict, since we still do not know whether to predict \( A_j \) or the nullable alternative \( A_k \).

(f) \( S \rightarrow xNab \mid yNbb \mid N \rightarrow a \mid \varepsilon \). \( \text{FOLLOW}_2(N) = \{ab, bb\} \). Strong-LL(2) cannot decide between \( N \rightarrow a \{ab, bb\} \) which applies for look-aheads \( \{aa, ab\} \) and \( N \rightarrow \varepsilon \{ab, bb\} \) which applies for look-aheads \( \{ab, bb\} \). Full-LL(2) has separate prediction pairs \((N, ab)\), occurring after \( x \) and \( (N, bb) \), occurring after \( y \). So it can decide both between \( N \rightarrow a \{ab\} \) which applies for look-ahead \( \{aa\} \) and \( N \rightarrow \varepsilon \{ab\} \) which applies for look-ahead \( \{ab\} \), and between \( N \rightarrow a \{bb\} \) which applies for look-ahead \( \{ab\} \) and \( N \rightarrow \varepsilon \{bb\} \) which applies for look-ahead \( \{bb\} \).

2.43 See Figure Answers.8.

---

<table>
<thead>
<tr>
<th>stack continuation</th>
<th>FIRST set</th>
</tr>
</thead>
<tbody>
<tr>
<td>paremthesized_expression rest_expression EOF</td>
<td>'{' '{' }</td>
</tr>
<tr>
<td>(' expression ') rest_expression EOF</td>
<td>'{' }</td>
</tr>
<tr>
<td>expression ')' rest_expression EOF</td>
<td>IDENTIFIER '{' '}'</td>
</tr>
<tr>
<td>term rest_expression ')' rest_expression EOF</td>
<td>IDENTIFIER '{' '}'</td>
</tr>
<tr>
<td>IDENTIFIER rest_expression ')' rest_expression EOF</td>
<td>IDENTIFIER '{' '}'</td>
</tr>
<tr>
<td>rest_expression ')' rest_expression EOF</td>
<td>'{' '+' '}'</td>
</tr>
<tr>
<td>'}' rest_expression EOF</td>
<td>'{' '+' '}'</td>
</tr>
<tr>
<td>rest_expression EOF</td>
<td>EOF</td>
</tr>
<tr>
<td>EOF</td>
<td>EOF</td>
</tr>
</tbody>
</table>

Figure Answers.8 Stack continuations with their FIRST sets.

Acceptable set: \{ '{' '{' '}' '}' IDENTIFIER EOF \}.

2.44 The acceptable set of a non-terminal \( N \) is the union of FIRST(\( N \)) and the acceptable set of the shortest alternative of \( N \). So, the acceptable sets of all non-terminals can be precomputed using a closure algorithm. Now, if the prediction stack is available directly (as an array or a linked list), we can traverse the stack and compute the union of the acceptable sets of the symbols in it. In \( LLgen \), however, the prediction stack is just the \( C \) stack and is not available for traversal. \( LLgen \) keeps an integer array indexed by grammar symbols counting how many times a given symbol is present on the stack. This information is easily maintained and suffices to compute the acceptable set.

2.45 Since the input ends in a EOF token, \( \alpha \) must consist of zero or more grammar symbols, followed by EOF or it would never match the EOF. The imaginary parser steps forced by an empty input remove the grammar symbols one by one, leaving the single EOF as the last stack configuration. Its FIRST set contains EOF.

2.48 (a) When the ACTION table calls for a ‘reduce using rule \( N \rightarrow \alpha \)’, the item set corresponding to the state on the top of the stack contains the item \( N \rightarrow \alpha \). The dot can only be at the end of \( \alpha \) when it has just passed over the last member of \( \alpha \), which must therefore be just below the top state on the stack. This reasoning applies successively to all other members of \( \alpha \), which must therefore also be on the stack.

(b) The item set preceding \( \alpha \) on the stack must contain the item \( N \rightarrow \alpha \), or no \( \alpha \) would be recognized and no item \( N \rightarrow \alpha \) would eventually be found. The item \( N \rightarrow \alpha \) must have originated from some item \( P \rightarrow \beta \). The presence of this item guarantees that a transition on \( N \) is possible, leading to a state that includes \( P \rightarrow \beta \).
2.49 A value ‘shift’ in an ACTION table entry does not conflict with another ‘shift’ value in that same entry, but a ‘shift’ and a ‘reduce’ do. So do a ‘reduce’ and another ‘reduce’, since they are actually two different ‘reduces’: ‘reduce to $M$’ and ‘reduce to $N$’.

2.50 See Figures Answers.9, Answers.10, and Answers.11. The LR(0) automaton is identical to those in Figures Answers.9 and Answers.11 with the look-ahead sets removed and does not need to be shown here.

---

2.51 The tree has the form $[ (x \ ]^{n-1} (x) ]^{n-1}$ and the last $x$ is the first handle, in any bottom-up parser. So all the $[ (x \ ]^{n-1}$ must be stacked.

2.52 (e) Yes, if $A \rightarrow P \ Q \ | \ Q$ and $Q \rightarrow A \ | \ ....$

2.54 See Figure Answers.12.

This grammar is unambiguous and will pair the $else$ to the nearest unpaired $if$.

2.55 After rule 2, add: ‘If $t$ and $u$ are the same operator: if the operator is left-associative, reduce, otherwise shift.’

2.57 When meeting empty input, the only stack element is the state $S_n$, which is easily derived from that in Figure 2.97 and which is shown in Figure Answers.13. No elements are removed from the stack since there is already an error-recovering state on top. Next a dummy node erroneous_A (called err_A in the diagram) is constructed and stacked. This results in state $S_n$ to be stacked. Since EOF ($\$) is the next input token, erroneous_A is reduced to A, which is stacked. Parsing then proceeds as normal. The resulting parse tree is $S \rightarrow A \rightarrow erroneous_A$.

2.58 In a pure bottom-up parser no such pointers exist: trees are constructed before their parents, and the only pointer to a tree is the one on the stack that is used to discard the tree; the stack entry that contains it is removed by the recovery process. If other pointers have been created outside the parsing mechanism, these must be found and zeroed.
Figure Answers.10 The LR(1) automaton for $S \rightarrow x \; S \; x \mid x$.

Figure Answers.11 The LALR(1) automaton for $S \rightarrow x \; S \; x \mid x$. 
if_statement → short_if_statement | long_if_statement
short_if_statement → 'if' '(' expression ')' statement
long_if_statement → 'if' '(' expression ')' statement_but_not_short_if 'else' statement
statement_but_not_short_if → long_if_statement | other_statement
statement → if_statement | other_statement
other_statement → ...

Figure Answers.12 An unambiguous grammar for the if-then-else statement.

Answers for Chapter 3

3.1 (a) N (non-terminal); (b) N; (c) P (production rule); (d) P; (e) N; (f) P; (g) P; (h) N.

3.2 For a non-terminal N, some of its production rules could set some attributes and other rules could set other attributes. Then the attributes in a tree with a node for N in it could be evaluable for one production (tree) of that N, and not for another. This destroys the composability of context-free grammars, which says that
anywhere an $N$ is specified, any production of $N$ is acceptable.

### 3.3

The topological sort algorithm of Figure 3.16 will fail with infinite recursion when there is a cycle in the dependencies. A modified algorithm with cycle detection is given in Figure Answers.14.

```plaintext
FUNCTION Topological sort of (a set Set) RETURNING a list:
    SET List TO Empty list;
    SET Busy list TO Empty list;
    WHILE there is a Node in Set but not in List:
        Append Node and its predecessors to List;
    RETURN List;

PROCEDURE Append Node and its predecessors to List:
    // Check if Node is already (being) dealt with; if so, there
    // is a cycle:
    IF Node is in Busy list:
        Panic with "Cycle detected";
        RETURN;
    Append Node to Busy list;
    // Append the predecessors of Node:
    FOR EACH Node_1 IN the Set of nodes that Node is dependent on:
        IF Node_1 is not in List:
            Append Node_1 and its predecessors to List;
    Append Node to List;
```

**Figure Answers.14** Outline code for topological sort with cycle detection.

### 3.4

See Figures Answers.15 and Answers.16.

![Dependency graphs for S, A, and B.](image)

**Figure Answers.15** Dependency graphs for $S$, $A$, and $B$.

### 3.5

See Figures Answers.17 and Answers.18.

### 3.6

(a) Figure Answers.19 shows the dependency graph of $S$, Figure Answers.20 the IS-IS graph of $S$.

(b) $\{(i_1, i_2), \{s_1, s_2\}\}$. 
Figure Answers.16 IS-SI graph of A.

---

Figure Answers.17 Dependency graphs for S and A.

(c) 1 routine:

parent prepares S.i1 and S.i2:

PROCEDURE Visit_1 to S (i1, i2, s1, s2):
    // S.i1 and S.i2 available →
    SET U.i TO f2(S.i2); // → U.i available
    Visit_1 to U (U.i, U.s); // → U.s available
    SET T.i TO f1(S.i1, U.s); // → T.i available
    Visit_1 to T (T.i, T.s); // → T.s available
    SET S.s1 TO f3(T.s); // → S.s1 available
    SET S.s2 TO f4(U.s); // → S.s2 available

or any other dependency-conforming order. (Exercise: find another order).

3.7 (a) See Figure Answers.21.
(b) \((\{i2, s2\}), (\{i1, s1\})\).
(c) 2 routines:
IS-SI graph set of $A$:

merged IS-SI graph of $A$:

**Figure Answers.18** IS-SI graph sets and IS-SI graph of $A$.

**Figure Answers.19** Dependency graph of $S$.

**Figure Answers.20** IS-SI graph of $S$.

parent prepares $S.i2$:

PROCEDURE Visit_1 to $S$ $(i_1, i_2, s_1, s_2)$:

// $s.i2$ available →
SET $U.i$ TO $f_2(s.i2)$; // $\rightarrow U.i$ available
Visit_1 to $U$ $(U.i, U.s)$; // $\rightarrow U.s$ available
SET $S.s2$ TO $f_4(U.s)$; // $\rightarrow S.s2$ available

parent receives $S.s2$
3.8 From the absence of SI arrows in the IS-SI graph of $S$ we can conclude that all inherited attributes are available when $S$ is visited; so there is no L-attribute problem there. The call to $T$, however, in $\text{Visit}_1$ to $S()$ must now precede the one to $U$. We cannot supply it with $T.i$, though, since that is available only after the visit to $U$. So we pass a dummy parameter to $T$ and extend $T$ with another synthesized attribute $T.cont_i$, which contains a representation of the computations to be performed when $T.i$ is known. Depending on the implementation language of the attribute evaluator, this representation could have the form of an expression tree, a continuation routine, or perhaps just an ad-hoc indication. Anyway, its activation results in $T.s$ to be computed; we assume a function $\text{Compute}()$ to be available for the purpose. Note that $T.cont_i$ may be very complicated since $T$ may have been forced to pass the problem on to its children, and so on. All this results in the routine $\text{Visit to } S()$ shown in Figure Answers.22.

3.9 The fact that no intervening visits to other children are needed shows that the production rule already has all the information for the second visit. This may, however, not be the case in all production rules that have this type of child, so other production rules may require two non-consecutive visits.

3.10 See Figure Answers.23. Here, $\text{Concat}$ concatenates a character representation to a string, and $\text{Checked number value}$ converts a string to an integer, checking each digit in the string.
3.12 Hints: For each rule for each non-terminal $N$, do the following. Turn all inherited attributes of the children of $N$ into local variables of $N$. If a child used to get an inherited attribute $i$ and returned a synthesized attribute $s$, it now returns a function to be called with the value of $i$ once it becomes available, which yields the value of $s$. If $i$ does not come available inside $N$, $s$ cannot be computed now, and a new routine is created to be passed on upwards.

