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

```
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:

```
parameter_list →
    in_out_option identifier next_identifier_sequence_option
in_out_option → 'IN' | 'OUT' | ε
next_identifier_sequence_option →
    ',' 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.

```
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 $\texttt{T}_1\rightarrow\texttt{R}_1$ over more than one `Length`.

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

2.15 For example $T\rightarrow\alpha(R\bullet)^+\beta$. The $^+$ means that 1 or more Rs 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 β: $T\rightarrow\alpha(R)^+\bullet\beta$. 2. There is another R coming; the dot moves to the position immediately in front of the R: $T\rightarrow\alpha(\bullet R)^+\beta$.

2.16 See Figure Answers.3.

$$T\rightarrow\alpha\bullet(R_1\&R_2\&...\&R_n)\beta \quad \Rightarrow$$

$$\begin{aligned}&T\rightarrow\alpha\bullet R_1(R_2\&R_3\&...\&R_n)\beta\\&T\rightarrow\alpha\bullet R_2(R_1\&R_3\&...\&R_n)\beta\\&...\\&T\rightarrow\alpha\bullet R_n(R_1\&R_2\&...\&R_{n-1})\beta\end{aligned}$$

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:

```
integral_number → (• [0-9])+
integral_number → ([0-9])+ •                        <<<< recognized
fixed_point_number → (• [0-9])* '.' ([0-9])+
fixed_point_number → ([0-9])* • '.' ([0-9])+
fixed_point_number → ([0-9])* '.' (• [0-9])+
fixed_point_number → ([0-9])* '.' ([0-9])+ •  <<<< recognized
```

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.

```
0  (1, 0)     -     (2, 0)
1           (1, 1)     -     (2, 1)
2                                   (3, 2)     -     -
3  __________________________________________(3, 3)__-___-_
   (1, 0)  (1, 1)  (2, 0)  (2, 1)  (3, 2)  (3, 3)   -   -
```

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-

```
%Start    Comment

Layout              ([ \t])

AnyQuoted           (\\.)
StringChar          ([ˆ"\n\\]|{AnyQuoted})
CharChar            ([ˆ'\n\\]|{AnyQuoted})

StartComment        ("/*")
EndComment          ("*/")
SafeCommentChar     ([ˆ*\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;}
```

Figure Answers.5 Lex filter for removing comments from C program text.

mare.

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 MOD* (2**N*) is either *k MOD N* or *k 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 Figure Answers.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 ε-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` $\rightarrow$ `aNb | Nc; N` $\rightarrow$ 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.

(c) For $\beta \to \beta_1 ... \beta_n$, stack n pairs (β_k, κ), where κ = FIRST($\beta_{k+1} ... \beta_n$) if $\beta_{k+1} ... \beta_n$ does not produce ε, and κ = FIRST($\beta_{k+1} ... \beta_n$) $\cup\ \sigma$ if $\beta_{k+1} ... \beta_n$ does produce ε.
(d) On input `b`, the strong-LL(1) parser predicts `N` $\to$ ε, since `b` is in FOLLOW(`N`), but the full-LL(1) parser detects the error since `b` is not in {`a`, `$`}. 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 FOLLOW(`N`). Since `t` is in FOLLOW(`N`), there must be a full-LL(1) pair (`N`, σ), with `t` in σ. That pair has a full-LL(1) conflict, since we still do not know whether to predict A_k or the nullable alternative A_j.
(f) `S -> xNab | yNbb ; N -> a | ε`. FOLLOW_2(N) = {`ab`, `bb`}. Strong-LL(2) cannot decide between `N -> a {ab, bb}` which applies for look-aheads `{aa, ab}` and `N -> ε {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 -> a {ab}` which applies for look-ahead `{aa}` and `N -> ε {ab}` which applies for look-ahead `{ab}`, and between `N -> a {bb}` which applies for look-ahead `{ab}` and `N -> ε {bb}` which applies for look-ahead `{bb}`.

2.43 See Figure Answers.8.

stack continuation	FIRST set
`parenthesized_expression rest_expression EOF`	`{ '(' }`
`'(' expression ')' rest_expression EOF`	`'('`
`expression ')' rest_expression EOF`	`{ IDENTIFIER '(' }`
`term rest_expression ')' rest_expression EOF`	`{ IDENTIFIER '(' }`
`IDENTIFIER rest_expression ')' rest_expression EOF`	`IDENTIFIER`
`rest_expression ')' rest_expression EOF`	`{ '+' ε }`
`')' rest_expression EOF`	`')'`
`rest_expression EOF`	`{ '+' ε }`
`EOF`	`EOF`

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, α 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 $N \to \alpha$', the item set corresponding to the state on the top of the stack contains the item $N \to \alpha \bullet$. The dot can only be at the end of α when it has just passed over the last member of α, which must therefore be just below the top state on the stack. This reasoning applies successively to all other members of α, which must therefore also be on the stack.
(b) The item set preceding α on the stack must contain the item $N \to \bullet \alpha$, or no α would be recognized and no item $N \to \alpha \bullet$ would eventually be found. The item $N \to \bullet \alpha$ must have originated from some item $P \to \beta \bullet N \gamma$. The presence of this item guarantees that a transition on N is possible, leading to a state that includes $P \to \beta N \bullet \gamma$.

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.

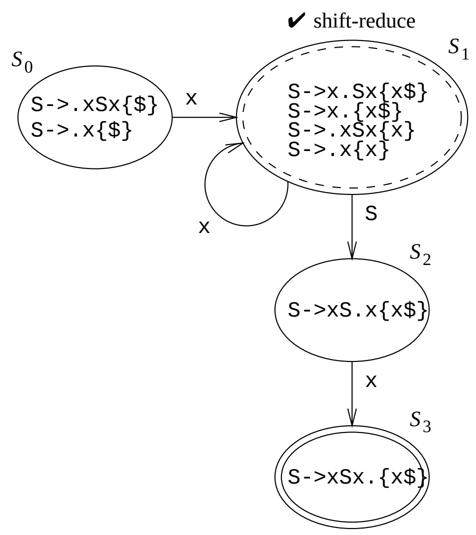


Figure Answers.9 The SLR(1) automaton for `S` $\rightarrow$ `x S x | x`.

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_0, 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_{4a} 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.

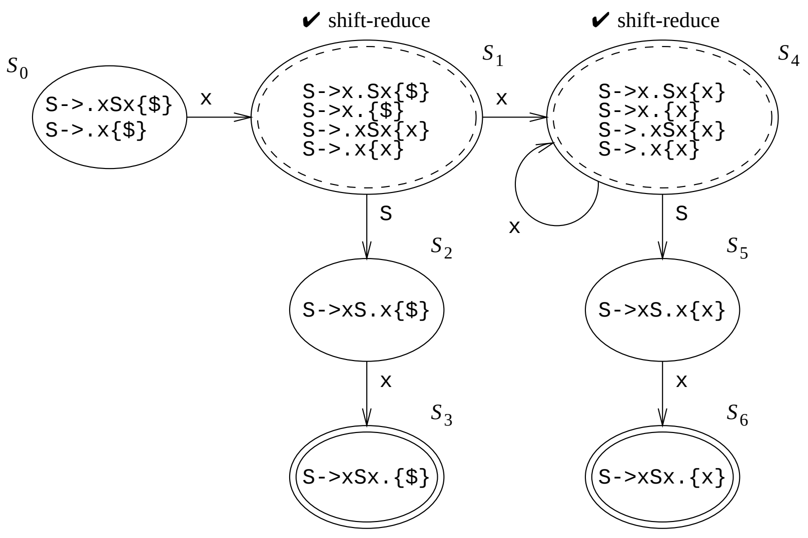


Figure Answers.10 The LR(1) automaton for `S → x S x | x`.

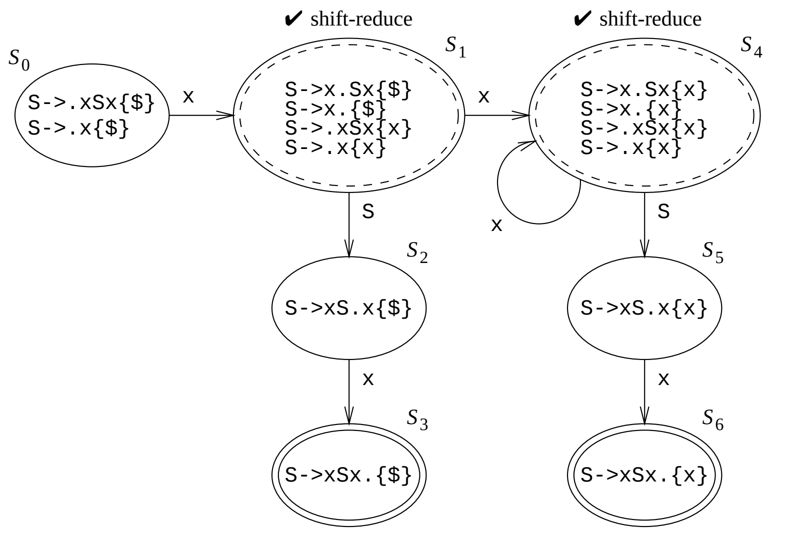


Figure Answers.11 The LALR(1) automaton for `S → x S x | x`.

