1

Introduction

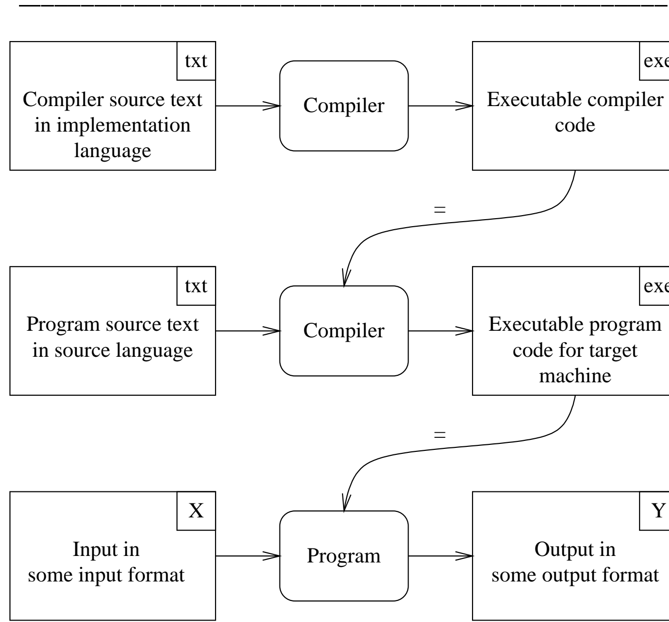


Figure 1.1 Compiling and running a compiler.

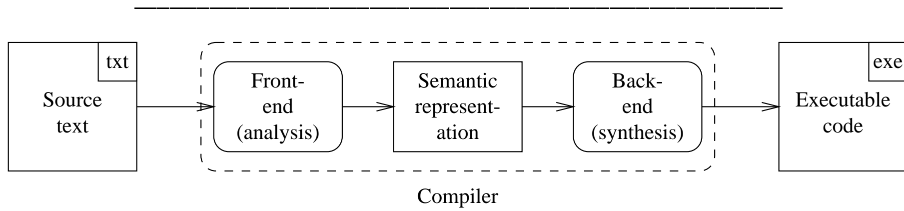


Figure 1.2 Conceptual structure of a compiler.

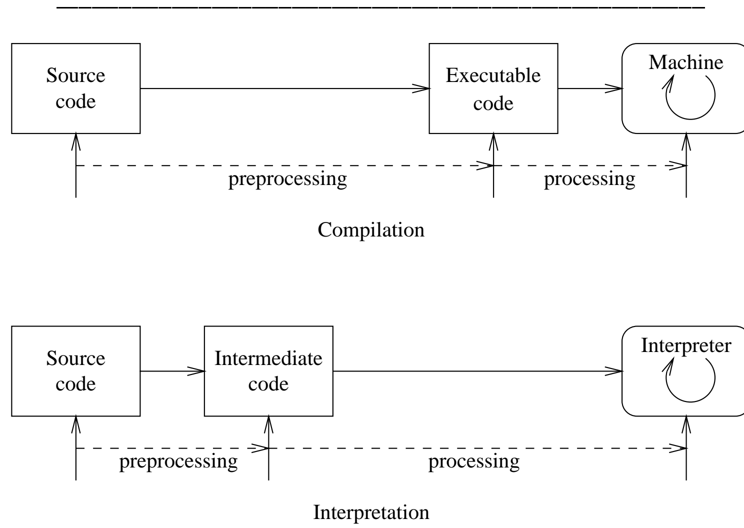


Figure 1.3 Comparison of a compiler and an interpreter.

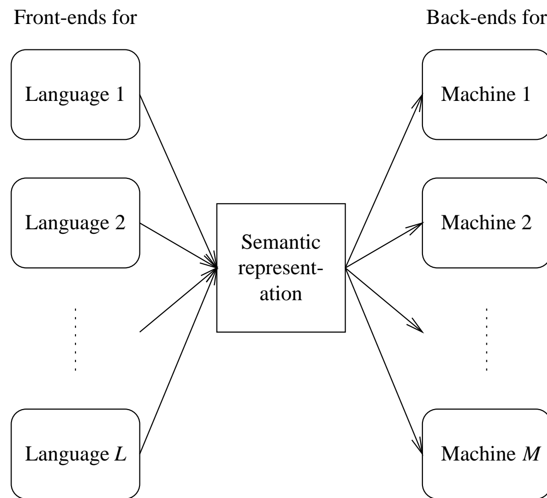


Figure 1.4 Creating compilers for L languages and M machines.