3.13 See Figure Answers.24.

3.14 See Figure Answers.25.

3.15 See Figure Answers.26, and note that the code is a simplification over that from Figure 3.37.

3.16 Pass the list (stack representation) through the condition, since the condition is the first to be executed at run time. Keep a copy of the resulting list, pass the resulting list through the body of the while statement, and merge with the copy. This combines the possibilities of zero and multiple passes through the loop at run time.

3.17 Pass the list (stack representation) through the body of the repeat-until statement, since the body is the first to be executed at run time. Then pass the resulting list through the condition of the repeat-until statement, since it will always be executed at run time.

3.18 We need two variables, the actual number needed here and a high-water mark. Simple symbolic interpretation suffices.

3.19 The successor of the then-part is the merge node at the end of the if-statement rather than its else-part, and it
is correct that we enter that node with an empty list during symbolic interpretation, since we will never reach the end of the if-statement from the end of the then-part when the program is run, due to the intervening jump. Full symbolic interpretation works on the threaded AST rather than on the linear program text.

3.20 Simple symbolic interpretation touches the program text only once, so it runs in linear time. Full symbolic interpretation involves repetition until convergence, which may in principle be non-linear. But as with most closure algorithms, the number of repetitions involved barely depends on the size of the input. Three to four repetitions are almost always sufficient to reach convergence, so full symbolic interpretation too runs in linear time, albeit with a larger constant factor. The same applies to the data-flow equations.

3.21 It violates requirement 4 in Section 3.2.2.1: the actions to be taken on constants do not subsume those taken on variables. Quite to the contrary, any constant can be handled by code generated for variables but not the other way around.
3.22 There is no way to represent the value $V$ in the $IN$ and $OUT$ sets nor can it be propagated using fixed $KILL$ and $GEN$ sets.

3.23 For each IN parameter $KILL = GEN = \emptyset$, for each INOUT or OUT parameter $KILL = 10$, $GEN = 01$.

3.24 $x$ becomes initialized. Con: It sounds unreasonable and counterintuitive to get a variable initialized by assigning the value of an uninitialized variable to it. Pro: The error in the program is probably the lack of initialization of $y$: the further usage of $x$ is independent of this error. Since a warning is already given on the assignment, no further warnings on subsequent – probably correct – uses of $x$ seem appropriate.

3.25 (a) Note that the meet operator must be intersection, because if the expression is to be very busy at a point, it must be evaluated on all paths going through this point. The equations are:

$$\text{OUT}(N) = \bigcap_{M \text{ minimal occurrence of } N} \text{IN}(M)$$

$$\text{IN}(N) = (\text{OUT}(N) \setminus \text{KILL}(N)) \cup \text{GEN}(N)$$

(b) The expression is killed by an assignment to any of its operands. The $GEN$ and $KILL$ bits for $x \times x$ are given below.

$$\begin{align*}
\text{GEN}(1) &= 0 & \text{KILL}(1) &= 1 \\
\text{GEN}(2) &= 0 & \text{KILL}(2) &= 0 \\
\text{GEN}(3) &= 1 & \text{KILL}(3) &= 0 \\
\text{GEN}(4) &= 0 & \text{KILL}(4) &= 0 \\
\text{GEN}(5) &= 0 & \text{KILL}(5) &= 0 \\
\text{GEN}(6) &= 0 & \text{KILL}(6) &= 0
\end{align*}$$

(c) Solving the equations gives:
\[ \text{IN}(1) = 0 \quad \text{OUT}(1) = 1 \\
\text{IN}(2) = 1 \quad \text{OUT}(2) = 1 \\
\text{IN}(3) = 1 \quad \text{OUT}(3) = 0 \\
\text{IN}(4) = 0 \quad \text{OUT}(4) = 0 \\
\text{IN}(5) = 0 \quad \text{OUT}(5) = 0 \\
\text{IN}(6) = 0 \quad \text{OUT}(6) = 0 \]

The conclusion is that the evaluation of \( x \times x \) can be moved to the position between statements 1 and 2, thus moving it out of the loop.

3.26 Consider any routine with a flow-of-control graph that is a linear list from routine entry to routine exit. Whatever the contents of the KILL and GEN sets, the IN and OUT sets will be computed in one scan through the list, and there is no way to transport the information about the routine exit back to the last use of a variable.

### Answers for Chapter 4

4.1 Base the recursive descent on the rule that the shortest distance from a node \( N \) to a leaf is one plus the minimum of the shortest distances from the children of \( N \) to the leaf. The recursive descent process will visit the nodes close to leaves exponentially often, causing the algorithm to be exponential in the size of the graph.

Now add memoization by saving the distance found in each node. Now no value needs to be recomputed, and each node is visited only once, resulting in a linear-time algorithm. Enjoy the spectacular speed-up!

4.2 See Figure Answers.27.

```plaintext
CASE Operator type:
  SET Left value TO Pop working stack();
  SET Right value TO Pop working stack();
  SELECT Active node .operator:
    CASE '+' : Push working stack (Left value + Right value);
    CASE '*' : ... 
    CASE ... 
  SET Active node TO Active node .successor;

Figure Answers.27 Iterative interpreter code for operators.
```

4.3 In principle, recursive interpreters, iterative interpreters, and compiled code can give the same error messages; the difference lies in the amount of work involved in producing the message, and therewith the likelihood that the work will be expended and the message produced.

Recursive interpreters usually retain and continually update the full symbol table and can, at any moment, produce a fully symbolic snapshot of all data used by the program, including the calling stack. They also test the input values to any operation extensively and catch erroneous values before disaster strikes.

Iterative interpreters usually retain the symbol table but do not use it as a database for storing the data. A snapshot will have to be reconstructed from the bare data, matched to the symbol table. Input data to operations are still checked extensively.

Compiled code usually does not retain the symbol table, and a separate program may be needed to pry information from the unstructured data found in memory. The testing of input data to operations is usually left to machine instructions, which may or may not react to errors by producing traps or erroneous values. Traps are caught by code which has little knowledge of what went wrong; erroneous values may propagate.
4.4 A self-extracting archive works exactly like the ‘compiler’ of Section 4.2.1: there, the executable file contains both the AST and the interpreter. A self-extracting archive contains both the contents of the archive and the extraction code. Often, the archive is compressed, and the extraction code also contains decompression code.

4.5 The program:

```c
int Program[] = {'D',7,'D',1,'D',5,'P','+','P','*','!',0};
```

The interpreter:

```c
int main(void) {
    int PC = 0;
    int Instruction;
    while ((Instruction = Program[PC++]) != 0) {
        switch (Instruction) {
            case 'D': Expression_D(Program[PC++]); break;
            case 'P': Expression_P(Program[PC++]); break;
            case '!': Print(); break;
        }
    }
    return 0;
}
```

4.6 Instead of a routine `Expression_P`, we could have two routines `Expression_P_43` and `Expression_P_42`, with bodies

```c
void Expression_P_43(void) {
    int e_left = Pop(); int e_right = Pop();
    Push(e_left + e_right);
}
```

```c
void Expression_P_42(void) {
    int e_left = Pop(); int e_right = Pop();
    Push(e_left * e_right);
}
```

and generate

```c
Expression_P_43();
Expression_P_42();
```

instead of the `Expression_P` calls in Figure 4.14.

4.7 (a) As we did for the register machine, we generate code for the heaviest tree first, but now, we can only do this for commutative operators, because we cannot exchange operands. The weight computation must account for this too, as shown in Figure Answers.28.

(b) The resulting code sequence:

```c
Push_Local #b
Push_Local #b
Mult_Top2
Push_Local #a
Push_Local #c
Mult_Top2
Push_Const 4
Mult_Top2
Subtr_Top2
```
FUNCTION Weight of (Node) RETURNING an integer:
    SELECT Node .type:
    CASE Constant type: RETURN 1;
    CASE Variable type: RETURN 1;
    CASE ...
    CASE Add type:
        SET Required left TO Weight of (Node .left);
        SET Required right TO Weight of (Node .right);
        IF Required left > Required right: RETURN Required left;
        IF Required left < Required right: RETURN Required right;
        // Required left = Required right
        RETURN Required left + 1;
    CASE Sub type:
        SET Required left TO Weight of (Node .left);
        SET Required right TO Weight of (Node .right);
        IF Required left > Required right: RETURN Required left;
        RETURN Required right + 1;
    CASE ...

Figure Answers.28 Adapted weight function for minimizing the stack height.

4.8 See Figure Answers.29.

FUNCTION Weight of (Node, Left or right) RETURNING an integer:
    SELECT Node .type:
    CASE Constant type: RETURN 1;
    CASE Variable type:
        IF Left or right = Left: RETURN 1;
        ELSE Left or right = Right: RETURN 0;
    CASE ...
    CASE Add type:
        SET Required left TO Weight of (Node .left, Left);
        SET Required right TO Weight of (Node .right, Right);
        IF Required left > Required right:
            RETURN Required left;
        IF Required left < Required right:
            RETURN Required right;
        // Required left = Required right
        RETURN Required left + 1;
    CASE ...

Figure Answers.29 Revised weight function for register-memory operations.

4.9 See Figure Answers.30; the 'then' gets 0.7, the 'else' 0.3; loop skipping gets 0.1, loop entering 0.9; the cases get 0.4, 0.4, 0.2. Traffic at routine entry is arbitrarily set to 1. The first column gives the 17 equations; all can be solved by simple substitution, except those for e, f, and g, which need elimination. The results are given in the second column. Note that we predict that for every time the routine is called, the loop body A will be executed 6.3 times. Also note that the traffic out of the routine is again 1; what goes in must come out.
### 4.10
The flow graph is in Figure Answers.31 and the equations and their solution in Figure Answers.32. Gaussian elimination will find the solution, but simple substitution suffices here: substitute $h = j$ in $j = 0.9(g+h)$, yielding $h = 0.9(g+h)$; solving for $h$ yields $h = 9g$; substituting this in $i = 0.1(g+h)$ yields $i = g$ (what enters the second while loop must come out); we also have $g = f = e = d, i = b$, and $a = 1$. Substituting all this in $d = 0.9(a+b)$ yields $g = 0.9(1+g)$, which after solving leads to $g = 9$. Note that $c = 1$. Why is that important? The rest is straightforward substitution.

### 4.11
These dependencies also express the requirement that all assignments to a variable are executed in sequential, left-to-right order.

### 4.12
See Figures Answers.33 and Answers.34.

### 4.13
(a) See Figure Answers.35.
(b) The input $p$ of the second `*p++` is dependent on the output $p$ of the first `*p++` and so its dependencies differ from those of the input $p$ of the first `*p++`.

### 4.15
See Figure Answers.36.

### 4.16
$S$ and $N$ cannot be the same node, since that would make the dependency graph contain a cycle because $S$ refers to $N$.

### 4.17
A ladder sequences starts at each graph root, except when that root has an incoming dependency. Not all roots can have incoming dependencies, or the dependency graph would be cyclic.

### 4.18
Doing so will destroy the contents of register $I1$ and may lead to incorrect code if $I1$ is still used in another operation.

### 4.19
First code $x, +, +$ into
Figure Answers.31 Flow graph for static profiling of a nested while loop.

<table>
<thead>
<tr>
<th>Equation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 1</td>
<td>1.0</td>
</tr>
<tr>
<td>b = i</td>
<td>9.0</td>
</tr>
<tr>
<td>c = 0.1 (a+b)</td>
<td>1.0</td>
</tr>
<tr>
<td>d = 0.9 (a+b)</td>
<td>9.0</td>
</tr>
<tr>
<td>e = d</td>
<td>9.0</td>
</tr>
<tr>
<td>f = e</td>
<td>9.0</td>
</tr>
<tr>
<td>g = f</td>
<td>9.0</td>
</tr>
<tr>
<td>h = j</td>
<td>81.0</td>
</tr>
<tr>
<td>i = 0.1 (g+h)</td>
<td>9.0</td>
</tr>
<tr>
<td>j = 0.9 (g+h)</td>
<td>81.0</td>
</tr>
</tbody>
</table>

Figure Answers.32 Traffic equations and their solution for Figure Answers.31.