```
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.

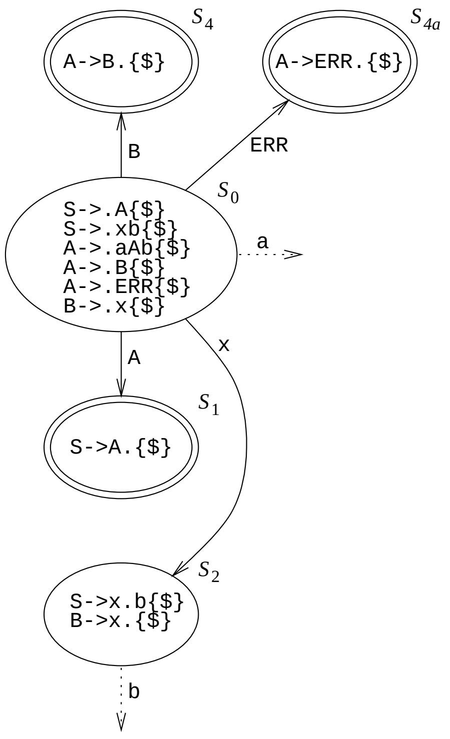


Figure Answers.13 State S_0 of the error-recovering parser.

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.

```
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.

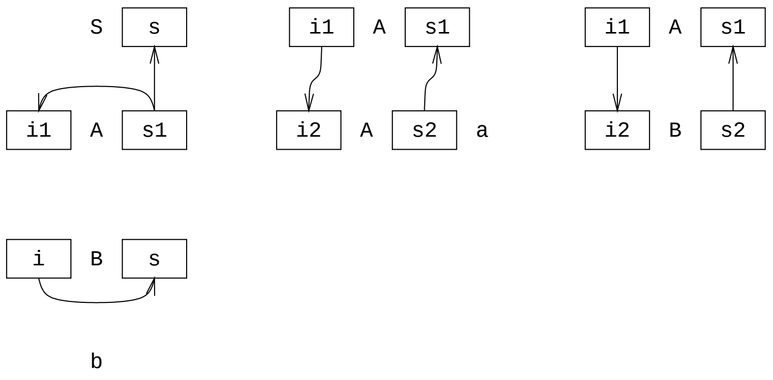


Figure Answers.15 Dependency graphs for S, A, and B.

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-SI graph of S.
(b) ({i1, i2}, {s1, s2}).

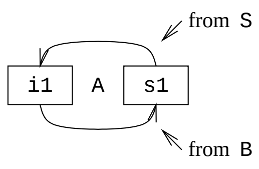


Figure Answers.16 IS-SI graph of A.

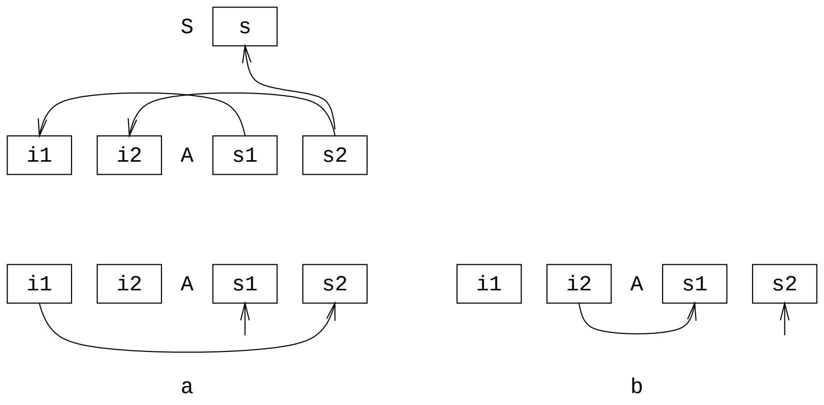


Figure Answers.17 Dependency graphs for S and A.

(c) 1 routine:

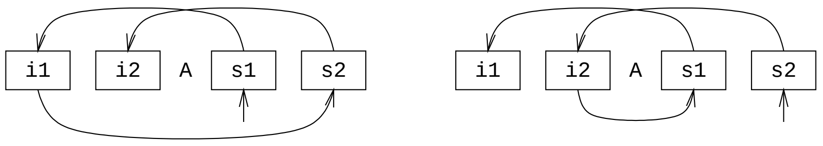
parent prepares S.i1 and S.i2:

```
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:

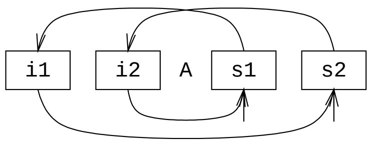


Figure Answers.18 IS-SI graph sets and IS-SI graph of A.

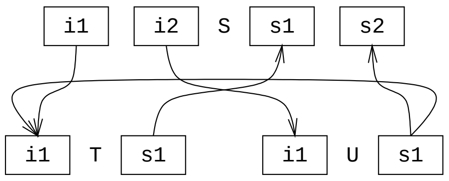


Figure Answers.19 Dependency graph of S.

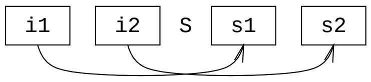


Figure Answers.20 IS-SI graph of S.

parent prepares `S.i2`:

```
PROCEDURE Visit_1 to S (i1, i2, s1, s2):
    // 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 S.s2 TO f4(U.s);          // → S.s2 available
```

parent receives `S.s2`

parent prepares `S.i1`:

```
PROCEDURE Visit_2 to S (i1, i2, s1, s2):
    // S.i2, U.i, U.s, S.s2, S.i1 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
```

parent receives `S.s1`

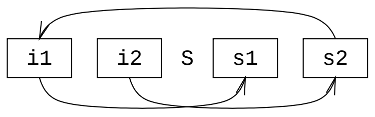


Figure Answers.21 IS-SI graph of S.

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 `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 `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 `Visit to S()` shown in Figure Answers.22.

parent prepares `S.i1` and `S.i2`:

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

Figure Answers.22 `Visit to S()`.

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, `Concat` concatenates a character representation to a string, and `Checked number value` converts a string to an integer, checking each digit in the string.

```
Number(SYN value) →
    Digit_Seq  Base_Tag
    ATTRIBUTE RULES
        SET Number .value TO Checked number value(
            Digit_Seq .repr,
         Base_Tag .base);

Digit_Seq(SYN repr) →
    Digit_Seq[1]  Digit
    ATTRIBUTE RULES
        SET Digit_Seq .repr TO
            Concat(Digit_Seq[1] .repr , Digit .repr);
|
    Digit
    ATTRIBUTE RULES
        SET Digit_Seq .repr TO Digit .repr;

Digit(SYN repr) →
    Digit_Token
    ATTRIBUTE RULES
        SET Digit .repr TO Digit_Token .repr [0];

Base_Tag(SYN base) →
    'B'
    ATTRIBUTE RULES
        SET Base_Tag .base TO 8;
|
    'D'
    ATTRIBUTE RULES
        SET Base_Tag .base TO 10;
```

Figure Answers.23 An L-attributed grammar for `Number`.

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

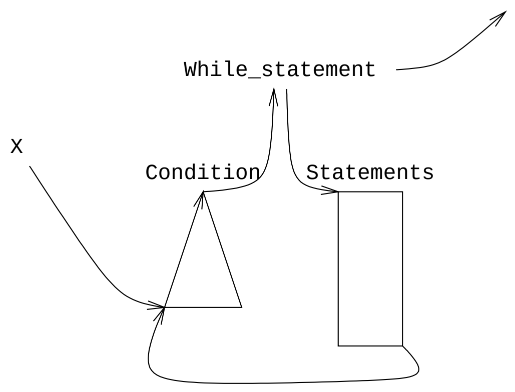


Figure Answers.24 Threaded AST of the while statement.

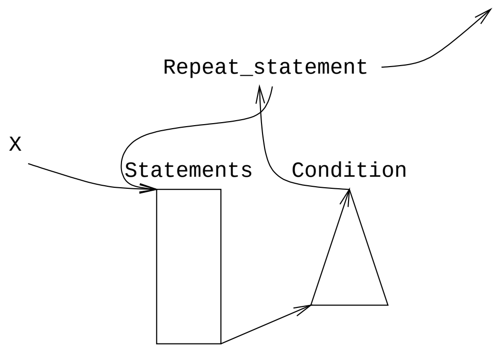


Figure Answers.25 Threaded AST of the repeat statement.