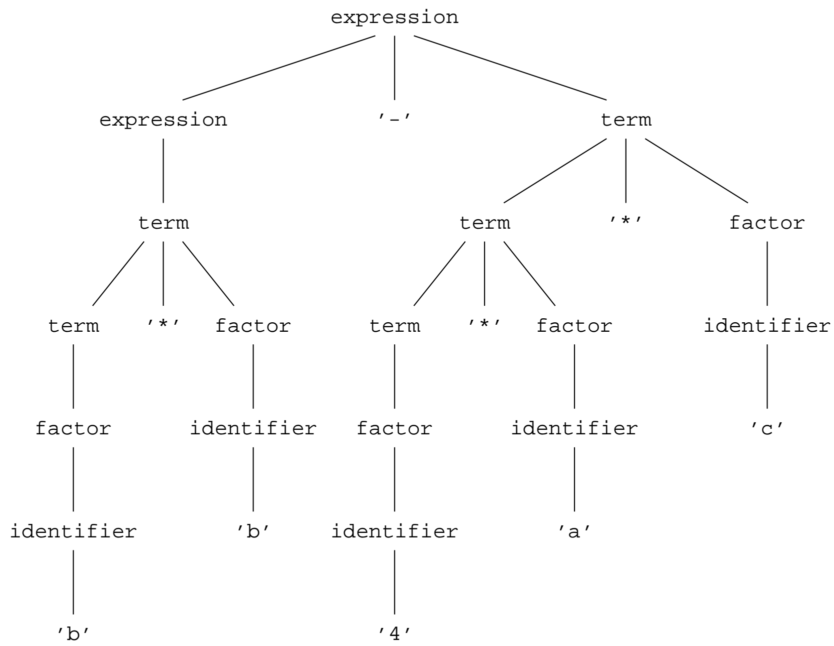


Figure 1.5 The expression `b*b - 4*a*c` as a parse tree.

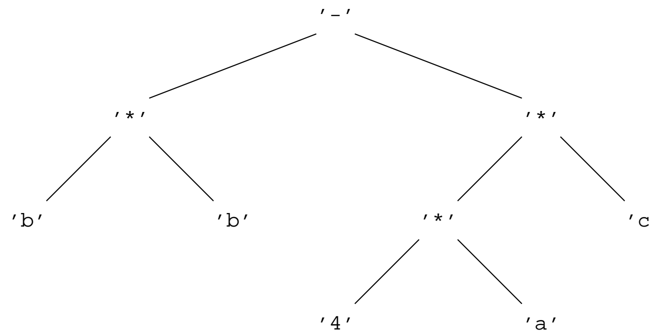


Figure 1.6 The expression `b*b - 4*a*c` as an AST.

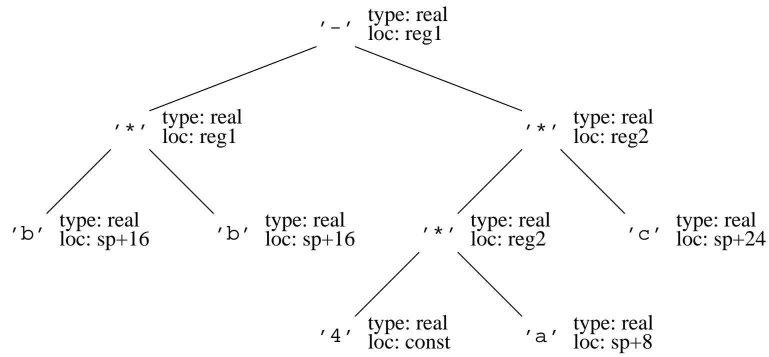


Figure 1.7 The expression `b*b - 4*a*c` as an annotated AST.