Load_Reg R2,R1  
Add_Reg R3,R1  
Add_Reg R4,R1  
Store_Reg R1,x

yielding Figure Answers.37. Next code \(y, +, -\) into
Load_Reg R2,R1
Subtr_Reg R3,R1
Add_Reg R4,R1
Store_Reg R1,y

yielding Figure Answers.38. Then code R3, *, * into

Load_Const 2,R1
Mult_Mem a,R1
Mult_Mem b,R1
Store_Reg R1,R3
Figure Answers.35 The dependency graph of the expression *p++.

Figure Answers.36 The data dependency graph of Figure Answers.34 in triple representation.

<table>
<thead>
<tr>
<th>position</th>
<th>triple</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a * a</td>
</tr>
<tr>
<td>2</td>
<td>a * 2</td>
</tr>
<tr>
<td>3</td>
<td>@2 * b</td>
</tr>
<tr>
<td>4</td>
<td>@1 + @3</td>
</tr>
<tr>
<td>5</td>
<td>b * b</td>
</tr>
<tr>
<td>6</td>
<td>@4 + @5</td>
</tr>
<tr>
<td>7</td>
<td>@6 =: x</td>
</tr>
<tr>
<td>8</td>
<td>@1 - @3</td>
</tr>
<tr>
<td>9</td>
<td>@8 + @5</td>
</tr>
<tr>
<td>10</td>
<td>@9 =: y</td>
</tr>
</tbody>
</table>

yielding Figure Answers.39. Then code R2, * into

Load_Mem a,R1  
Mult_Mem a,R1  
Store_Reg R1,R2

yielding Figure Answers.40. Finally code R4, * into

Load_Mem b,R1  
Mult_Mem b,R1  
Store_Reg R1,R4

resulting in the empty dependency graph.

4.20  
1. Step 2 assigns registers to nodes. If the top node of a ladder has already been assigned to a (pseudo-)register, use that register instead of R1, and mark the (pseudo-)register so it will be given a real register. Note that you can do this only a limited number of times, and that this procedure interferes with further register allocation.  
2. If the node M in step 2 is the bottom of the ladder, use R as the ladder register in step 3.

4.22  
It would be useful since it would for example add the pattern trees:

cst*(reg*reg), (reg*reg)*cst
4.23 Suppose the token set \{a, bcd, ab, c, d\} and the input abcd. Immediately returning the a yields 2 tokens, whereas 3 can be achieved, obviously. Assume the entire input is in memory. Record in each item its starting position and the number of tokens recognized so far. At each reduce item that says that \(N\) tokens have been recognized, add the Initial item set with token counter \(N+1\) and the present location as starting point. Having arrived at the end of the input, find the reduce item with the largest token counter and isolate the token it identifies. Work backwards, identifying tokens.

4.25 See Figure Answers.41 for the tree. A label evicts another label in the dynamic programming part when its rewrite shows a gain on the cost or register usage scale, and no loss on the other scale. The resulting code is shown in Figure Answers.42.

4.26 Remove d, e, a, b, c. Add e with color 1, b with color 2, a with color 3, e with color 2, d with color 1.
4.27 (a) See Figure Answers.43.  
(3).

4.28 For the first instruction in the sequence we have $20 \times 2 = 40$ combinations, using $R1, R1$ and $R1, R2$, or more compactly $\{R1\}, \{R1, R2\}$. For the second instruction we have $20 \times 2 \times 3 = 120$ combinations, using $\{R1, R2\}, \{R1, R2, R3\}$; for the further instructions we have $20 \times 3 \times 3 = 180$ combinations each. In total
Load_Const 8, R ; 1 unit
Mult_Mem a, R ; 6 units
Store_Reg R, tmp ; 3 units
Load_Const 4, R ; 1 unit
Mult_Mem tmp, R ; 6 units
Store_Reg R, tmp ; 3 units
Load_Mem b, R ; 3 units
Add_Mem tmp, R ; 3 units
Total = 26 units

Figure Answers.42 Code generated by bottom-up pattern matching for 1 register.

a ─────── tmp_2ab ─── tmp_bb ─── x ─── y  
     |                     |        |
     v                     v        v
b ─── tmp_aa

Figure Answers.43 The register interference graph for Exercise 4.27.

4800x180^{N-2} combinations.

Estimating a weekend at 2.5 days, each of about 80,000 seconds, we have about 2x10^{11} seconds, or 2x10^{10} tests. So we want the largest N for which 4800x180^{N-2} is smaller than 2x10^{10}. Now, 4800x180^4=1.5x10^8 and 4800x180^5=2.7x10^{10}, so N=4.

4.30 (a) Since the AST of P() actually corresponds to the C code

```c
void P(int i) { if (i < 1) goto _L_end; else Q(); _L_end; }
```

we get

```c
(int i = 0; if (i < 1) goto _L_end; else Q(); _L_end;)
```

(b)

```c
(int i = 0; if (0 < 1) goto _L_end; else Q(); _L_end;)
```

(c)

```c
(int i = 0; if (1) goto _L_end; else Q(); _L_end;)
```

(d)

```c
(int i = 0; goto _L_end; _L_end;)
```

(e) Elimination of unused variables. The information whether a variable is unused can be obtained through the techniques on checking the use of uninitialized variables described in Section 3.2.2.1.

4.34 Advantages of PC-relative addressing modes and instructions are:
- they require no relocation, thus reducing the work of the linker;
- they allow position-independent code, code that can be loaded anywhere in memory, without any modifications;
- they may allow shorter instructions: an offset may fit in a byte whereas a full address usually does not. Even if it does, the assembler still has to reserve space for a full address, because the linker may modify it.

4.35 An unlimited conditional jump can be translated to a conditional jump with the contrary condition over an unconditional jump. For example,
Jump_Not_Equal label1  # jump to label1 if not equal

Jump_Equal L1  # jump over Jump if equal
Jump label1
L1:

Answers for Chapter 5

5.1 Assume Beginning of available memory is a multiple of 32. In Figure 5.2:

SET the polymorphic chunk pointer First_chunk pointer TO
Beginning of available memory + 28;
SET First_chunk pointer .size TO
(Size of available memory - 28) / 32 * 32;

In Figure 5.3:

SET Requested chunk size TO (Block size + 3) / 32 * 32 + 32;

5.2 The garbage collector will free chunks only when they are unreachable. If they are unreachable they cannot
be freed by the user since the user does not have a pointer to them any more. So it is safe to call the garbage
collector from Malloc().

5.3 One can keep track of all calls to malloc() and free() during program execution, and analyze these
calls with a post-mortem program. Several leak-finding tools exist; they typically use modified versions of
malloc() and free() that write information about these calls to a file; the information typically
includes the address and size of the block, and a stack trace. The file is processed by an off-line program,
that finds blocks of memory that have been malloced but not freed; the program then also prints the rou-
tines from which malloc() was called (using the stack trace information).

5.5 How do you find this counter starting from the pointer to the record and how do you get the pointer by
which to return the block?

5.6 In the allocation of the arrays into which the code and data segments will be compiled; perhaps in the allo-
cation of the external symbol table.

5.7 In general, the garbage collection algorithms inspect pointer values, which will be (simultaneously) changed
by the application. Some garbage collection algorithms (for example two-space copying and compaction)
copy data, which is dangerous if the application can access the data in the meantime. Some algorithms
assume that the garbage collection only becomes active at specific points in the program (see Section 5.2.2),
which is difficult to guarantee with a concurrent garbage collection.

5.8 Garbage collection algorithms will not handle this correctly, since they are unable to determine that the
block that p (or rather p & 0x7fffffff) points to is still reachable. Even conservative garbage collect-
ion will fail; in fact, it will probably not even recognize p as a pointer variable.

5.9 If the assignment p:=p is not optimized away, if p points to a chunk P with reference count 1, if P contains
a pointer to another chunk Q and if the reference count of Q is also 1, then first decreasing the reference
count of P causes P to be freed, which causes Q to be freed. Subsequently increasing the reference count of
P will not raise the reference count of Q again and the pointer to Q in P will be left dangling. Also, on some
systems freeing P might cause compaction to take place, after which the chunk P would be gone entirely
and incrementing its reference count would overwrite an arbitrary memory location.

5.10 (c) Call number n to A will avoid the nodes that have been marked already by calls number 1...n−1 and will
thus work on a smaller unmarked graph.

5.11 Refer to Figures Answers.44 and Answers.45. Introduce a global pointer Scan pointer, which points
to the first chunk that has not yet been scanned. Coalescing has been incorporated into the function
Pointer to free block of size (Block size), which starts coalescing immediately and starts scanning at the chunk pointed to by Scan pointer. If that does not work, perform the marking phase of the garbage collection only, and scan, coalesce and search from the beginning.

SET the polymorphic chunk pointer Scan pointer TO Beginning of available memory;

FUNCTION Malloc (Block size) RETURNING a polymorphic block pointer:
SET Pointer TO Pointer to free block of size (Block size);
IF Pointer /= Null pointer: RETURN Pointer;
Perform the marking part of the garbage collector;
SET Scan pointer TO Beginning of available memory;
SET Pointer TO Pointer to free block of size (Block size);
IF Pointer /= Null pointer: RETURN Pointer;
RETURN Solution to out of memory condition (Block size);

FUNCTION Pointer to free block of size (Block size) RETURNING a polymorphic block pointer:
// Note that this is not a pure function
SET Chunk pointer TO First_chunk pointer;
SET Requested chunk size TO Administration size + Block size;
WHILE Chunk pointer /= One past available memory:
  IF Chunk pointer >= Scan pointer:
    Scan chunk at (Chunk pointer);
    IF Chunk pointer .free:
      Coalesce with all following free chunks (Chunk pointer);
      IF Chunk pointer .size - Requested chunk size >= 0:
        // large enough chunk found:
        Split chunk (Chunk pointer, Requested chunk size);
        SET Chunk pointer .free TO False;
        RETURN Chunk pointer + Administration size;
    // try next chunk:
    SET Chunk pointer TO Chunk pointer + Chunk pointer .size;
  RETURN Null pointer;

Figure Answers.44 A Malloc() with incremental scanning.

5.12 See Siklóssy (1972).

5.13 Similarities:
(1) both gather reachable nodes and copy them to the beginning of a memory segment;
(2) the resulting free space is a single contiguous block of memory;
(3) the nodes change place so all pointers have to be relocated.
Differences:
(1) one versus two spaces (obviously);
(2) depth-first (compaction) versus breadth-first (copying) traversal;
(3) compaction touches all nodes in the heap when scanning for free nodes, copying touches reachable nodes only;
(4) compaction uses a separate phase to compute the new addresses, copying computes them on-the-fly;
(5) compaction requires an additional pointer per node to hold the ‘new’ address, copying stores the forward address in the ‘old’ node, thus avoiding space overhead per node.

5.14 The ‘overlapping lists’ in the paper are dags.
PROCEDURE Scan chunk at (Chunk pointer):
    IF Chunk pointer .marked = True:
        SET Chunk pointer .marked TO False;
    ELSE Chunk pointer .marked = False:
        SET Chunk pointer .free TO True;
    SET Scan pointer TO Chunk pointer + Chunk pointer .size;
PROCEDURE Coalesce with all following free chunks (Chunk pointer):
    SET Next_chunk pointer TO Chunk pointer + Chunk pointer .size;
    IF Next_chunk pointer >= Scan pointer:
        Scan chunk at (Next_chunk pointer);
    WHILE Next_chunk pointer /= One past available memory
        AND Next_chunk pointer .free:
            // Coalesce them:
            SET Chunk pointer .size TO
               Chunk pointer .size + Next_chunk pointer .size;
            SET Next_chunk pointer TO Chunk pointer + Chunk pointer .size;
        IF Next_chunk pointer >= Scan pointer:
            Scan chunk at (Next_chunk pointer);

Figure Answers.45  Auxiliary routines for the Malloc() with incremental scanning.