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.

```
#include    "parser.h"      /* for types AST_node and Expression */
#include    "thread.h"      /* for self check */

                                        /* PRIVATE */
static AST_node *Thread_expression(Expression *expr, AST_node *succ) {
    switch (expr->type) {
    case 'D':
        expr->successor = succ; return expr;
        break;
    case 'P':
        expr->successor = succ;
        return
            Thread_expression(expr->left,
                Thread_expression(expr->right, expr)
            );
        break;
    }
}
                                        /* PUBLIC */
AST_node *Thread_start;

void Thread_AST(AST_node *icode) {
    Thread_start = Thread_expression(icode, 0);
}
```

Figure Answers.26 Alternative threading code for the demo compiler from Section 1.2.

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* = ∅, 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:

$$OUT(N) = \bigcap_{M=\text{dynamic successor of } N} IN(M)$$

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

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

$GEN(1) = 0$	$KILL(1) = 1$
$GEN(2) = 0$	$KILL(2) = 0$
$GEN(3) = 1$	$KILL(3) = 0$
$GEN(4) = 0$	$KILL(4) = 0$
$GEN(5) = 0$	$KILL(5) = 0$
$GEN(6) = 0$	$KILL(6) = 0$

(c) Solving the equations gives:

$IN(1) = 0$	$OUT(1) = 1$
$IN(2) = 1$	$OUT(2) = 1$
$IN(3) = 1$	$OUT(3) = 0$
$IN(4) = 0$	$OUT(4) = 0$
$IN(5) = 0$	$OUT(5) = 0$
$IN(6) = 0$	$OUT(6) = 0$

The conclusion is that the evaluation of `x*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.

```
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:

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

The interpreter:

```
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

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

and

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

and generate

```
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:

```
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.

Equation	Value
`a = 1`	1.0
`b = 0.7 a`	0.7
`c = 0.3 a`	0.3
`d = b`	0.7
`e = 0.1 (d+f)`	0.7
`f = g`	6.3
`g = 0.9 (d+f)`	6.3
`h = c`	0.3
`i = 0.4 h`	0.12
`j = 0.4 h`	0.12
`k = 0.2 h`	0.06
`l = i`	0.12
`m = j`	0.12
`n = k`	0.06
`o = e`	0.7
`p = l+m+n`	0.30
`q = o+p`	1.0

Figure Answers.30 Traffic equations and their solution for Figure 4.97.

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

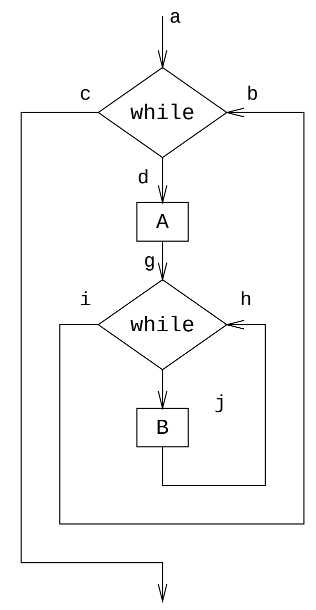


Figure Answers.31 Flow graph for static profiling of a nested while loop.

Equation	Value
`a = 1`	`1.0`
`b = i`	`9.0`
`c = 0.1 (a+b)`	`1.0`
`d = 0.9 (a+b)`	`9.0`
`e = d`	`9.0`
`f = e`	`9.0`
`g = f`	`9.0`
`h = j`	`81.0`
`i = 0.1 (g+h)`	`9.0`
`j = 0.9 (g+h)`	`81.0`

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

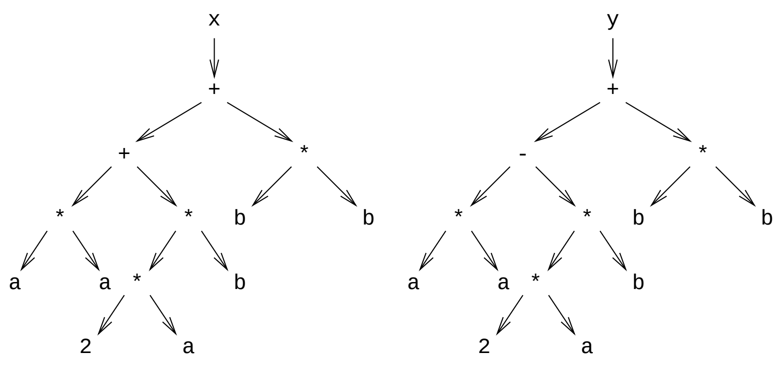


Figure Answers.33 The dependency graph before common subexpression elimination.

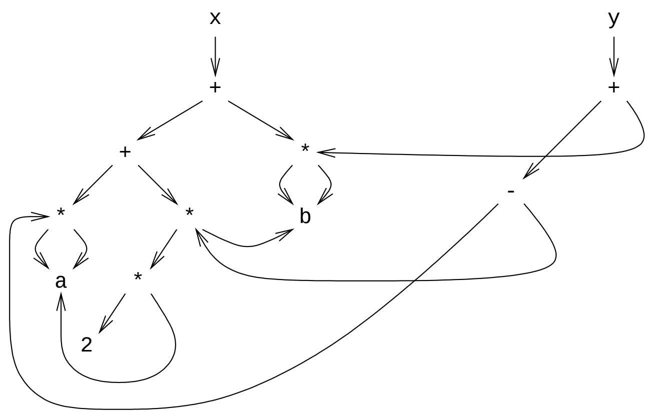


Figure Answers.34 The dependency graph after common subexpression elimination.

```
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
```

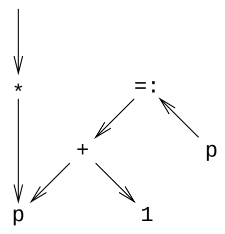


Figure Answers.35 The dependency graph of the expression `*p++`.

position	triple
1	`a * a`
2	`a * 2`
3	`@2 * b`
4	`@1 + @3`
5	`b * b`
6	`@4 + @5`
7	`@6 =: x`
8	`@1 - @3`
9	`@8 + @5`
10	`@9 =: y`

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

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
```

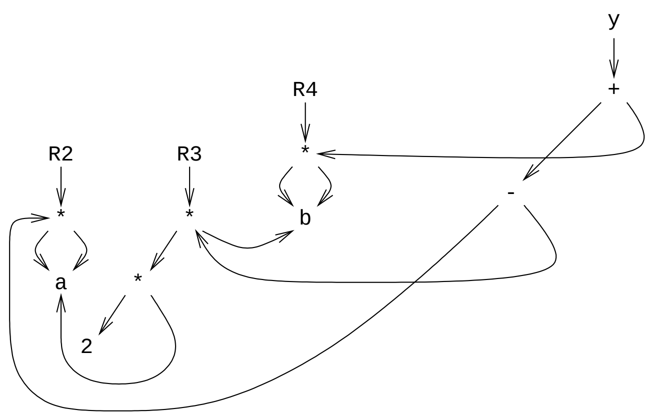


Figure Answers.37 The dependency graph of Figure Answers.34 with first ladder sequence removed.

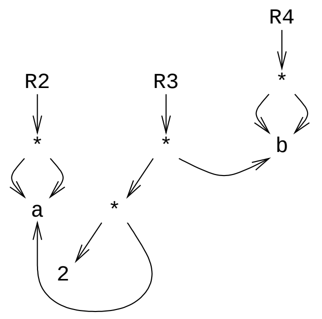


Figure Answers.38 The dependency graph of Figure Answers.37 with second ladder sequence removed.

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 `c` with color 1, `b` with color 2, `a` with color 3, `e` with color 2, `d` with color 1.

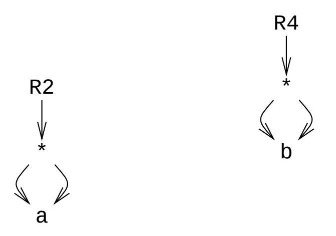


Figure Answers.39 The dependency graph of Figure Answers.38 with third ladder sequence removed.

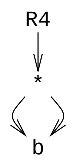


Figure Answers.40 The dependency graph of Figure Answers.39 with fourth ladder sequence removed.

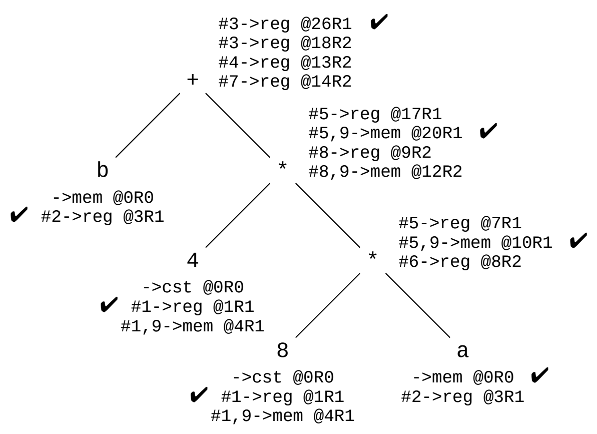


Figure Answers.41 Bottom-up pattern matching with costs and register usage.

4.27 (a) See Figure Answers.43.
(b) 3.

4.28 For the first instruction in the sequence we have 20*2=40 combinations, using `R1,R1` and `R1,R2`, or more compactly `{R1},{R1,R2}`. For the second instruction we have 20*2*3=120 combinations, using `{R1,R2},{R1,R2,R3}`; for the further instructions we have 20*3*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.

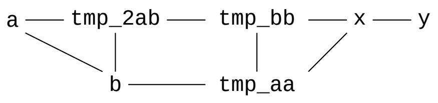


Figure Answers.43 The register interference graph for Exercise 4.27.

$4800\times180^{N-2}$ combinations.
Estimating a weekend at 2.5 days, each of about 80 000 seconds, we have about 2×10^{11} μseconds, or 2×10^{10} tests. So we want the largest N for which $4800\times180^{N-2}$ is smaller than 2×10^{10}. Now, $4800\times180^{4-2}=1.5\times10^{8}$ and $4800\times180^{5-2}=2.7\times10^{10}$, so N=4.

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

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

we get