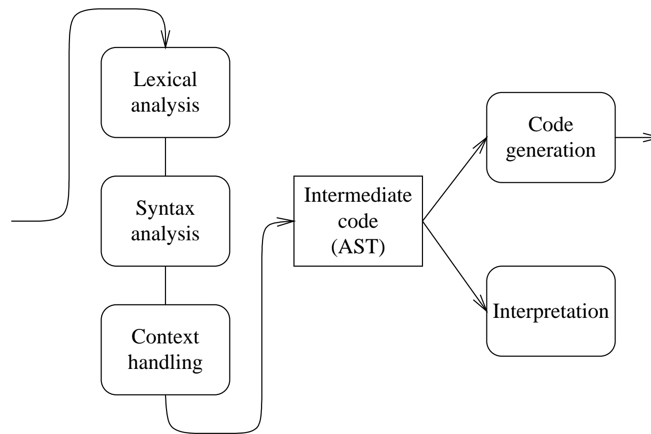


Figure 1.8 Structure of the demo compiler/interpreter.

```
expression → digit | '(' expression operator expression ')'
operator → '+' | '*'
digit → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Figure 1.9 Grammar for simple fully parenthesized expressions.

```
#include    "parser.h"      /* for type AST_node */
#include    "backend.h"     /* for Process() */
#include    "error.h"       /* for Error() */

int main(void) {
    AST_node *icode;

    if (!Parse_program(&icode)) Error("No top-level expression");
    Process(icode);

    return 0;
}
```

Figure 1.10 Driver for the demo compiler.

```
/* Define class constants */
/* Values 0-255 are reserved for ASCII characters */
#define    EOF      256
#define    DIGIT    257

typedef struct {int class; char repr;} Token_type;

extern Token_type Token;
extern void get_next_token(void);
```

Figure 1.11 Header file `lex.h` for the demo lexical analyzer.

```
#include    "lex.h"         /* for self check */
                                        /* PRIVATE */
static int Layout_char(int ch) {
    switch (ch) {
    case ' ': case '\t': case '\n': return 1;
    default:                        return 0;
    }
}
                                        /* PUBLIC */
Token_type Token;

void get_next_token(void) {
    int ch;

    /* get a non-layout character: */
    do {
        ch = getchar();
        if (ch < 0) {
            Token.class = EOF; Token.repr = '#';
            return;
        }
    } while (Layout_char(ch));

    /* classify it: */
    if ('0' <= ch && ch <= '9') {Token.class = DIGIT;}
    else {Token.class = ch;}

    Token.repr = ch;
}
```

Figure 1.12 Lexical analyzer for the demo compiler.

```
static int Parse_operator(Operator *oper) {
    if (Token.class == '+') {
        *oper = '+'; get_next_token(); return 1;
    }
    if (Token.class == '*') {
        *oper = '*'; get_next_token(); return 1;
    }
    return 0;
}

static int Parse_expression(Expression **expr_p) {
    Expression *expr = *expr_p = new_expression();

    /* try to parse a digit: */
    if (Token.class == DIGIT) {
        expr->type = 'D'; expr->value = Token.repr - '0';
        get_next_token();
        return 1;
    }

    /* try to parse a parenthesized expression: */
    if (Token.class == '(') {
        expr->type = 'P';
        get_next_token();
        if (!Parse_expression(&expr->left)) {
            Error("Missing expression");
        }
        if (!Parse_operator(&expr->oper)) {
            Error("Missing operator");
        }
        if (!Parse_expression(&expr->right)) {
            Error("Missing expression");
        }
        if (Token.class != ')') {
            Error("Missing )");
        }
        get_next_token();
        return 1;
    }

    /* failed on both attempts */
    free_expression(expr); return 0;
}
```

Figure 1.13 Parsing routines for the demo compiler.

```
#include    "lex.h"
#include    "error.h"       /* for Error() */
#include    "parser.h"      /* for self check */
                                        /* PRIVATE */
static Expression *new_expression(void) {
    return (Expression *)malloc(sizeof (Expression));
}
static void free_expression(Expression *expr) {free((void *)expr);}
static int Parse_operator(Operator *oper_p);
static int Parse_expression(Expression **expr_p);

                                        /* PUBLIC */
int Parse_program(AST_node **icode_p) {
    Expression *expr;

    get_next_token();           /* start the lexical analyzer */
    if (Parse_expression(&expr)) {
        if (Token.class != EOF) {
            Error("Garbage after end of program");
        }
        *icode_p = expr;
        return 1;
    }
    return 0;
}
```

Figure 1.14 Parser environment for the demo compiler.