Answers for Chapter 6

6.1 Values of type unsigned integer are always $\geq 0$.

6.2 In most languages we need to know if an identifier is a keyword or the name of a macro, long before its name space is known. If we want to postpone the identification to the time that the proper name space is known, we will need other mechanisms to solve the keyword and macro name questions.

6.3 The declarations are equivalent to:

```
type t1 = array[1..10] of int;
type t2 = array[1..10] of int;
type t3 = array[1..10] of int;
```

So $A$ and $B$ have the same type, $B$ and $C$ have different types (which are also different from the type of $A$ and $B$).

6.6 We have $rvalue ? V : V \rightarrow rvalue$. In principle, $rvalue ? ivalue : ivalue$ could yield an $lvalue$, but ANSI C defines it to yield an $rvalue$. In GNU C an $lvalue$ results, but a warning is given under the `pedantic' flag.

6.7 Can’t be. The last scope rule forbids the creation of such values.

6.9 (a) Size is 24, alignment is 8.
(b) Size is 16, alignment is 8.
(c) This depends on what the language manual specifies, but even if the language manual does not forbid the reorganization of record fields, there are several reasons why a programmer may not want this: interoperability between languages, the modeling of I/O devices, etc.

6.10 See Figure Answers.46 and Answers.47.

6.11 Set union: the complication here is that set elements must be represented only once in the linked list. So, the set union of two sets is constructed by first copying the first set, and then adding elements of the second set to the list, but only if they are not present in the first set.

Set intersection: the set intersection of two sets is constructed by creating an empty result set, and then
\[ zeroth_{\text{offset}}(A) = -(L_{B1} \times L_{\text{PRODUCT}_1} + L_{B2} \times L_{\text{PRODUCT}_2} + \ldots + L_{Bn} \times L_{\text{PRODUCT}_n}) \]

Figure Answers.46 Formula for \( zeroth_{\text{offset}}(A) \).

\[ \text{base}(A) + zeroth_{\text{offset}}(A) + i_1 \times L_{\text{PRODUCT}_1} + \ldots + i_n \times L_{\text{PRODUCT}_n} \]

Figure Answers.47 The address of \( A[i_1, \ldots, i_n] \).

checking for each element in the first set, whether it is also a member of the second set; if it is, we add it to the result set, if it is not, we ignore it.

6.12 See Figure Answers.48.

<table>
<thead>
<tr>
<th>method table for Rectangle</th>
<th>method table for Square</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsShape_Shape_Shape</td>
<td>IsShape_Shape_Shape</td>
</tr>
<tr>
<td>IsRectangle_Shape_Rectangle</td>
<td>IsRectangle_Shape_Rectangle</td>
</tr>
<tr>
<td>IsSquare_Shape_Shape</td>
<td>IsSquare_Shape_Square</td>
</tr>
<tr>
<td>SurfaceArea_Shape_Rectangle</td>
<td>SurfaceArea_Shape_Rectangle</td>
</tr>
</tbody>
</table>

Figure Answers.48 Method tables for Rectangle and Square.

6.13 At run time, a class may be represented by a class descriptor which contains, among others, the method table of the class. Such a class descriptor could also contain a pointer to the class descriptor of its parent class. An object then contains a reference to its class descriptor instead of a reference to the method table. Then, the implementation of the \texttt{instanceof} operator becomes easy, see Figure Answers.49.

6.14 The code for these calls is:
\[
\begin{align*}
&\text{*(e->dispatch_table[0])}(e); \\
&\text{*(e->dispatch_table[2])}((\text{class D } *)(\text{char } *)e + \text{sizeof(class C)})); \\
&\text{*(e->dispatch_table[3])}((\text{class D } *)(\text{char } *)e + \text{sizeof(class C)}));
\end{align*}
\]

Note that although \( m4 \) is redefined in class \( E \), it still requires a pointer to an object of class \( D \).

6.15 The code for method \( m5 \) is:
FUNCTION Instance of (Obj, Class) RETURNING a boolean;
    SET Object Class TO Obj. Class;
    WHILE Object Class /= No class:
        IF Object Class = Class:
            RETURN true;
        ELSE Object Class /= Class:
            SET Object Class TO Object Class .Parent;
        RETURN false;

Figure Answers.49 Implementation of the instanceof operator.

void m5_E_E(Class_E *this) {
    *(int *)(char *)this + this->index_table[5]) =
    *(int *)(char *)this + this->index_table[4]) +
    *(int *)(char *)this + this->index_table[1]);
}

6.16 For example, when the caller calls several routines consecutively, the ‘caller saves’ scheme allows saving
and restoring only once, whereas the ‘callee saves’ scheme has no option but to do it for every call. Also, in
the ‘callee saves’ scheme the callee has to save and restore all registers that might be needed by any caller,
whereas the ‘caller saves’ scheme allows for saving and restoring only those registers that are needed for
this particular caller, at this particular call site.

6.18 See the code in Figure Answers.50.

void do_elements(int n, int element()) {
    int elem = read_integer();
    int new_element(int i) {
        return (i == n ? elem : element(i));
    }
    if (elem == 0) {
        printf("median = %d0, element((n-1)/2));
    } else {
        do_elements(n + 1, new_element);
    }
}

void print_median(void) {
    int error(int i) {
        printf("There is no element number %d0, %d, i); abort();
    }
    do_elements(0, error);
}

Figure Answers.50 Code for implementing an array without using an array.

6.19 It cannot, since the same closure may be fully curried and invoked several times, and different activation
records have to result.
6.23 The binary tree must be balanced, or the generated code could be as inefficient as the linear search scheme described earlier. If the tree has the smallest case label on top, all other case labels will be in the right branch, and the generated code is as inefficient as linear search in a sorted list.

6.24 We consider the following case statement:

```
CASE case expression IN
  I₁; statement sequence₁,
  ...,
  Iₙ; statement sequenceₙ,
ELSE else-statement sequence
END CASE;
```

The hash table entries consist of \((I_k, label_k)\) pairs, and the hash table keys are the \(I_k\) values. We then generate the code of Figure Answers.51.

```
tmp_hash_entry := get_hash_entry(case expression);
IF tmp_hash_entry = NO_ENTRY THEN GOTO label_else;
GOTO tmp_hash.entry.label;
label₁:
code for statement sequence₁,
GOTO label_next;
...
labelₙ:
code for statement sequenceₙ,
GOTO label_next;
label Else:
code for else-statement sequence
label_next:
```

Figure Answers.51 Intermediate code for a hash-table implementation of case statements.

6.25 A possible translation of the repeat statement to intermediate code is:

```
repeat_label:
code for statement sequence
code to evaluate Boolean expression into condition register
IF NOT condition register THEN GOTO repeat_label;
```

6.26 For the two while statement schemes, the `continue_label` should be placed at the `test_label`; the `break_label` should be placed at the end (where `end_label` is in the first scheme). In the general for-statement scheme of Figure 6.40, the `break_label` should be placed at the `end_label`; the `continue_label` should be placed right in front of the decrement of `tmp_loop_count`.

6.27 A continue statement transfers control to `expr3` in the body of the for-statement, but to `expr2` in the body of the while-statement, as shown in Figure Answers.52. A continue statement inside the body of a for-loop would have the effect of a jump to `forloop.continue_label`: `expr3` would still be evaluated before continuing with the next iteration; in a while loop, the effect is a jump to `whileloop.continue_label`.

6.28 This may cause overflow, because the controlled variable may then be incremented beyond the upper bound (and thus possibly beyond the maximum representable value).

6.29 See Figure Answers.53.

6.30 Pushing the last parameter first makes sure that the first parameter ends up on the top of the stack, regardless of the number of parameters. Unstacking the parameters will therefore yield them in textual order.
expr1;
while (expr2) {
    body;
    forloop_continue_label:
        expr3;
    /* whileloop_continue_label: */
}
When implementing instantiation through dope vectors, a generic routine has a code address, and can be passed as a parameter, just like an ordinary routine.

**Answers for Chapter 7**

### 7.2
The type of *foldr* is:

\[
foldr :: (a \to b \to b) \to b \to [a] \to b
\]

### 7.3
The definition of the dot operator is

\[
dot :: (a \to b) \to (c \to a) \to c \to b
\]
\[
dot f \cdot g \cdot x = f \cdot (g \cdot x)
\]

### 7.4
The functional-core equivalent of "list" is \((\text{Cons '}1\text{'} \ (\text{Cons '}i\text{'} \ (\text{Cons '}s\text{'} \ (\text{Cons '}t\text{'} \ []))))\), that of \([1..481]\) is \text{range 1 481}.

### 7.5
See Figure Answers.54.

---

\[
\begin{align*}
\text{unique a1} &= \text{if } (_\text{type_constr a1} == \text{Cons} \&\& _\text{type_constr} (_\text{type_field 2 a1}) == \text{Cons}) \text{ then} \\
& \quad \text{let} \\
& \quad \quad a = _\text{type_field 1 a1} \\
& \quad \quad b = _\text{type_field 1} (_\text{type_field 2 a1}) \\
& \quad \quad cs = _\text{type_field 2} (_\text{type_field 2 a1}) \\
& \quad \quad \text{in} \\
& \quad \quad \text{if} (a == b) \text{ then } a : \text{unique cs} \\
& \quad \quad \text{else } a : \text{unique (b:cs)} \\
& \quad \text{else a1}
\end{align*}
\]

*Figure Answers.54* A functional-core equivalent of the function *unique*.

---

### 7.6
(a) One should not write functions that fail for some arguments (but sometimes one has to). (b) One should not call a function with arguments for which it will fail (but that is sometimes hard to know).

### 7.8
See Figure Answers.55.

---

\[
\begin{align*}
\text{qsort \ [\ ]} &= \ [\ ] \\
\text{qsort \ (x:xs)} &= \text{qsort \ (mappend \ qleft \ xs) ++ [x]} \\
& \quad ++ \text{qsort \ (mappend \ qright \ xs)} \\
\text{where} \\
& \quad \text{qleft y} = \text{if} \ \{y < x\} \ \text{then} \ [y] \ \text{else} \ [] \\
& \quad \text{qright y} = \text{if} \ \{y \geq x\} \ \text{then} \ [y] \ \text{else} \ []
\end{align*}
\]

*Figure Answers.55* A functional-core equivalent of the function *qsort*.