```
{int i = 0; if (i < 1) goto _L_end; else Q(); _L_end:;}
```

(b)

```
{int i = 0; if (0 < 1) goto _L_end; else Q(); _L_end:;}
```

(c)

```
{int i = 0; if (1) goto _L_end; else Q(); _L_end:;}
```

(d)

```
{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
```

can be translated by

```
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 mallocked but not freed; the program then also prints the routines 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 allocation 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 collection 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 >= 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;
A, B: t1; B: t2; C: t3;
```

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?lvalue`:`lvalue 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

$$\begin{aligned} zeroth\_offset(A) = -(&LB_1 \times LEN\_PRODUCT_1 \\ &+LB_2 \times LEN\_PRODUCT_2 \\ &... \\ &+LB_n \times LEN\_PRODUCT_n) \end{aligned}$$

Figure Answers.46 Formula for *zeroth_offset*(A).

$$\begin{aligned} &base(A)+zeroth\_offset(A) \\ &\quad +i_1 \times LEN\_PRODUCT_1 + ... + i_n \times LEN\_PRODUCT_n \end{aligned}$$

Figure Answers.47 The address of $A[i_1, ..., i_n]$.

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.

`IsShape_Shape_Shape`
`IsRectangle_Shape_Rectangle`
`IsSquare_Shape_Shape`
`SurfaceArea_Shape_Rectangle`

method table for `Rectangle`

`IsShape_Shape_Shape`
`IsRectangle_Shape_Rectangle`
`IsSquare_Shape_Square`
`SurfaceArea_Shape_Rectangle`

method table for `Square`

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 `instanceof` operator becomes easy, see Figure Answers.49.

6.14 The code for these calls is:

```
(*(e->dispatch_table[0]))(e);
(*(e->dispatch_table[2]))((class D *)((char *)e + sizeof(class C)));
(*(e->dispatch_table[3]))((class D *)((char *)e + sizeof(class C)));
```

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, 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
    I_1: statement sequence_1
    ...
    I_n: statement sequence_n
    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;
label_1:
    code for statement sequence_1
    GOTO label_next;
    ...
label_n:
    code for statement sequence_n
    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: */
}
```

Figure Answers.52 A while loop that is exactly equivalent to the for-loop of Figure 6.41.

```
    i := lower bound;
    tmp_ub := upper bound - unrolling factor + 1;
    IF i > tmp_ub THEN GOTO end_label;
    tmp_loop_count := (tmp_ub - i) DIV unrolling factor + 1;
loop_label:
    code for statement sequence
    i := i + 1;
    ...
    code for statement sequence
    i := i + 1;                  // these two lines unrolling factor times
    tmp_loop_count := tmp_loop_count - 1;
    IF tmp_loop_count = 0 THEN GOTO end_label;
    GOTO loop_label;
end_label:
```

Figure Answers.53 Loop-unrolling code.

6.31 A divide-by-zero exception may occur for the first division; the handler may execute any code, for example perform assignments to A and B:

```
begin
    X := A / B;
    Y := A / B;
exception
    when DIVIDE_BY_ZERO_ERROR =>
        B := 1; A := A - 1;
end;
```

The ‘optimized’ code will then clearly behave differently from the original code.

6.32 Two new operations must be supported: instantiation of a generic routine which is a parameter, and passing a generic routine as a parameter to an instantiation. The consequences from a compiler construction point of view are modest. When implementing instantiation through expansion, the first instantiation to be dealt with resides within a non-generic unit. Therefore, a generic routine parameter is available as an AST, which can be copied and processed just as the AST of an ordinary generic unit.

One issue that must be dealt with, weather the language has generic routine parameters or not, is cycle detection: when one generic unit contains an instantiation of another generic unit, the result is a chain of instantiations, which must be terminated by the instantiation of a generic unit which does not contain further instantiations. If this is not the case, the implementation through expansion scheme will fail. Without generic routine parameters, detecting this situation is easy: when a generic unit occurs twice on the instantiation chain, there is a cycle. With generic routine parameters, when an identical instantiation (same generic unit and same instantiation parameters) occurs twice on the instantiation chain, there is a cycle.

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 -> b -> b) -> b -> [a] -> b
```

7.3 The definition of the dot operator is

```
dot :: (a -> b) -> (c -> a) -> c -> b
dot f g x = f (g x)
```

7.4 The functional-core equivalent of `"list"` is `(Cons 'l' (Cons 'i' (Cons 's' (Cons 't' []))))`, that of `[1..481]` is `range 1 481`.

7.5 See Figure Answers.54.

```
unique a1 = if (_type_constr a1 == Cons &&
                _type_constr (_type_field 2 a1) == Cons) then
              let
                 a  = _type_field 1 a1
                 b  = _type_field 1 (_type_field 2 a1)
                 cs = _type_field 2 (_type_field 2 a1)
              in
                if (a == b) then a : unique cs
                else a : unique (b:cs)
            else
              a1
```

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.

```
qsort []     = []
qsort (x:xs) = qsort (mappend qleft xs) ++ [x]
               ++ qsort (mappend qright xs)
               where
                 qleft y  = if (y < x)  then [y] else []
                 qright y = if (y >= x) then [y] else []
```

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-

```
qsort []     = []
qsort (x:xs) = qsort (mappend (qleft x) xs) ++ [x]
               ++ qsort (mappend (qright x) xs)

qleft x y  = if (y < x)  then [y] else []

qright x y = if (y >= x) then [y] else []
```

Figure Answers.56 A functional-core lambda-lifted equivalent of the function `qsort`.

and else-expressions before calling the conditional function:

```
cond b t e = if b then t else e

fac n = cond (n==0) 1 (n * fac (n-1))
```

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

```
tree 0 val = val
tree n val = let t = tree (n-1) val in t*t

main = print (tree 10 (fac 20))
```

7.12 `Eval()` must be extended with an additional case to handle the indirection nodes:

```
case IND:
    node = node->nd.ind;
    break;
```

Also, the update must be changed into:

```
    root->tag = IND;    /* update */
    root->ind = node;
```

7.13 C does not allow forward references to local variables.