```
int P(...) {
    /* try to parse the alternative A1 A2 ... An */
    if (A1(...)) {
        if (!A2(...)) Error("Missing A2");
        ...
        if (!An(...)) Error("Missing An");
        return 1;
    }
    /* try to parse the alternative B1 B2 ... */
    if (B1(...)) {
        if (!B2(...)) Error("Missing B2");
        ...
        return 1;
    }
    ...
    /* failed to find any alternative of P */
    return 0;
}
```

Figure 1.15 A C template for a grammar rule.

```
typedef int Operator;

typedef struct _expression {
    char type;                          /* 'D' or 'P' */
    int value;                          /* for 'D' */
    struct _expression *left, *right;   /* for 'P' */
    Operator oper;                      /* for 'P' */
} Expression;

typedef Expression AST_node;            /* the top node is an Expression */

extern int Parse_program(AST_node **);
```

Figure 1.16 Parser header file for the demo compiler.

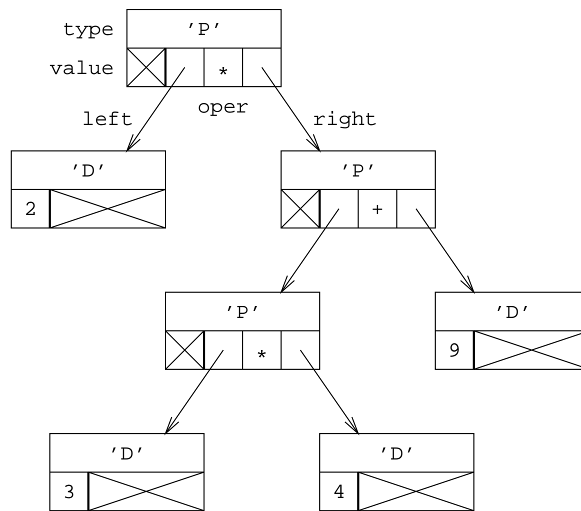


Figure 1.17 An AST for the expression `(2*((3*4)+9))`.

```
#include    "parser.h"      /* for types AST_node and Expression */
#include    "backend.h"     /* for self check */
                                        /* PRIVATE */
static void Code_gen_expression(Expression *expr) {
    switch (expr->type) {
    case 'D':
        printf("PUSH %d\n", expr->value);
        break;
    case 'P':
        Code_gen_expression(expr->left);
        Code_gen_expression(expr->right);
        switch (expr->oper) {
        case '+': printf("ADD\n"); break;
        case '*': printf("MULT\n"); break;
        }
        break;
    }
}
                                        /* PUBLIC */
void Process(AST_node *icode) {
    Code_gen_expression(icode); printf("PRINT\n");
}
```

Figure 1.18 Code generation back-end for the demo compiler.

```
#include    "parser.h"      /* for types AST_node and Expression */
#include    "backend.h"     /* for self check */
                                        /* PRIVATE */
static int Interpret_expression(Expression *expr) {
    switch (expr->type) {
    case 'D':
        return expr->value;
        break;
    case 'P': {
        int e_left = Interpret_expression(expr->left);
        int e_right = Interpret_expression(expr->right);
        switch (expr->oper) {
        case '+': return e_left + e_right;
        case '*': return e_left * e_right;
        }}
        break;
    }
}
                                        /* PUBLIC */
void Process(AST_node *icode) {
    printf("%d\n", Interpret_expression(icode));
}
```

Figure 1.19 Interpreter back-end for the demo compiler.