---

### 7.9
See Figure Answers.56.

### 7.10
Any recursive function will fail to terminate when its if-then-else expression is replaced by an applicative-order function that implements the conditional, since applicative-order reduction will evaluate both the then-
and else-expressions before calling the conditional function:

\[
\text{cond } b \ t \ e = \text{if } b \text{ then } t \text{ else } e
\]

\[
fac \ n = \text{cond } (n==0) \ 1 \ (n \times fac \ (n-1))
\]

7.11 The following program uses the value of the expression \(fac \ 20\) \(2^{10}\) times:

\[
\begin{align*}
\text{tree } 0 \ \text{val} &= \text{val} \\
\text{tree } n \ \text{val} &= \text{let } t = \text{tree } (n-1) \ \text{val} \ \text{in} \ t\times t \\
\text{main} &= \text{print } \left( \text{tree } 10 \ (\text{fac } 20) \right)
\end{align*}
\]

7.12 \text{Eval()} must be extended with an additional case to handle the indirection nodes:

\[
\begin{align*}
\text{case IND:} \\
\quad \text{node} &= \text{node}\to nd.ind; \\
\quad \text{break;}
\end{align*}
\]

Also, the update must be changed into:

\[
\begin{align*}
\text{root}\to tag &= \text{IND}; \quad /* \text{update */} \\
\text{root}\to ind &= \text{node;}
\end{align*}
\]

7.13 C does not allow forward references to local variables.

\[
\begin{align*}
\text{Pnode rep}(\text{Pnode } \_\text{arg}) \{
\quad \text{Pnode } n &= \_\text{arg}[0]; \\
\quad \text{Pnode lst} &= \text{Cons}(n, \text{NULL}); \\
\quad \text{lst}\to cons.tl &= \text{lst;}
\quad \text{return} \ \text{lst;}
\}
\end{align*}
\]

7.15 The function \(f\) is strict in \(p\) only.

7.16 Substitute \(\text{equal}_\text{evaluated}, \text{mul}_\text{evaluated}, \text{and sub}_\text{evaluated}\) by their operators, which is allowed due to strictness. This is the partial evaluation component: actions that will certainly be performed at run time are done at compile time. Result:

\[
\begin{align*}
\text{Pnode fac}_\text{evaluated}(\text{Pnode } \_\text{arg0}) \{
\quad \text{Pnode } n &= \_\text{arg0}; \\
\quad \text{return} \ \text{Num}(n\to nd.num == \text{Num}(0)\to nd.num)\to nd.num ? \text{Num}(1) \\
\quad &\quad : \text{Num}(n\to nd.num \ast \\
\quad &\quad \quad \text{fac}_\text{evaluated}(\text{Num}(n\to nd.num - \text{Num}(1)\to nd.num))\to nd.num); \\
\}
\end{align*}
\]

Replace \(B \ ? \ F(x) : F(y)\) by \(F(B \ ? x : y)\); replace \(\text{Num}(x) \to nd.num\) by \(x\); result:
Pnode fac_evaluated(Pnode _arg0) {
    Pnode n = _arg0;
    return Num(
        n->nd.num == 0 ? 1 :
        n->nd.num * fac_evaluated(Num(n->nd.num - 1))->nd.num
    );
}

Introduce int n_num == n->nd.num; in fac_evaluated() and substitute; introduce int
fac_unboxed(int n) {return fac_evaluated (Num(n))->nd.num;}; replace fac_evaluated (Num(n))->nd.num by fac_unboxed(n) in fac_evaluated(); in-line the
call of fac_evaluated in fac_unboxed(); result:

int fac_unboxed(int n_arg) {
    Pnode n = Num(n_arg);
    int n_num = n->nd.num;
    return (Num(n_num == 0 ? 1 : n_num * fac_unboxed(n_num - 1))
    )->nd.num;
}

Clean up the in-lining debris; replace more Num(x) ->nd.num by x; result:

int fac_unboxed(int n_arg) {
    return n_arg == 0 ? 1 : n_arg * fac_unboxed(n_arg - 1);
}

7.17 We need zero or more temporary locations Temp[ ] at run time. Store the assignment set A in the variable
Assignment set. First remove from Assignment set all assignments of the form P_new[i] := P_old[i], for any i. Then use the algorithm from Figure Answers.57.

WHILE Assignment set /= Empty assignment set:
// Get a doable assignment:
    IF there is an Index SUCH THAT
        (P_new[Index] := α) is in the Assignment set AND
        P_old[Index] occurs at most in α:
    // P_new[Index] := α is doable:
        SET Next TO Index;
    ELSE there is no such Index:
        // Make room for a doable assignment:
        CHOOSE a (P_new[Index] := α) FROM the Assignment set;
        CHOOSE Scratch SUCH THAT
        Temp[Scratch] does not occur in the Assignment set;
        Generate code to move P_old[Next] to Temp[Scratch];
        Substitute Temp[Scratch] for P_old[Next]
        in the Assignment set;
        // Now P_new[Index] := α is doable:
        SET Next TO Index;
        Generate code for (P_new[Next] := α);
        Remove (P_new[Next] := α) from the Assignment set;

Figure Answers.57 Outline code for doing in situ assignments.

Extended exercise: The choices of the assignment from Assignment set and of the temporary variable
influence the amount of code and the number of temporary variables needed. Think of reasonable choice
criteria.
Although the ++ (append) operator is associative, the amount of work differs for a ++ (b ++ c) and (a ++ b) ++ c. This is a consequence of the append operator, which essentially creates a copy of its first argument to 'replace' the null pointer with a pointer to the second list. In (a ++ b) ++ c the list a is copied twice, whereas only one copy is required for a ++ (b ++ c). Transforming mappend into an accumulating argument function would cause the sublists to be copied many times (depending on their position).

There is no reduction in heap space since all lazy expressions allocated in the heap consist of a (built-in) function with two arguments, which is the break-even point. The benefit is in the graph reducer, which does not need to unwind application spines, but can call the suspended function immediately.

### Answers for Chapter 8

8.1 For \(-\) \text{grandparent}(\text{arne}, Z) left-to-right is more efficient. It first binds Z to \text{james} and tries \text{parent}((\text{james}, Z), which fails. Next, it binds Z to \text{sachiko}, and tries \text{parent}((\text{sachiko}, Z), which succeeds with Z=\text{rivka}. With a right-to-left search, the system will first try to solve \text{parent}(Y,Z), where both Y and Z are unbound. So, the program will try all possible combinations of Y and Z for which \text{parent}(Y,Z) holds, until it finds a combination for which \text{parent}((\text{arne}, Y) holds. With the program of Section 8.1, the fourth combination (\text{sachiko}, \text{rivka}) succeeds. For \(-\) \text{grandparent}(X, \text{rivka}), the situation is reversed, as now the first argument is unbound. So, there is no single optimal search order.

8.2 The initial stack is again
\[
gp(\text{arne},X),\lllll("X",X)
\]
Applying 'Attach clauses' with optimization results in:
\[
[\gp(\text{arne},X) \rightleftharpoons \gp(X1,Z1)],pa(X1,Y1),pa(Y1,Z1),\lllll("X",X)
\]
Unification results in:
\[
[\gp(\text{arne},X) \rightleftharpoons \gp(\text{arne},X)],pa(\text{arne},Y1),pa(Y1,X),\lllll("X",X)
\]
Then, 'Match' removes the two unified goals:
\[
pa(\text{arne},Y1),pa(Y1,X),\lllll("X",X)
\]
Applying 'Attach clauses' with optimizations results in:
\[
[pa(\text{arne},Y1) \rightleftharpoons pa(\text{arne},\text{james})],pa(Y1,X),\lllll("X",X)
\]
\[
[pa(\text{arne},Y1) \rightleftharpoons pa(\text{arne},\text{sachiko})],pa(Y1,X),\lllll("X",X)
\]
Unification on the top entry gives:
\[
[pa(\text{arne},\text{james}) \rightleftharpoons pa(\text{arne},\text{james})],pa(\text{james},X),\lllll("X",X)
\]
\[
[pa(\text{arne},Y1) \rightleftharpoons pa(\text{arne},\text{sachiko})],pa(Y1,X),\lllll("X",X)
\]
'Match' then removes the unified goals:
\[
\text{pa}(\text{james},X),\lllll("X",X)
\]
\[
[pa(\text{arne},Y1) \rightleftharpoons pa(\text{arne},\text{sachiko})],pa(Y1,X),\lllll("X",X)
\]
'Attach clauses' on the top entry leaves no entries, so we are left with
\[
[pa(\text{arne},Y1) \rightleftharpoons pa(\text{arne},\text{sachiko})],pa(Y1,X),\lllll("X",X)
\]
which after unification and matching results in
\[
\text{pa}(\text{sachiko},X),\lllll("X",X)
\]
Again an 'Attach clauses' is required (with optimizations):
\[
[pa(\text{sachiko},X) \rightleftharpoons pa(\text{sachiko},\text{rivka})],\lllll("X",X)
\]
Unification succeeds with X=rivka:
\[ \text{pa}(\text{sachiko}, \text{rivka}) == \text{pa}(\text{sachiko}, \text{rivka}) \Rightarrow (\text{"X", rivka}) \]

‘Match’ removes the unified goals, leaving us with a display goal:
\[ \Rightarrow (\text{"X", rivka}) \]

An effective optimization indeed.

8.4 The new paradigm implements breadth-first search rather than depth-first search. This usually takes (much) more time, but will find a solution if one exists, unlike depth-first search which may work itself in an infinite branch of the search tree. The discussed optimizations are all applicable to some degree.

8.5 The only interpreter instruction that may be affected is ‘Unify’. Perhaps surprisingly, during unification a relation name can be handled just like a constant.

8.7 See Figure Answers.58. The code is an improvement over that of Figure 8.39 since it scans the list iteratively whereas naive application of unify_structures() would scan the list recursively.

```c
int unify_lists(struct list *l_goal, struct list *l_head) {
    int counter;
    if (l_goal->arity != l_head->arity) return 0;
    for (counter = 0; counter < l_head->arity; counter++) {
        if (!unify_terms(
                l_goal->components[counter], l_head->components[counter])
        )
            return 0;
    }
    return 1;
}
```

Figure Answers.58 C code for the unification of two lists.

8.8 E to C, C to B, D to B, B to A.

8.9 The number of values returned in general is unbound, so an infinite list may have to be returned.

8.10 See Figure Answers.59

8.11 Replace the code segment

```c
/* translation of ‘parent(Y,Z).’ */
void first_gp_2_clause_1_goal_3(void) {
    parent_2(Y, goal_arg2, goal_list_tail);
}
```

in Figure 8.24 by

```c
/* translation of ‘parent(Y,Z).’ */
void first_gp_2_clause_1_goal_3(void) {
    parent_2(Y, goal_arg2, goal_list_tail);
}
```
void intersect(
    void list1(void (*Action)(int v)),
    void list2(void (*Action)(int v)),
    void (*Action)(int v)
) {
    void in_list2(int v) {
        void equal_to_v(int v2) {
            if (v == v2) Action(v);
        }
        list2(equal_to_v);
    }
    list1(in_list2);
}

Figure Answers.59 A list procedure that implements integer list intersection.

/* translation of 'parent(Y,Z),!.' */
void first_gp_2_clause_1 Goal_3(void) {
    /* translation of '!' */
    void first_gp_2_clause_1_goal_4(void) {
        goal_list_tail();
        goto L_cut;
    }
    /* translation of 'parent(Y,Z),.' */
    parent_2(Y, goal_arg2, first_gp_2_clause_1_goal_4);
}

8.12 The asserts and retracts should not be undone by backtracking. Since the data structure in which the asserted clauses are kept and the corresponding counters (for example, number_of_clauses_added_at_end_for_parent_2 in Figure 8.25) are global variables, no backtracking will occur, as required.

8.14 (a) Translate ..., var(X), α. to
    built_in_var(X, routine_for_α);
with built_in_var() defined as in Figure Answers.60.

void built_in_var(Term *t, Action goal_list_tail) {
    Term *arg = deref(t);
    if (arg->type == Is_Variable) {
        goal_list_tail();
    }
}

Figure Answers.60 A C routine for implementing built_in_var().

(b) See Figure Answers.61

8.15 A first implementation is shown in Figure Answers.62 where Term *put_integer(i) creates a constant term with the value i, but the code needs optimization.
void built_in_is(Term *t1, Term *t2, Action goal_list_tail) {
    Term *arg = deref(t1);
    Term *expr = evaluate_expression(t2);
    if (arg->type == Is_Variable) {
        arg->term.variable.term = expr;
        goal_list_tail();
        arg->term.variable.term = 0;
    } else
    if (arg->type == Is_Constant
        && atoi(arg->term.constant) == atoi(expr->term.constant)) {
        goal_list_tail();
    }
}

Figure Answers.61 A C routine for implementing built_in_is().

void built_in_between(
    Term *t1, Term *t2, Term *t3,
    Action goal_list_tail
) {
    Term *arg = deref(t1);
    Term *expr1 = evaluate_expression(t2);
    Term *expr2 = evaluate_expression(t3);
    if (arg->type == Is_Variable) {
        int from = atoi(expr1->term.constant);
        int to = atoi(expr2->term.constant);
        int i;
        for (i = from; i < to; i++) {
            arg->term.variable.term = put_integer(i);
            goal_list_tail();
            arg->term.variable.term = 0;
        }
    } else
    if (arg->type == Is_Constant
        && atoi(expr1->term.constant) <= atoi(arg->term.constant)
        && atoi(arg->term.constant) <= atoi(expr2->term.constant)) {
        goal_list_tail();
    }
}

Figure Answers.62 A first implementation of a compiled between.