```
Pnode rep(Pnode *arg)
{
    Pnode n = arg[0];
    Pnode lst = Cons(n, NULL);
    lst->cons.tl = lst;

    return lst;
}
```

7.15 The function `f` is strict in `p` only.

7.16 Substitute `equal_evaluated`, `mul_evaluated`, and `sub_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:

```
Pnode fac_evaluated(Pnode _arg0) {
    Pnode n = _arg0;

    return Num(n->nd.num == Num(0)->nd.num)->nd.num ? Num(1)
        : Num(n->nd.num *
            fac_evaluated(Num(n->nd.num - Num(1)->nd.num))->nd.num);
}
```

Replace *B* `?` *F*`(`*x*`)` `:` *F*`(`*y*`)` by *F*`(`*B* `?` *x* `:` *y*`)`; replace `Num(`*x*`)->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.

7.18 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).

7.19 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 `?- grandparent(arne, Z)` left-to-right is more efficient. It first binds `Z` to `james` and tries `parent(james, Z)`, which fails. Next, it binds `Z` to `sachiko`, and tries `parent(sachiko, Z)`, which succeeds with `Z=rivka`. With a right-to-left search, the system will first try to solve `parent(Y,Z)`, where both `Y` and `Z` are unbound. So, the program will try all possible combinations of `Y` and `Z` for which `parent(Y,Z)` holds, until it finds a combination for which `parent(arne, Y)` holds. With the program of Section 8.1, the fourth combination (`sachiko`, `rivka`) succeeds. For `?- grandparent(X, 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(arne,X),<<("X",X)
```

Applying 'Attach clauses' with optimization results in:

```
[gp(arne,X) ?= gp(X1,Z1)],pa(X1,Y1),pa(Y1,Z1),<<("X",X)
```

Unification results in:

```
[gp(arne,X) == gp(arne,X)],pa(arne,Y1),pa(Y1,X),<<("X",X)
```

Then, 'Match'removes the two unified goals:

```
pa(arne,Y1),pa(Y1,X),<<("X",X)
```

Applying 'Attach clauses' with optimizations results in:

```
[pa(arne,Y1) ?= pa(arne,james)],pa(Y1,X),<<("X",X)
[pa(arne,Y1) ?= pa(arne,sachiko)],pa(Y1,X),<<("X",X)
```

Unification on the top entry gives:

```
[pa(arne,james) ?= pa(arne,james)],pa(james,X),<<("X",X)
[pa(arne,Y1) ?= pa(arne,sachiko)],pa(Y1,X),<<("X",X)
```

'Match' then removes the unified goals:

```
pa(james,X),<<("X",X)
[pa(arne,Y1) ?= pa(arne,sachiko)],pa(Y1,X),<<("X",X)
```

'Attach clauses' on the top entry leaves no entries, so we are left with

```
[pa(arne,Y1) ?= pa(arne,sachiko)],pa(Y1,X),<<("X",X)
```

which after unification and matching results in

```
pa(sachiko,X),<<("X",X)
```

Again an 'Attach clauses' is required (with optimizations):

```
[pa(sachiko,X) ?= pa(sachiko,rivka)],<<("X",X)
```

Unification succeeds with `X=rivka`:

```
[pa(sachiko,rivka) == pa(sachiko,rivka)],<<("X",rivka)
```

'Match' removes the unified goals, leaving us with a display goal:

```
<<("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.

```
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

```
/* 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

```
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 `msg` to processor `P`, do:

```
out("message", "P", msg);
```

If processor `P` wants to receive a message, it executes:

```
in("message", "P", ? &msg);
```

To create a shared variable `X` with initial value `V` execute:

```
out("shared variable", "X", V);
```

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

```
read("shared variable", "X", ? &var);
```

To write a new value into the variable, execute:

```
in("shared variable", "X", ? &var); /* discard old value */
out("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

Aho, A. V., Ganapathi, M., and Tjiang, S. W. K. (Oct. 1989). Code generation using tree pattern matching and dynamic programming. *ACM Trans. Programming Languages and Systems*, **11**(4), 491-516

Aho, A. V. and Johnson, S. C. (March 1976). Optimal code generation for expression trees. *J. ACM*, **23**(3), 488-501

Aho, A. V., Johnson, S. C., and Ullman, J. D. (Jan. 1977). Code generation for expressions with common subexpressions. *J. ACM*, **24**(1), 146-160

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques and Tools*. Addison-Wesley

Aho, A. V. and Ullman, J. D. (1973). *The Theory of Parsing, Translation and Compiling*, Vol. I: *Parsing*, Vol. II: *Compiling*. Prentice Hall

Ahuja, S., Carriero, N., and Gelernter, D. (Aug. 1986). Linda and friends. *IEEE Computer*, **19**(8), 26-34

Aït-Kaci, H. (1991). *Warren's Abstract Machine − A Tutorial Reconstruction*. MIT Press

Almasi, G. S. and Gottlieb, A. (1994). *Highly Parallel Computing*. 2nd Edn. Benjamin/Cummings

Anderson, J. P. (March 1964). A note on some compiling algorithms. *Comm. ACM*, **7**(3), 149-150

Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. (Oct. 1991). Scheduler activations: Effective kernel support for the user-level management of parallelism. In *13th ACM Symposium on Operating Systems Principles* (Edward D. Lazowska, ed.), pp. 95-109. ACM

Andrews, G. R. (March 1991). Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, **23**(1), 49-90

Anonymous (1840). *Des Freiherrn von Münchhausen wunderbare Reisen und Abentheuer zu Wasser und zu Lande*. Dieterichsche Buchhandlung, Göttingen

Appel, A. W. (1987). Garbage collection can be faster than stack allocation. *Information Processing Letters*, **25**, 275-279

Appel, A. W. (1992). *Compiling with Continuations*. Cambridge University Press

Appel, A. W. (1997). *Modern Compiler Implementation in C/ML/Java*. Cambridge University Press

Appel, A. W. and Supowit, K. J. (June 1987). Generalizations of the Sethi−Ullman algorithm for register allocation. *Software − Practice and Experience*, **17**(6), 417-421

Assmann, W. (Oct. 1992). Another solution of scoping problems in symbol tables. In *Compiler Construction, 4th International Conference, CC'92* (U. Kastens and P. Pfahler, ed.), pp. 66-72. Springer-Verlag

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

Bacon, D. F., Graham, S. L., and Sharp, O. J. (Dec. 1994). Compiler transformations for high-performance computing. *ACM Computing Surveys*, **26**(4), 345-420

Bal, H. E., Bhoedjang, R., Hofman, R., Jacobs, C., Langendoen, K., Rühl, T., and Kaashoek, M. F. (Feb. 1998). Performance evaluation of the Orca shared-object system. *ACM Trans. Computer Systems*, **16**(1), 1-40

Bal, H. E. and Grune, D. (1994). *Programming Language Essentials*. Addison-Wesley

Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S. (March 1992). Orca: A language for parallel programming of distributed systems. *IEEE Trans. Software Engineering*, **18**(3), 190-205

Bal, H. E., Steiner, J. G., and Tanenbaum, A. S. (Sept. 1989). Programming languages for distributed computing systems. *ACM Computing Surveys*, **21**(3), 261-322

Barach, D. R., Taenzer, D. H., and Wells, R. E. (May 1982). A technique for finding storage allocation errors in C-language programs. *ACM SIGPLAN Notices*, **17**(5), 16-23

Barrett, D. A. and Zorn, B. G. (June 1995). Garbage collection using a dynamic threatening boundary. *ACM SIGPLAN Notices*, **30**(6), 301-314

Baskett, F. (April 1978). The best simple code generation technique for while, for, and do loops. *ACM SIGPLAN Notices*, **13**(4), 31-32

Bell, J. R. (June 1973). Threaded code. *Commun. ACM*, **16**(6), 370-372

Ben-Ari, M. (July 1984). Algorithms for on-the-fly garbage collection. *ACM Trans. Programming Languages and Systems*, **6**(3), 333-344

Bernstein, R. L. (Oct. 1985). Producing good code for the case statement. *Software − Practice and Experience*, **15**(10), 1021-1024

Bertsch, E. and Nederhof, M.-J. (Jan. 1999). On failure of the pruning technique in ‘Error repair in shift-reduce parsers’. *ACM Trans. Programming Languages and Systems*, **21**(1), 1-10

Bhoedjang, R. A. F., Rühl, T., and Bal, H. E. (Nov. 1998). User-level network interface protocols. *IEEE Computer*, **31**(11), 53-60

Bird, R. J. (1998). *Introduction to Functional Programming using Haskell*. 2nd Edn. Prentice-Hall Europe

Birtwistle, G. M., Dahl, O.-J., Myhrhaug, B., and Nygaard, K. (1975). *SIMULA begin*. Petrocelli/Charter

Bjornson, R., Carriero, N., Gelernter, D., and Leichter, J. (Jan. 1988). Linda, the Portable Parallel. *Technical Report RR-520*, Yale University

Blume, W., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T., Lee, J., Padua, D., Paek, Y., Pottenger, B., Rauchwerger, L., and Tu, P. (Dec. 1996). Parallel programming with Polaris. *IEEE Computer*, **29**(12), 78-83

Boehm, H.-J. (June 1993). Space-efficient conservative garbage collection. *ACM SIGPLAN Notices*, **28**(6), 197-206

Boehm, H.-J. and Weiser, M. (Sept. 1988). Garbage collection in an uncooperative environment. *Software − Practice and Experience*, **18**(9), 807-820

Boyd, M. R. and Whalley, D. B. (June 1993). Isolation and analysis of optimization errors. *ACM SIGPLAN Notices*, **28**(6), 26-25

Boyland, J. and Castagna, G. (Oct. 1997). Parasitic methods − Implementation of multi-methods for Java. *ACM SIGPLAN Notices*, **32**(10), 66-76

Briggs, P., Cooper, K. D., Kennedy, K., and Torczon, L. (July 1989). Coloring heuristics for register allocation. *ACM SIGPLAN Notices*, **24**(7), 275-284

Brooker, R. A., MacCallum, I. R., Morris, D., and Rohl, J. S. (1963). The compiler compiler. *Annual Review Automatic Programming*, **3**, 229-322

Bruno, J. and Sethi, R. (July 1976). Code generation for a one-register machine. *J. ACM*, **23**(3), 502-510

Burn, G. L. (1991). *Lazy Functional Languages: Abstract Interpretation and Compilation*. Pitman

Carriero, N. and Gelernter, D. (Sept. 1989). How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, **21**(3), 323-357

Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., and Markstein, P. W. (1981). Register allocation via coloring. *Computer Languages*, **6**(1), 45-57

Chapman, N. P. (1987). *LR Parsing: Theory and Practice*. Cambridge University Press

Cheney, C. J. (Nov. 1970). A non-recursive list compacting algorithm. *Commun. ACM*, **13**(11), 677-678

Cocke, J. and Kennedy, K. (Nov. 1977). An algorithm for reduction of operator strength. *Commun. ACM*, **20**(11), 850-856

Cohen, J. (1981). Garbage collection of linked data structures. *ACM Computing Surveys*, **13**(3), 341-367

Collins, G. E. (Dec. 1960). A method for overlapping and erasure of lists. *Commun. ACM*, **3**(12), 655-657

Colomb, R. M. (March 1988). Assert, retract and external processes in Prolog. *Software − Practice and Experience*, **18**(3), 205-220

Conway, M. E. (July 1963). Design of a separable transition-diagram compiler. *Commun. ACM*, **6**(7), 396-408

Cooper, K. D., Hall, M. W., and Torczon, L. (March 1992). Unexpected side effects of inline substitution: A case study. *ACM Letters on Programming Languages and Systems*, **1**(1), 22-32

Coulouris, G., Dollimore, J., and Kindberg, T. (1994). *Distributed Systems – Concepts and Design*. 2nd Edn. Addison-Wesley

Davidson, J. W. and Fraser, C. W. (June 1984a). Automatic generation of peephole optimizations. *ACM SIGPLAN Notices*, **19**(6), 111-116

Davidson, J. W. and Fraser, C. W. (Oct. 1984b). Code selection through object code optimization. *ACM Trans. Programming Languages and Systems*, **6**(4), 505-526

Davidson, J. W. and Whalley, D. B. (Jan. 1989). Quick compilers using peephole optimizations. *Software – Practice and Experience*, **19**(1), 79-97

Debray, S. K. (1994). Implementing logic programming systems – The quiche-eating approach. In *Implementations of Logic Programming Systems* (E. Tick and G. Succi, ed.), pp. 65-88. Kluwer Academic

DeRemer, F. L. (July 1971). Simple LR(*k*) grammars. *Commun. ACM*, **14**(7), 453-460

DeRemer, F. L. (1974). Lexical analysis. In *Compiler Construction, An Advanced Course* (F.L. Bauer and J. Eickel, ed.), pp. 109-120. Springer-Verlag

Dewar, R. (June 1975). Indirect threaded code. *Commun. ACM*, **18**(6), 330-331

Dijkstra, E. W. and Lamport, L. (Nov. 1978). On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, **21**(11), 966-975

Douence, R. and Fradet, P. (March 1998). A systematic study of functional language implementations. *TOPLAS*, **20**(2), 344-387

Driesen, K. and Hölzle, U. (Oct. 1995). Minimizing row displacement dispatch tables. *ACM SIGPLAN Notices*, **30**(10), 141-155

DuJardin, E., Amiel, E., and Simon, E. (Jan. 1998). Fast algorithms for compressed multimethod dispatch tables. *ACM Trans. Programming Languages and Systems*, **20**(1), 116-165

Dybvig, R. K. (1996). *The Scheme Programming Language: ANSI Scheme*. 2nd Edn. Prentice Hall

Earley, J. (Feb. 1970). An efficient context-free parsing algorithm. *Commun. ACM*, **13**(2), 94-102

Engelfriet, J. (1984). Attribute grammars: attribute evaluation methods. In *Methods and Tools for Compiler Construction* (B. Lorho, ed.), pp. 102-138. Cambridge University Press

Engelfriet, J. and De Jong, W. (1990). Attribute storage optimization by stacks. *Acta Informatica*, **27**, 567-581