Answers for Chapter 9

9.1 No; shared variables are easier to use than message passing. Also, on a shared-memory machine, shared-
variable programs often get better performance than message passing programs, because message passing programs do more copying.

9.2 To send a message \( \text{msg} \) to processor \( P \), do:

\[
\text{out}(\text{"message"}, \ "P", \ \text{msg});
\]

If processor \( P \) wants to receive a message, it executes:

\[
\text{in}(\text{"message"}, \ "P", \ ? \ &\text{msg});
\]

To create a shared variable \( X \) with initial value \( V \) execute:

\[
\text{out}(\text{"shared variable"}, \ "X", \ V);
\]

To read the value of the shared variable into a local variable, execute:

\[
\text{read}(\text{"shared variable"}, \ "X", \ ? \ &\text{var});
\]

To write a new value into the variable, execute:

\[
\text{in}(\text{"shared variable"}, \ "X", \ ? \ &\text{var}); \ /* \ discard \ old \ value */
\]

\[
\text{out}(\text{"shared variable"}, \ "X", \ newvalue);
\]

9.3 Figure 9.4 suggests that most of the thread state is independent of the CPU architecture, except maybe for pointer sizes. However, the register information depends strongly on the CPU type. Some CPUs, for example the Intel architectures, have few registers to save, while others, for example the Sparc, have many registers.

9.4 No; the lock that is used to protect the administration can use spinning, because the operations on the list are very simple and will not block for a long time.

9.5 First try to acquire the lock several times using busy waiting; if it still is taken after a certain number of tries, then do a thread switch. This will work efficiently for fine-grained locks, which are acquired for only a very short time.

9.6 (a) The first operation that succeeds in acquiring the monitor lock will continue. The second operation, however, will block on the monitor lock, and thus cannot continue until the first operation has released the lock. Thus, although the two operations could in principle be executed simultaneously, the implementation of a monitor runs them sequentially.

(b) 1. Use symbolic interpretation to find those operations that are read-only. 2. Use a multi-state lock to protect the monitor: nobody inside, some readers inside, some readers inside and one or more writers waiting, and one writer inside.

9.7 Without this restriction, a message sent to a port can be accepted by many different processors, so the run-time system would have to communicate with different processors to check which one currently has a process that is willing to receive a message from the port. With the restriction, the run-time system merely needs to maintain a mapping between ports and processors; a message sent to a port is sent to the processor currently listening to the port.

9.9 The first case is fairly easy to implement: the run-time system just checks among the messages that have been sent to this process whether there is one that matches the criteria. This can be implemented without doing any extra communication. In the second case, the run-time system would have to communicate with other processors, to see if they have a process that is ready to receive.

9.10 It is difficult to copy the stack of a thread, because there may exist variables that point into the stack. In C, for example, it is possible to take the address of a local variable (stored on the stack) and store this value in a global variable. If the stack is copied to a new region in memory, the global variable will still point to the location in the original stack. So, migrating a stack from one memory region to another is hard to implement transparently.

9.11 Since \( X \) contains a thread, it is harder to migrate, since threads always have a state that has to be migrated (for example, a stack); moreover, thread migration has the same implementation difficulty as stack migration, as described in the answer to Exercise 9.10. Migrating object \( Y \) is easier and less expensive, since it
does not contain a thread. On the other hand, if multiple active objects (like X) located on different processors do operations on object Y, object Y will repeatedly be moving between these processors. In this case, it may be more efficient to move the active objects to the processor of Y.

9.12 Suppose two different processes do a write operation on two different objects, X and Y, with different primary copies. Each primary copy broadcasts the operation to other processors containing a copy. The two broadcast messages, however, are not ordered relative to each other, so some processors may get the update of X first while others get the update of Y first, resulting in inconsistency. With totally ordered broadcast, all broadcasts are ordered, so either the update of X arrives first at all processors, or the update of Y arrives first at all processors.

9.13 (a) Hash-based: almost all CPUs will do 3 point-to-point communications to the same CPU, the one containing the max tuple. Broadcast outs: all CPUs will do 2 broadcasts (in and out) and one local operation (read). Do outs locally: almost all CPUs will do 2 broadcasts (for in and read).
(b) Optimized operations use some form of spanning tree to collect the result; every CPU sends a point-to-point message to its parent in the tree, the parent takes the maximum and sends it to its parent, and so on, until the result reaches the root of the tree. The root then does a broadcast. This is more efficient than the Linda version.

9.14 The matching tuples may be on different processors; other Tuple Space operations should be prevented from executing while the fetch_all() is being executed.

9.15 No, all statements assign to different elements of A. There would have been a dependency between the first two assignments if the upper bound of the loop had been 10.

9.16 The cache performance of the transformed loops will be much worse than that of the original code, assuming that arrays are stored in row-order in memory. The reason is that cache entries are always fetched in blocks of contiguous memory words, called cache lines. If the cache is too small to contain the entire array, the transformed code will repeatedly experience cache misses. On every miss, it will fetch a cache line, but only use one word of it. The original code will use the entire cache line, and thus get fewer cache misses, because it accesses contiguous words in memory.

Answers for Appendix A

A.1 (a) In Java, C++, and probably in most other object-oriented languages, the ‘constructor’ is not a constructor at all but rather an initializer. The real constructor is the new, and when it has finished constructing the object (building the house), the constructor is called to initialize it (move in the furniture). In Expression Program = new Expression();, the new allocates space for an Expression, which will not be large enough to hold any of its extended subtypes, and the constructor cannot stretch this space afterwards. Also, it is the new that returns the object (or a pointer to it), not the constructor.
(b) Its constructor should be a real constructor, in which the programmer can allocate the space, possibly for a subtype, initialize its fields, and return the constructed object.
References


Austin, T. M., Breach, S. E., and Sohi, G. S. (June 1994). Efficient detection of all pointer and array access errors. ACM SIGPLAN Notices, 29(6), 290-312


Farrow, R. (June 1984). Sub-protocol-evaluators for attribute grammars. ACM SIGPLAN Notices, 19(6), 70-80
Goldberg, B. (June 1991). Tag-free garbage collection for strongly typed programming languages. ACM SIGPLAN Notices, 26(6), 165-176


Siklóssy, L. (June 1972). Fast and read-only algorithms for traversing trees without an auxiliary stack. *Information Processing Letters*, 1(4), 149-152