Farrow, R. (June 1984). Sub-protocol-evaluators for attribute grammars. *ACM SIGPLAN Notices*, **19**(6), 70-80

Farrow, R. and Yellin, D. (1986). A comparison of storage optimizations in automatically generated attribute evaluators. *Acta Informatica*, **23**, 393-427

Feijs, L. M. G. and Van Ommering, R. C. (Aug. 1997). Abstract derivation of transitive closure algorithms. *Information Processing Letters*, **63**(3), 159-164

Fortes Gálvez, J. (Oct. 1992). Generating LR(1) parsers of small size. In *Compiler Construction, 4th International Conference, CC'92* (U. Kastens and P. Pfahler, ed.), pp. 16-29. Springer-Verlag

Fraser, C. W. and Hanson, D. R. (1995). *A Retargetable C Compiler – Design and Implementation*. Benjamin/Cummings, Redwood City, Ca.

Freeman, E., Hupfer, S., and Arnold, K. (1999). *JavaSpaces – Principles, Patterns, and Practice*. Addison-Wesley

Freiburghouse, R. A. (Nov. 1974). Register allocation via usage counts. *Commun. ACM*, **17**(11), 638-642

Geurts, L., Meertens, L., and Pemberton, S. (1989). *The ABC Programmer's Handbook*. Prentice Hall

Goldberg, B. (June 1991). Tag-free garbage collection for strongly typed programming languages. *ACM SIGPLAN Notices*, **26**(6), 165-176

Granlund, T. and Kenner, R. (July 1992). Eliminating branches using a super-optimizer and the GNU C compiler. *ACM SIGPLAN Notices*, **27**, 341-352

Griswold, R. E. and Griswold, M. T. (1983). *The Icon Programming Language*. Prentice Hall

Griswold, R. E. and Griswold, M. T. (1986). *The Implementation of the Icon Programming Language*. Princeton University Press

Grune, D. and Jacobs, C. J. H. (1990). *Parsing Techniques: a Practical Guide*. Ellis Horwood

Hall, M. W., Anderson, J. M., Amarasinghe, S. P., Murphy, B. R., Liao, S.-W., Bugnion, E., and Lam, M. S. (Dec. 1996). Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, **29**(12), 84-89

Hanson, D. R. (Dec. 1985). Compact recursive-descent parsing of expressions. *Software – Practice and Experience*, **15**(12), 1205-1212

Hartmann, A. C. (1977). *A Concurrent-Pascal compiler for minicomputers*. Springer-Verlag

Hastings, R. and Joyce, B. (1992). Purify – Fast detection of memory leaks and access errors. In *Winter '92 USENIX Conference* (Eric Allman, ed.), pp. 125-136. USENIX Association

Hemerik, C. and Katoen, J. P. (Jan. 1990). Bottom-up tree acceptors. *Science of Computer Programming*, **13**, 51-72

Hennessy, J. L. and Mendelsohn, N. (Sept. 1982). Compilation of the Pascal case statement. *Software – Practice and Experience*, **12**(9), 879-882

Holmes, J. (1995). *Object-Oriented Compiler Construction*. Prentice-Hall International

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley

Hummel, J., Hendren, L. J., and Nicolau, A. (Aug. 1994). A framework for data dependence testing in the presence of pointers. In *1994 International Conference on Parallel Processing* (Vol. II) (K.C. Tai, ed.), pp. 216-224. CRC Press

Hwu, W.-M. and Chang, P. P. (July 1989). Inline function expansion for compiling C programs. *ACM SIGPLAN Notices*, **24**(7), 246-257

Jazayeri, M. and Pozefsky, D. (Oct. 1981). Space efficient storage management for attribute grammars. *ACM Trans. Programming Languages and Systems*, **3**(4), 388-404

Johnson, W. L., Porter, J. S., Ackley, S. I., and Ross, D. T. (Dec. 1968). Automatic generation of efficient lexical processors using finite state techniques. *Commun. ACM*, **11**(12), 805-813

Johnsson, T. (June 1984). Efficient compilation of lazy evaluation. *ACM SIGPLAN Notices*, **19**(6), 58-69

Johnsson, T. (Sept. 1987). Attribute grammars as a functional programming paradigm. In *3rd Functional Programming Languages and Computer Architecture Conference* (G. Kahn, ed.), pp. 154-173. Springer-Verlag

Jones, N. D., Gomard, C. K., and Sestoft, P. (April 1993). *Partial Evaluation and Program Generation*. Prentice Hall

Jones, R. and Lins, R. (1996). *Garbage Collection – Algorithms for Automatic Dynamic Memory Management*. John Wiley

Jourdan, M. (June 1984). Strongly non-circular attribute grammars and their recursive evaluation. *ACM SIGPLAN Notices*, **19**(6), 81-93

Jul, E., Levy, H., Hutchinson, N., and Black, A. (Feb. 1988). Fine-grained mobility in the Emerald system. *ACM Trans. Computer Systems*, **6**(1), 109-133

Kannan, S. and Proebsting, T. A. (Feb. 1994). Correction to 'Producing good code for the case statement'. *Software – Practice and Experience*, **24**(2), 233

Karp, A. H. (May 1987). Programming for parallelism. *IEEE Computer*, **20**(5), 43-57

Kastens, U. (1987). Lifetime analysis for attributes. *Acta Informatica*, **24**, 633-652

Kastens, U., Hutt, B., and Zimmermann, E. (1982). *GAG: A Practical Compiler Generator*. Springer-Verlag

Katayama, T. (July 1984). Translation of attribute grammars into procedures. *ACM Trans. Programming Languages and Systems*, **6**(3), 345-369

Keleher, P., Cox, A. L., Dwarkadas, S., and Zwaenepoel, W. (Jan. 1994). TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Winter 94 USENIX Conference* (Jeffrey Mogul, ed.), pp. 115-132. USENIX Association

Kennedy, K. (1981). A survey of data flow analysis techniques. In *Program Flow Analysis* (Steven S. Muchnick and Neil D. Jones, ed.), pp. 5-54. Prentice Hall

King, J. (July 1976). Symbolic execution and program testing. *Commun. ACM*, **19**(7), 385-394

Klint, P. (Sept. 1981). Interpretation techniques. *Software – Practice and Experience*, **11**(9), 963-973

Knuth, D. E. (1965). On the translation of languages from left to right. *Inform. Control*, **8**, 607-639

Knuth, D. E. (1968). Semantics of context-free languages. *Math. Syst. Theory*, **2**(2), 127-145

Knuth, D. E. (1971). Semantics of context-free languages – correction. *Math. Syst. Theory*, **5**(1), 95-96

Knuth, D. E. (1973). *The Art of Computer Programming* – Vol 1: *Fundamental Algorithms*. 2nd Edn. Addison-Wesley

Knuth, D. E. and Stevenson, F. R. (1973). Optimal measurement points for program frequency counts. *BIT*, **13**, 313-322

Koskimies, K. (June 1991). Object-orientation in attribute grammars. In *Attribute Grammars, Applications and Systems* (H. Alblas & B. Melichar, ed.), pp. 297-329. Springer-Verlag

Kowaltowski, T. (1981). Parameter passing mechanisms and run-time data structures. *Software − Practice and Experience*, **11**(7), 757-765

Kristensen, A. (1998). Template resolution in XML/HTML. *Computer Networks and ISDN Systems*, **30**(1-7), 239-249

Kumar, V., Grama, A., Gupta, A., and Karypis, G. (1994). *Introduction to Parallel Computing − Design and Analysis of Algorithms*. Benjamin/Cummings

Kursawe, P. (1987). How to invent a Prolog machine. *New Generation Computing*, **5**, 97-114

LaLonde, W. R., Lee, E. S., and Horning, J. J. (1971). An LALR(*k*) parser generator. In *IFIP Congress 71* (C.V. Freiman, ed.), pp. 153-157. North-Holland

Landin, P. J. (April 1964). The mechanical evaluation of expressions. *Computer J.*, **6**(4), 308-320

Lemkin, P. F. (Oct. 1988). PSAIL: a portable SAIL to C compiler − description and tutorial. *ACM SIGPLAN Notices*, **23**(10), 149-171

Levine, J. R. (1999). *Linkers and Loaders*. Morgan-Kaufmann

Levine, J. R., Mason, T., and Brown, D. (1992). *Lex and Yacc*. 2nd Edn. O'Reilly

Lewis, H. R. and Papadimitriou, C. H. (1998). *Elements of the Theory of Computation*. Prentice Hall

Lewis II, P. M. and Stearns, R. E. (1968). Syntax-directed transduction. *J. ACM*, **15**(3), 465-488

Li, K. and Hudak, P. (Nov. 1989). Memory coherence in shared virtual memory systems. *ACM Trans. Computer Systems*, **7**(4), 321-359

Linz, P. (1997). *An Introduction to Formal Languages and Automata*. 2nd Edn. Jones and Bartlett

Louden, K. C. (1997). *Compiler Construction − Principles and Practice*. PWS Publishing

Loveman, D. B. (Feb. 1993). High performance Fortran. *IEEE Parallel and Distributed Technology*, **1**(1), 25-42

Maassen, J., Van Nieuwpoort, R., Veldema, R., Bal, H. E., and Plaat, A. (Aug. 1999). An efficient implementation of Java's remote method invocation. *ACM SIGPLAN Notices*, **34**(8), 173-182

Martelli, A. and Montanari, U. (April 1982). An efficient unification algorithm. *ACM Trans. Programming Languages and Systems*, **4**(2), 258-282

Massalin, H. (Oct. 1987). Superoptimizer − A look at the smallest program. *ACM SIGPLAN Notices*, **22**(10), 122-126

McCarthy, J. (April 1960). Recursive functions of symbolic expressions and their computation by machine. *Commun. ACM*, **3**(4), 184-195

McKenzie, B. J., Harries, R., and Bell, T. C. (Feb. 1990). Selecting a hashing algorithm. *Software − Practice and Experience*, **20**(2), 209-224

McKenzie, B. J., Yeatman, C., and De Vere, L. (July 1995). Error repair in shift-reduce parsers. *ACM Trans. Programming Languages and Systems*, **17**(4), 672-689

Milner, R., Tofte, M., Harper, R., and MacQueen, D. (1997). *The Definition of Standard ML*. revised Edn. MIT Press

Morel, E. (1984). Data flow analysis and global optimization. In *Methods and Tools for Compiler Construction* (B. Lorho, ed.), pp. 289-315. Cambridge University Press

Morgan, R. (1998). *Building an Optimizing Compiler*. Digital Press/Butterworth-Heinemann

Muchnick, S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann

Muchnick, S. S. and Jones, N. D. (1981). *Program Flow Analysis*. Prentice Hall

Naur, P. (1965). Checking of operand types in ALGOL compilers. *BIT*, **5**, 151-163

Nguyen, T.-T. and Raschner, E. W. (May 1982). Indirect threaded code used to emulate a virtual machine. *ACM SIGPLAN Notices*, **17**(5), 80-89