Index

2-pass compiler 26
2-scan compiler 26

a posteriori type 455
a priori type 455
ABC 46
abstract class 473, 701, 703
abstract data type 471–472
abstract syntax tree 9, 22, 52, 55, 194
acceptable partitioning 222, 223–224
acceptable-set method 138–139
accumulating arguments 585, 595
acquiring a lock 660, 669–670, 674, 695, 750
action routine 616
ACTION table 160, 170, 191
activation record 32, 412, 462, 482
active node 285–286
active routine 32, 485, 497, 714
active-node pointer 285–287, 297
actual parameter in Linda 664
Ada 447, 696
address descriptor 319
address space 29, 376, 657, 659, 662, 668, 675
administration part 483–484, 514, 522
aggregate node allocation 588
Algol 68 457, 463–464, 488, 496, 622
Algol 68 format 187
alignment requirements 379, 399, 435, 465, 467, 533
alternative 37, 113–114, 702
ambiguous grammar 38, 111, 172, 192, 457, 710
amortized cost 397, 560
analysis–synthesis paradigm 6
ancestor routine 485, 490
AND/OR form 701, 708
annotated abstract syntax tree 9–10, 54, 279
annotation 9, 96, 194
anonymous type declaration 449
anti-dependence 687
ANTLR 132
application spine 561, 573, 746
applicative-order reduction 566, 594, 744
applied occurrence 199, 380, 440, 441
arithmetic sequence construct 541
arithmetic simplification 367, 370, 586
arity 546, 566, 587, 610, 639
array descriptor 469
array type 450, 467
array without array 534, 740
assembly code 374, 379–380, 583, 667
asserta() 630
assertz() 630–631
assignment 248, 277, 321, 416, 458, 464, 532
assignment under a pointer 334, 688
associative addressing 664
AST 9
asynchronous message passing 662–663, 692
atomic action 664
Attach clauses instruction 603, 746
attribute 9, 96, 135, 234, 547
attribute evaluation rule 196–197, 211, 223, 227, 274
attribute evaluator 196, 202, 204, 231, 273
attribute grammar 7, 195, 547
automatic parallelization 684, 695
automatic programming 33
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>auxiliary register</td>
<td>309</td>
</tr>
<tr>
<td>available ladder sequence</td>
<td>328</td>
</tr>
<tr>
<td>back-end</td>
<td>3, 6, 11, 32</td>
</tr>
<tr>
<td>backpack list</td>
<td>263, 380</td>
</tr>
<tr>
<td>backpatching</td>
<td>263, 380</td>
</tr>
<tr>
<td>backtracking</td>
<td>77, 117, 596, 607, 654</td>
</tr>
<tr>
<td>Backus Normal Form</td>
<td>37</td>
</tr>
<tr>
<td>Backus–Naur Form</td>
<td>37</td>
</tr>
<tr>
<td>backward data-flow</td>
<td>262, 277</td>
</tr>
<tr>
<td>bad pointer</td>
<td>462</td>
</tr>
<tr>
<td>bag</td>
<td>399</td>
</tr>
<tr>
<td>base address</td>
<td>467−468</td>
</tr>
<tr>
<td>basic block</td>
<td>254, 320, 322, 373</td>
</tr>
<tr>
<td>basic item</td>
<td>72−73, 80</td>
</tr>
<tr>
<td>basic operand</td>
<td>351</td>
</tr>
<tr>
<td>basic type</td>
<td>450, 460, 543, 582</td>
</tr>
<tr>
<td>BC(k,m) parsing</td>
<td>152</td>
</tr>
<tr>
<td>binary relation</td>
<td>598</td>
</tr>
<tr>
<td>binding a logical variable</td>
<td>596, 654</td>
</tr>
<tr>
<td>bison</td>
<td>178</td>
</tr>
<tr>
<td>bitset</td>
<td>470</td>
</tr>
<tr>
<td>block</td>
<td>397, 399</td>
</tr>
<tr>
<td>blocked thread</td>
<td>668−670, 695−696</td>
</tr>
<tr>
<td>BNF</td>
<td>37, 59, 158, 202</td>
</tr>
<tr>
<td>body</td>
<td>599</td>
</tr>
<tr>
<td>Boolean control expression</td>
<td>502</td>
</tr>
<tr>
<td>Boolean type</td>
<td>461</td>
</tr>
<tr>
<td>bootstrapping</td>
<td>2, 709</td>
</tr>
<tr>
<td>bottom-up parser</td>
<td>114, 150, 178, 191, 230, 234−235, 716</td>
</tr>
<tr>
<td>bottom-up pattern matching</td>
<td>341, 342−343, 346</td>
</tr>
<tr>
<td>bottom-up rewriting system</td>
<td>341</td>
</tr>
<tr>
<td>bottom-up tree rewriting</td>
<td>320, 357, 365, 372</td>
</tr>
<tr>
<td>bounded stack</td>
<td>436</td>
</tr>
<tr>
<td>boxing analysis</td>
<td>582</td>
</tr>
<tr>
<td>broad compiler</td>
<td>26−27, 96, 244, 320</td>
</tr>
<tr>
<td>bucket</td>
<td>100, 102</td>
</tr>
<tr>
<td>BURS</td>
<td>337, 341, 349, 357, 494</td>
</tr>
<tr>
<td>busy waiting</td>
<td>669, 695</td>
</tr>
<tr>
<td>C</td>
<td>25, 117, 375, 442, 549, 558, 597</td>
</tr>
<tr>
<td>C preprocessor</td>
<td>107</td>
</tr>
<tr>
<td>caching behavior</td>
<td>688</td>
</tr>
<tr>
<td>call by reference</td>
<td>514</td>
</tr>
<tr>
<td>call by result</td>
<td>514</td>
</tr>
<tr>
<td>call by value</td>
<td>514</td>
</tr>
<tr>
<td>call by value-result</td>
<td>514</td>
</tr>
<tr>
<td>callee</td>
<td>29, 260, 513, 534, 576, 583</td>
</tr>
<tr>
<td>callee-saves scheme</td>
<td>515, 534</td>
</tr>
<tr>
<td>caller</td>
<td>260, 518, 576, 583</td>
</tr>
<tr>
<td>caller-saves scheme</td>
<td>515, 534</td>
</tr>
<tr>
<td>calling graph</td>
<td>41, 43</td>
</tr>
<tr>
<td>calling sequence</td>
<td>513, 515, 517, 634</td>
</tr>
<tr>
<td>candidate set</td>
<td>261</td>
</tr>
<tr>
<td>case label</td>
<td>506, 536</td>
</tr>
<tr>
<td>case statement</td>
<td>505–507, 536</td>
</tr>
<tr>
<td>cast</td>
<td>457</td>
</tr>
<tr>
<td>cell</td>
<td>411, 420</td>
</tr>
<tr>
<td>character move</td>
<td>73</td>
</tr>
<tr>
<td>child</td>
<td>112</td>
</tr>
<tr>
<td>child routine</td>
<td>485, 495</td>
</tr>
<tr>
<td>chunk</td>
<td>397, 399</td>
</tr>
<tr>
<td>class</td>
<td>471</td>
</tr>
<tr>
<td>class composition</td>
<td>700</td>
</tr>
<tr>
<td>class extension</td>
<td>700, 708</td>
</tr>
<tr>
<td>clause</td>
<td>596, 600</td>
</tr>
<tr>
<td>cloning</td>
<td>368, 370−371</td>
</tr>
<tr>
<td>closed hashing with chaining</td>
<td>100</td>
</tr>
<tr>
<td>closure</td>
<td>439, 498, 589</td>
</tr>
<tr>
<td>closure algorithm</td>
<td>42, 498</td>
</tr>
<tr>
<td>closure specification</td>
<td>43, 51</td>
</tr>
<tr>
<td>CM-5</td>
<td>657</td>
</tr>
<tr>
<td>coalesce</td>
<td>400, 403, 408, 737</td>
</tr>
<tr>
<td>code address of a routine</td>
<td>483, 485, 491, 496, 499, 587</td>
</tr>
<tr>
<td>code generated for a query</td>
<td>621</td>
</tr>
<tr>
<td>code generation</td>
<td>19, 290, 295, 297, 302, 320, 337, 374, 501, 523, 525, 568</td>
</tr>
<tr>
<td>code generation module</td>
<td>24</td>
</tr>
<tr>
<td>code hoisting</td>
<td>277</td>
</tr>
<tr>
<td>code in-lining</td>
<td>24, 369−370, 395, 582, 623, 638, 745</td>
</tr>
<tr>
<td>code segment</td>
<td>375, 396, 736</td>
</tr>
<tr>
<td>code selection</td>
<td>293</td>
</tr>
<tr>
<td>code-generating scan</td>
<td>341</td>
</tr>
<tr>
<td>coercion</td>
<td>448, 455</td>
</tr>
<tr>
<td>collision</td>
<td>100, 103, 444</td>
</tr>
<tr>
<td>column-major order</td>
<td>468</td>
</tr>
<tr>
<td>comb algorithm</td>
<td>90</td>
</tr>
<tr>
<td>comment starter</td>
<td>68, 187</td>
</tr>
<tr>
<td>common subexpression</td>
<td>324–325, 334–335, 391–392</td>
</tr>
<tr>
<td>common subexpression removal</td>
<td>310, 324–326, 549</td>
</tr>
<tr>
<td>communication daemon thread</td>
<td>672, 674</td>
</tr>
<tr>
<td>compaction</td>
<td>408, 421, 428, 437, 736</td>
</tr>
<tr>
<td>comp.compilers newsgroup</td>
<td>185</td>
</tr>
<tr>
<td>compilation by symbolic interpretation</td>
<td>318</td>
</tr>
<tr>
<td>compilation unit</td>
<td>31, 357, 450, 523</td>
</tr>
<tr>
<td>compiler</td>
<td>1</td>
</tr>
<tr>
<td>compiler compiler</td>
<td>8, 178</td>
</tr>
<tr>
<td>composition operator</td>
<td>58, 186</td>
</tr>
<tr>
<td>compound result type</td>
<td>517</td>
</tr>
<tr>
<td>concurrent execution</td>
<td>663, 675, 685</td>
</tr>
<tr>
<td>concurrent garbage collection</td>
<td>409, 435</td>
</tr>
<tr>
<td>condition code</td>
<td>286</td>
</tr>
<tr>
<td>condition synchronization</td>
<td>660</td>
</tr>
<tr>
<td>condition variable</td>
<td>660, 670</td>
</tr>
</tbody>
</table>
Index

intermediate code generation module 23
intermediate code optimization module 24
intermediate code tree 279
interpreter 3, 281, 295, 560, 603, 630, 699
interprocedural data-flow analysis 260, 688
invalidate replicated copies 677
IS-dependency 212
IS-SI graph 212, 274
item 71–72, 110, 153, 373
iterative interpreter 285, 297, 389, 603
iterator 485, 500
Java RMI 673
join operator 255
jump buffer 490
jump table 506
jumping out of a routine 488, 496
kind 449
ladder sequence 328
LALR(1) automaton 172
LALR(1) parsing 152, 170, 176, 191, 234
lambda lifting 499, 559, 594
language generated by a grammar 40
last-def analysis 253
late evaluation 224, 228, 328
latency 671
lattice 250, 277
L-attributed grammar 230, 269, 275
lazy Boolean operator 117–118, 503
lazy evaluation 328, 547, 560, 564, 566, 575
lcc 389
least fixed point 43
left-associative 153, 540, 716
left-factoring 129
left-hand side 35
leftmost derivation 36
leftmost innermost redex 566
leftmost outermost redex 566
left-recursion removal 129
left-recursive 38, 120, 127, 144
Lempel-Ziv compression 86
let-expression 543, 572, 594
lex 83, 94, 187
lexical analysis module 22
lexical analyzer 10, 61, 68, 83, 93–94, 541
lexical identification phase 98
lexical pointer 483–484, 491, 493, 514, 559
library 25, 281, 297, 376, 658, 672
lifetime 463
Linda 664
Linda kernel 678
Linda preprocessor 678
Linda Tuple Space 678
linear-time 31, 86, 111, 168, 224, 277, 406, 726
linked list 398, 403
linker 25, 376, 381, 389
list 247
list 541
list comprehension 542, 557
list procedure 616, 654
live analysis 262–263, 265, 319, 333, 358
live variable 262
LL(1) conflict 127–128, 715
LL(1) grammar 123, 127, 142
LL(1) parser generation 123
LL(1) parsing 121–122, 139, 715
LL(2) grammar 132, 190
LLgen 132, 142, 179, 232, 237, 715
LLgen directive 144
loader 376, 389
local attribute 203
local variable area 515
location 458
lock variable 660, 669
longjmp() 490
look-ahead set 165–166, 190
loop interchange 689
loop restructuring transformation 688
loop unrolling 511
loop-carried dependence 687
loop-exit list 248
LR automaton 159
LR item 153, 191
LR reduce item 154
LR shift item 154
LR(0) grammar 162
LR(0) parsing 152–153, 191
LR(1) parsing 152, 165, 191, 217
lvalue 458, 462, 470, 514
machine code generation 374
machine code generation module 24
macro application 103
macro call 103
macro definition 103, 444
makefile 32, 296
mark and scan 408, 420
mark and sweep 420
marked bit 420
marker rule 131
marking 420
marking phase 420
marshaling 672
Match instruction 603, 609, 746
match move 134
matching a goal 599
matching in regular expressions 58
matrix 467
maximal basic block 320
meaning of ‘to compile’ 3
meet operator 255
member 35
memoization 285, 348, 350, 389, 436
memory dump 520
memory fragmentation 408, 419, 427
memory management unit 462
message aggregation 692
message buffer 672
message combining 692
message handler 662
message passing 658, 671
message selection 674
method 663
method invocation 471, 479, 512, 673
method overriding 472
Miranda 593
module 523
module initialization 524
monitor 660, 669, 695
monitor lock 670
most recent binding 607, 613
MPI 658
multicomputer 657
multiple inheritance 476, 533
multiprocessor 657
multi-visit attribute evaluation 219, 230
mutual exclusion synchronization 660
name equivalence 454
name generation for modules 524
name list 98, 442
name space 442, 526, 532, 675
narrow compiler 26, 96, 232, 269, 302, 320–321, 444
nested routine 485, 558, 597, 618
newline 57, 185
non-correcting error recovery 116
non-cyclic attribute grammar 217, 275
non-deterministic 162, 362
non-deterministic-polynomial 90
non-local assignment 494
non-local goto 482, 488, 496, 518
non-local label 488, 491, 496
non-terminal symbol 35, 39
no-op instruction 379
normal-order reduction 566
NP-complete 89, 90, 293, 320, 360, 373
N-phase compiler 27
NPRD grammar 189
null sequence 367
nullable non-terminal 38, 121–122, 190
object 471, 658
object constructor 471, 527, 702, 708
object destructor 471, 527
object identifier 675
object location 675
object migration 676
object replication 677
object-oriented compiler construction 699
object-oriented parsing 702
occur check 608
offset table 479
offside rule 438, 541
OID 675
one-shot garbage collection 408
one-space copying 437
on-the-fly garbage collection 408
operation on an object 663
optimization 32, 64, 102, 403, 479, 511, 646, 648
ordered attribute evaluator 224
ordered attribute grammar 224
output in Linda 664
outer loop 686
outline code 46
output dependence 687
overloading 447
package 523
panic-mode error correction 139
parallel loop statement 665
parallel system 656
parameter area 514
parameter passing mechanism 514
parent class 472, 476, 481
parent routine 485
parse table 164
parse tree 9, 148
parser 36, 110, 702
parsing 9, 110, 540, 616
partial evaluation 297, 300, 390, 638, 744
partial parameterization 488, 499
partition (Tuple Space) 679, 683
partitioning (of attributes) 222
passing a routine as a parameter 463, 487, 495, 617
pattern matching 543, 553, 593
parse tree 337, 732
PDA 134
peephole optimization 333, 371
phase of a compiler 27
pointer chasing 677
pointer consistency 410
pointer layout 410
pointer problems 334
pointer (re)subtyping 474
pointer reversal 422
pointer scope rules 463, 559, 738
pointer supertyping 474
pointer tagging 588
pointer type 461
pointer validity 410
polymorphic function application 551
polymorphic type 543
polymorphic type checking 551
polynomial time 90
pool of threads 674
port 662
portability 32
portability of a compiler 32
position-independent code 735
post-order visit 112
precedence 59, 174
precedence parsing 152
precomputation 66, 80, 120, 157, 191, 285, 346, 469, 475
prediction move 134
prediction rule 155
prediction stack 134
predictive parser 121
predictive recursive descent parser 15, 117
pre-order visit 112
prerequisite to 200, 210
privatization 689
process 658, 659
process abstraction 667
production rule 35, 39, 196, 547
production step 35, 40
production tree 35
program counter 351, 395, 482, 493, 662, 667
program data area 409, 496
program generator 7, 66, 301
program text input module 22, 56
program-counter relative addressing 395
Prolog 43, 296, 597, 628, 630, 654
property list 247
pseudo-register 329, 362
pure register machine 304
pure stack machine 303
push-down automaton 134, 159
PVM 658