Nitzberg, B. and Lo, V. (Aug. 1991). Distributed shared memory − A survey of issues and algorithms. *IEEE Computer*, **24**(8), 52-60

Noonan, R. E. (1985). An algorithm for generating abstract syntax trees. *Computer Languages*, **10**(3/4), 225-236

Nuutila, E. (Nov. 1994). An efficient transitive closure algorithm for cyclic digraphs. *Information Processing Letters*, **52**(4), 207-213

Op den Akker, R. and Sluiman, E. (June 1991). Storage allocation for attribute evaluators using stacks and queues. In *Attribute Grammars, Applications and Systems* (H. Alblas & B. Melichar, ed.), pp. 140-150. Springer-Verlag

Pagan, F. G. (June 1988). Converting interpreters into compilers. *Software − Practice and Experience*, **18**(6), 509-527

Pagan, F. G. (1991). *Partial Computation and the Construction of Language Processors*. Prentice Hall

Pager, D. (1977). The lane-tracing algorithm for constructing LR(k) parsers and ways of enhancing its efficiency. *Inform. Sci.*, **12**, 19-42

Paige, R. and Koenig, S. (July 1982). Finite differencing of computable expressions. *ACM Trans. Programming Languages and Systems*, **4**(3), 402-452

Parr, T. J. and Quong, R. W. (1995). ANTLR: A predicated-LL(k) parser generator. *Software – Practice and Experience*, **25**(7), 789-810

Parsons, T. W. (1992). *Introduction to Compiler Construction*. Computer Science Press

Paulson, L. C. (1996). *ML for the Working Programmer*. 2nd Edn. Cambridge University Press

Pemberton, S. (1980). Comments on an error-recovery scheme by Hartmann. *Software – Practice and Experience*, **10**(3), 231-240

Pettersson, M. (Oct. 1992). A term pattern-match compiler inspired by finite automata theory. In *Compiler Construction, 4th International Conference, CC'92* (U. Kastens and P. Pfahler, ed.), pp. 258-270. Springer-Verlag

Peyton Jones, S. L. (1987). *The Implementation of Functional Programming Languages*. Prentice Hall

Peyton Jones, S. and Hughes, J. (Feb. 1999). Haskell 98: A non-strict, purely functional language. *Technical Report http://www.haskell.org/onlinereport*, Internet

Peyton Jones, S. L. and Lester, D. R. (1992). *Implementing Functional Languages*. Prentice Hall

Plaisted, D. A. (1984). The occur-check problem in Prolog. *New Generation Computing*, **2**, 309-322

Poonen, G. (Aug. 1977). Error recovery for LR(k) parsers. In *Information Processing 77* (Bruce Gilchrist, ed.), pp. 529-533. North Holland

Proebsting, T. A. (May 1995). BURS automata generation. *ACM Trans. Programming Languages and Systems*, **17**(3), 461-486

Purdom Jr., P. (1970). A transitive closure algorithm. *BIT*, **10**, 76-94

Quinn, M. J. (1994). *Parallel Computing – Theory and Practice*. McGraw-Hill

Räihä, K.-J. and Saarinen, M. (1982). Testing attribute grammars for circularity. *Acta Informatica*, **17**, 185-192

Ramalingam, G. and Srinivasan, H. (1997). A member lookup algorithm for C++. *ACM SIGPLAN Notices*, **32**(5), 18-30

Reps, T. (March 1998). Maximal-munch tokenization in linear time. *ACM Trans. Programming Languages and Systems*, **20**(2), 259-273

Révész, G. E. (1985). *Introduction to Formal Languages*. McGraw-Hill

Richter, H. (July 1985). Noncorrecting syntax error recovery. *ACM Trans. Programming Languages and Systems*, **7**(3), 478-489

Ritter, T. and Walker, G. (Sept. 1980). Varieties of threaded code for language implementation. *BYTE*, **5**(9), 206-227

Robinson, J. A (Jan. 1965). A machine-oriented logic based on the resolution principle. *J. ACM*, **12**(1), 23-41

Robinson, J. A. (1971). Computational logic: The unification computation. *Mach. Intell.*, **6**, 63-72

Röhrich, J. (Feb. 1980). Methods for the automatic construction of error correcting parsers. *Acta Informatica*, **13**(2), 115-139

Sale, A. (Sept. 1981). The implementation of case statements in Pascal. *Software – Practice and Experience*, **11**(9), 929-942

Saloman, D. (1992). *Assemblers and Loaders*. Ellis Horwood

Samelson, K. and Bauer, F. L. (Feb. 1960). Sequential formula translation. *Commun. ACM*, **3**(2), 76-83

Sankaran, N. (Sept. 1994). A bibliography on garbage collection and related topics. *ACM SIGPLAN Notices*, **29**(9), 149-158

Sassa, M. and Goto, E. (1976). A hashing method for fast set operations. *Information Processing Letters*, **5**(2), 31-34

Schnorr, C. P. (May 1978). An algorithm for transitive closure with linear expected time. *SIAM J. Comput.*, **7**(2), 127-133

Schorr, H. and Waite, W. M. (Aug. 1967). An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM*, **10**(8), 501-506

Sedgewick, R. (1988). *Algorithms*. Addison-Wesley

Sethi, R. and Ullman, J. D. (Oct. 1970). The generation of optimal code for arithmetic expressions. *J. ACM*, **17**(4), 715-728

Sheridan, P. B. (Feb. 1959). The arithmetic translator-compiler of the IBM FORTRAN automatic coding system. *Commun. ACM*, **2**(2), 9-21

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

Sippu, S. and Soisalon-Soininen, E. (1988/1990). *Parsing Theory,* Vol. I: *Languages and Parsing;* Vol. II: *LL(k) and LR(k) Parsing*. Springer-Verlag

Skillicorn, D. B. and Talia, D. (June 1998). Models and languages for parallel computation. *ACM Computing Surveys*, **30**(2), 123-169

Stirling, C. P. (March 1985). Follow set error recovery. *Software − Practice and Experience*, **15**(3), 239-257

Stumm, M. and Zhou, S. (May 1990). Algorithms implementing distributed shared memory. *IEEE Computer*, **23**(5), 54-64

Sudkamp, T. A. (1997). *Languages and Machines − An Introduction to the Theory of Computer Science*. 2nd Edn. Addison-Wesley

Sysło, M. M. and Dzikiewicz, J. (1975). Computational experiences with some transitive closure algorithms. *Computing*, **15**, 33-39

Tanenbaum, A. S. (1995). *Distributed Operating Systems*. Prentice Hall

Tanenbaum, A. S., Van Staveren, H., Keizer, E. G., and Stevenson, J. W. (Sept. 1983). A practical toolkit for making portable compilers. *Commun. ACM*, **26**(9), 654-660

Tanenbaum, A. S., Van Staveren, H., and Stevenson, J. W. (Jan. 1982). Using peephole optimization on intermediate code. *ACM Trans. Programming Languages and Systems*, **4**(1), 21-36

Tarditi, D. R., Lee, P., and Acharya, A. (June 1992). No assembly required: compiling Standard ML to C. *ACM Letters on Programming Languages and Systems*, **1**(2), 161-177

Tarjan, R. E. (April 1975). Efficiency of a good but not linear set merging algorithm. *J. ACM*, **22**(2), 215-225

Templ, J. (April 1993). A systematic approach to multiple inheritance. *ACM SIGPLAN Notices*, **28**(4), 61-66

Terry, P. D. (1997). *Compilers and Compiler Generators − An Introduction Using C++*. International Thomson

Thompson, K. (June 1968). Regular expression search algorithm. *Commun. ACM*, **11**(6), 419-422

Thompson, S. (1999). *Haskell: The Craft of Functional Programming*. 2nd Edn. Addison-Wesley

Turner, D. A. (1979). A new implementation technique for applicative languages. *Software − Practice and Experience*, **9**, 31-49

Uhl, J. S. and Horspool, R. N. (1994). Flow grammars − a flow analysis methodology. In *Compiler Construction: 5th International Conference, CC '94* (Peter A. Fritzson, ed.), pp. 203-217. Springer-Verlag

Ungar, D. M. (May 1984). Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, **19**(5), 157-167

Van Roy, P. (1994). 1983-1993: The wonder years of sequential Prolog implementation. *J. Logic Programming*, **19-20**, 385-441

Van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., Koster, C. H. A., Sintzoff, M., Lindsey, C. H., Meertens, L. G. L. T., and Fisker, R. G. (1975). Revised report on the algorithmic language Algol 68. *Acta Informatica*, **5**, 1-236

Verbrugge, C., Co, P., and Hendren, L. (1996). Generalized constant propagation − a study in C. In *Compiler Construction: 6th International Conference, CC'96* (Tibor Gyimóthy, ed.), pp. 74-90. Springer-Verlag

Vitek, J. and Horspool, R. N. (1996). Compact dispatch tables for dynamically typed object-oriented languages. In *Compiler Construction: 6th International Conference, CC'96* (Tibor Gyimóthy, ed.), pp. 309-325. Springer-Verlag

Waddle, V. E. (Jan. 1990). Production trees: a compact representation of parsed programs. *ACM Trans. Programming Languages and Systems*, **12**(1), 61-83

Warren, D. H. D. (Oct. 1983). An Abstract Prolog Instruction Set. *Technical Report Technical Note 309*, Artificial Intelligence Center, SRI

Warshall, S. (1962). A theorem on Boolean matrices. *J. ACM*, **9**, 11-12

Wegbreit, B. (Sept. 1975). Property extraction in well-founded property sets. *IEEE Trans. Software Engineering*, **SE-1**(3), 270-285

Wendt, A. L. (June 1990). Fast code generation using automatically generated decision trees. *ACM SIGPLAN Notices*, **25**(6), 9-15

Wentworth, E. P. (July 1990). Pitfalls of conservative garbage collection. *Software − Practice and Experience*, **20**(7), 719-727

Wichmann, B. A. (May-June 1977). How to call procedures, or second thoughts on Ackermann's function. *Software − Practice and Experience*, **7**(3), 317-329

Wilhelm, R. and Maurer, D. (1995). *Compiler Design*. Addison-Wesley
Wilson, G. V. (1995). *Practical Parallel Programming*. MIT Press
Wilson, G. V. and Lu, P. (1996). *Parallel Programming Using C++*. MIT Press
Wolfe, M. J. (1996). *High Performance Compilers for Parallel Computing*. Addison-Wesley
Yershov, A. P. (1971). *The Alpha Automatic Programming System*. Academic Press
Yuval, G. (July-Aug. 1977). The utility of the CDC 6000 registers. *Software – Practice and Experience*, **7**(4), 535-536

Index