qualifier 557
query 596, 599

reaching-definitions analysis 253
read operation in Linda 664
read/write mode 646
receive statement 662
reclamation 434
record type 465
recursive descent parser 15, 117
recursive interpreter 281
redex 561
reduce item 72, 186, 733
reduce move 160
reduce-reduce conflict 162
reducible expression 561
reduction engine 566
reduction (graph) 561
reduction (in parsing) 152
reduction stack 159
reference counting 407, 415
referential transparency 406, 544
refinement 482
register allocation 293, 316, 332–333, 357
register allocation by graph coloring 320, 357
register and variable descriptor 318
register descriptor 319
register interference graph 359
register spilling technique 314
register storage class 336
register tracking 320, 333
register-memory machine 327
regular description 56, 60
regular expression 34, 56, 58, 237, 342
regular grammar 34
regvar descriptor 318
relation 596, 598
releasing a lock 660
relocation 380
relocation information 377
repeated inheritance 478
repeated variables 593
repetition operator 58, 72, 124, 142
replacement pattern 371
replicated Tuple Space 683
rest of program, grammar for 116
resume statement 485
resuming a routine 483, 485
retargetability 32
return information 483
return list 247
returning a routine as a value 482, 487, 495, 559
right-associative 117, 192, 551, 593
right-hand side 35
right-recursion 38, 191
root set 409, 496
routine call 487, 512–513, 616
routine definition 487, 616
routine denotation 621
routine type 471
row displacement 87
row-major order 468
rule 596, 599
runnable thread 668
running routine 32, 439, 482, 485
running thread 668
run-time error 519
run-time system 25, 281, 396, 545, 560, 659, 667
rvalue 458, 462, 470, 514
safe pointer 464
S-attributed grammar 230, 235, 275–276
scalar expansion 689
scan phase 420
scan pointer 425, 736
scheduler 668, 686
scheduler activation 667
Schorr and Waite algorithm 422, 436
scope in block-structured languages 442
scope of a pointer 463, 491, 532, 559
screening 98
select statement 696
selection statement 505
self-descriptive chunk 411
self-organizing list 102
semantic checking 441
semantic representation 2
sentential form 35, 40
sentential form 35, 40
serialization 672
set type 470
Sethi–Ullman numbering 311
setjmp() 491
setjmp/longjmp mechanism 490
shadow memory 287
shared variable 658, 659, 668
shift item 72, 166, 172, 186
shift move 159
shift-reduce conflict 162
shift-shift conflict 191
short-circuiting a function application 573, 586
SI-dependency 212
signal handler 521, 537
signal operation 660, 670
signal statement 521
signature of a computer virus 83
simple symbolic interpretation 247, 277, 723
simulation on the stack 245
single inheritance 476
Single Program Multiple Data parallelism 685
SLR(1) parsing 152, 163, 190–191
Smalltalk 33
sorted set 399
source language 1
SP 513
SP-2 657
sparse transition table 86
specialization 370
spineless tagless G-machine 592
spinning 669
SPMD parallelism 685
SR 674
stack 513
stack, direction of growth 513
stack of input buffers 105
stack pointer 513
stack representation 245
stack segment 376, 396
start symbol 35, 39, 136, 144, 199
state 80
state transition 80
static array 470
static attribute evaluation 204, 218
static binding 474
static cycle checking 211
static link 484, 491, 514
status indicator 282
storage representation in Linda 680
strength reduction 367
strict argument 565, 574, 576, 579, 594
strictness analysis 576, 594
string 39
strong-LL(1) parser 124
strongly LL(1) 124
strongly non-cyclic 217, 275
structural equivalence 455
structure type 465
subclass 472
subset algorithm 81, 158, 346, 352
substitution in grammars 129
subtree 112
Succeed instruction 603, 605
suffix grammar 116, 188
supercompilation 363
suspend statement 485
synchronized routine 439, 482, 485
switch statement 505
symbol 39
symbol table 98, 133, 188, 284, 442
symbolic interpretation 245
synchronization primitive 660
synchronous message passing 662
syntactic sugar 540, 553
syntax analysis 9, 57, 110, 440
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax analysis module</td>
<td>22</td>
</tr>
<tr>
<td>Syntax tree</td>
<td>9, 110, 135, 230</td>
</tr>
<tr>
<td>Synthesized attribute</td>
<td>196</td>
</tr>
<tr>
<td>Table compression</td>
<td>86</td>
</tr>
<tr>
<td>Table lookup</td>
<td>68, 285, 347</td>
</tr>
<tr>
<td>Tail call</td>
<td>582</td>
</tr>
<tr>
<td>Target code optimization module</td>
<td>24</td>
</tr>
<tr>
<td>Target language</td>
<td>1, 375, 523, 549</td>
</tr>
<tr>
<td>Target register</td>
<td>309</td>
</tr>
<tr>
<td>Task</td>
<td>33, 656</td>
</tr>
<tr>
<td>Task parallelism</td>
<td>665</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>671</td>
</tr>
<tr>
<td>Term</td>
<td>596</td>
</tr>
<tr>
<td>Terminal</td>
<td>35</td>
</tr>
<tr>
<td>Terminal production</td>
<td>40</td>
</tr>
<tr>
<td>Terminal symbol</td>
<td>35, 39</td>
</tr>
<tr>
<td>Terminated routine</td>
<td>32, 462, 485</td>
</tr>
<tr>
<td>Test-and-set instruction</td>
<td>669</td>
</tr>
<tr>
<td>TeX</td>
<td>8</td>
</tr>
<tr>
<td>Thread</td>
<td>658, 662, 667</td>
</tr>
<tr>
<td>Thread control block</td>
<td>667</td>
</tr>
<tr>
<td>Thread preemption</td>
<td>668</td>
</tr>
<tr>
<td>Threaded code</td>
<td>297</td>
</tr>
<tr>
<td>Threading (of an AST)</td>
<td>239, 247, 276, 287</td>
</tr>
<tr>
<td>Throughput</td>
<td>671</td>
</tr>
<tr>
<td>Token</td>
<td>10, 35</td>
</tr>
<tr>
<td>Token description</td>
<td>61, 70, 77</td>
</tr>
<tr>
<td>Tokenize</td>
<td>10</td>
</tr>
<tr>
<td>Top-down parser</td>
<td>113, 120, 142, 232</td>
</tr>
<tr>
<td>To-space</td>
<td>408, 425</td>
</tr>
<tr>
<td>Totally-ordered group communication</td>
<td>678</td>
</tr>
<tr>
<td>Trail</td>
<td>610, 634</td>
</tr>
<tr>
<td>Transition diagram</td>
<td>82, 158, 161, 163</td>
</tr>
<tr>
<td>Transition function</td>
<td>80</td>
</tr>
<tr>
<td>Transitive closure</td>
<td>42, 498</td>
</tr>
<tr>
<td>Transitive closure algorithm</td>
<td>44</td>
</tr>
<tr>
<td>Translation</td>
<td>1</td>
</tr>
<tr>
<td>Traversing a node</td>
<td>112</td>
</tr>
<tr>
<td>Traversing a tree</td>
<td>112</td>
</tr>
<tr>
<td>Tree rewriting</td>
<td>290</td>
</tr>
<tr>
<td>Triple</td>
<td>324, 326</td>
</tr>
<tr>
<td>Tuple matching</td>
<td>664</td>
</tr>
<tr>
<td>Tuple Space</td>
<td>658, 664, 678</td>
</tr>
<tr>
<td>Tuple Space server</td>
<td>678</td>
</tr>
<tr>
<td>Two-scans assembly</td>
<td>380</td>
</tr>
<tr>
<td>Two-space copying</td>
<td>408, 420, 425, 437, 545</td>
</tr>
<tr>
<td>Type</td>
<td>449</td>
</tr>
<tr>
<td>Type checking</td>
<td>449, 673</td>
</tr>
<tr>
<td>Type declaration</td>
<td>449, 544, 551</td>
</tr>
<tr>
<td>Type equivalence</td>
<td>454–455, 551</td>
</tr>
<tr>
<td>Type extension</td>
<td>472</td>
</tr>
<tr>
<td>Type table</td>
<td>450, 552</td>
</tr>
<tr>
<td>Type variable</td>
<td>544, 551</td>
</tr>
<tr>
<td>Typed pointer</td>
<td>462</td>
</tr>
<tr>
<td>UDP</td>
<td>671</td>
</tr>
<tr>
<td>Unary relation</td>
<td>598</td>
</tr>
<tr>
<td>Unbound variable</td>
<td>599, 612, 625</td>
</tr>
<tr>
<td>Undiscriminated union</td>
<td>467</td>
</tr>
<tr>
<td>Unification</td>
<td>544, 596, 607, 609, 612, 634, 654–655</td>
</tr>
<tr>
<td>Uniform distribution in Linda</td>
<td>683</td>
</tr>
<tr>
<td>Uniform self-identifying data representation</td>
<td>281</td>
</tr>
<tr>
<td>Unify instruction</td>
<td>603, 609, 747</td>
</tr>
<tr>
<td>Union tag</td>
<td>467</td>
</tr>
<tr>
<td>Union type</td>
<td>466</td>
</tr>
<tr>
<td>Unmarshaling</td>
<td>672</td>
</tr>
<tr>
<td>Unrolling factor</td>
<td>511</td>
</tr>
<tr>
<td>Unwinding</td>
<td>566, 575, 589, 746</td>
</tr>
<tr>
<td>Update replicated copies</td>
<td>678</td>
</tr>
<tr>
<td>URL syntax</td>
<td>185</td>
</tr>
<tr>
<td>Usage count</td>
<td>317, 362</td>
</tr>
<tr>
<td>Useless non-terminal</td>
<td>38</td>
</tr>
<tr>
<td>User space</td>
<td>667</td>
</tr>
<tr>
<td>Validity span</td>
<td>463</td>
</tr>
<tr>
<td>Value</td>
<td>458</td>
</tr>
<tr>
<td>Value of a variable in an assembler</td>
<td>379</td>
</tr>
<tr>
<td>Variable descriptor</td>
<td>319</td>
</tr>
<tr>
<td>Variable-length graph node</td>
<td>587</td>
</tr>
<tr>
<td>Vector</td>
<td>467</td>
</tr>
<tr>
<td>Vector apply node</td>
<td>588, 595</td>
</tr>
<tr>
<td>Very busy expression</td>
<td>277</td>
</tr>
<tr>
<td>Virtual method</td>
<td>473, 479</td>
</tr>
<tr>
<td>Visiting a node</td>
<td>112</td>
</tr>
<tr>
<td>Visiting routine</td>
<td>220</td>
</tr>
<tr>
<td>Vocabulary</td>
<td>39</td>
</tr>
<tr>
<td>Void type</td>
<td>460</td>
</tr>
<tr>
<td>Wait operation</td>
<td>660, 670</td>
</tr>
<tr>
<td>WAM</td>
<td>597, 623, 636, 638, 648, 653</td>
</tr>
<tr>
<td>Warren Abstract Machine</td>
<td>597, 653</td>
</tr>
<tr>
<td>Weight of a subtree</td>
<td>310–311, 313–314, 316, 365, 390</td>
</tr>
<tr>
<td>Weighted register allocation</td>
<td>310, 339, 349</td>
</tr>
<tr>
<td>While statement</td>
<td>276, 507, 509, 741</td>
</tr>
<tr>
<td>Working stack</td>
<td>308, 412, 461, 483, 513, 515, 517</td>
</tr>
<tr>
<td>Yacc</td>
<td>176, 178, 234</td>
</tr>
<tr>
<td>Yield statement</td>
<td>485</td>
</tr>
<tr>
<td>Zeroth element</td>
<td>469, 533</td>
</tr>
</tbody>
</table>