```
extern void Process(AST_node *);
```

Figure 1.20 Common back-end header for code generator and interpreter.

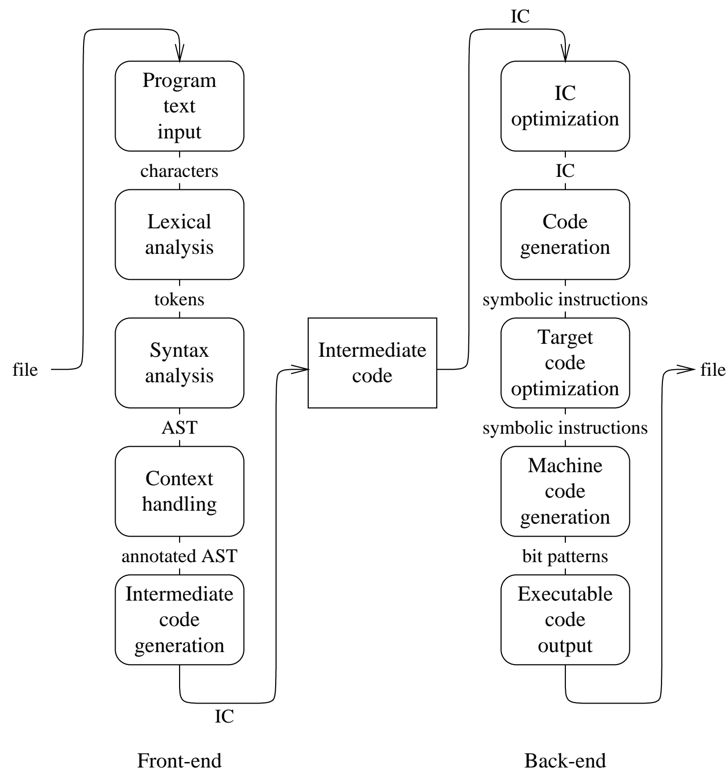


Figure 1.21 Structure of a compiler.

```
SET Object code TO
    Assembly(
        Code generation(
            Context check(
                Parse(
                    Tokenize(
                        Source code
                    )
                )
            )
        )
    );
```

Figure 1.22 Flow-of-control structure of a broad compiler.

```
WHILE NOT Finished:
    Read some data D from the source code;
    Process D and produce the corresponding object code, if any;
```

Figure 1.23 Flow-of-control structure of a narrow compiler.

```
WHILE Obtained input character Ch from previous module:
    IF Ch = 'a':
        // See if there is another 'a':
        IF Obtained input character Ch1 from previous module:
            IF Ch1 = 'a':
                // We have 'aa':
                Output character 'b' to next module;
            ELSE Ch1 /= 'a':
                Output character 'a' to next module;
                Output character Ch1 to next module;
        ELSE Ch1 not obtained:
            Output character 'a' to next module;
            EXIT WHILE;
    ELSE Ch /= 'a':
        Output character Ch to next module;
```

Figure 1.24 The filter aa → b as a main loop.

```
SET the flag Input exhausted TO False;
SET the flag There is a stored character TO False;
SET Stored character TO Undefined;     // can never be an 'a'

FUNCTION Filtered character RETURNING a Boolean, a character:
    IF Input Exhausted: RETURN False, No character;
    ELSE IF There is a stored character:
        // It cannot be an 'a':
        SET There is a stored character TO False;
        RETURN True, Stored character;
    ELSE Input not exhausted AND There is no stored character:
        IF Obtained input character Ch from previous module:
            IF Ch = 'a':
                // See if there is another 'a':
                IF Obtained input character Ch1 from previous module:
                    IF Ch1 = 'a':
                        // We have 'aa':
                        RETURN True, 'b';
                    ELSE Ch1 /= 'a':
                        SET Stored character TO Ch1;
                        SET There is a stored character TO True;
                        RETURN True, 'a';
                ELSE Ch1 not obtained:
                    SET Input exhausted TO True;
                    RETURN True, 'a';
            ELSE Ch /= 'a':
                RETURN True, Ch;
        ELSE Ch not obtained:
            SET Input exhausted TO True;
            RETURN False, No character;
```

Figure 1.25 The filter aa → b as a pre-main module.

```
        expression
1@1     '(' expression operator expression ')'
2@2     '(' '1' operator expression ')'
4@3     '(' '1' '*' expression ')'
1@4     '(' '1' '*' '(' expression operator expression ')' ')'
2@5     '(' '1' '*' '(' '1' operator expression ')' ')'
3@6     '(' '1' '*' '(' '1' '+' expression ')' ')'
2@7     '(' '1' '*' '(' '1' '+' '1' ')' ')'
```

Figure 1.26 Leftmost derivation of the string `(1*(1+1))`.

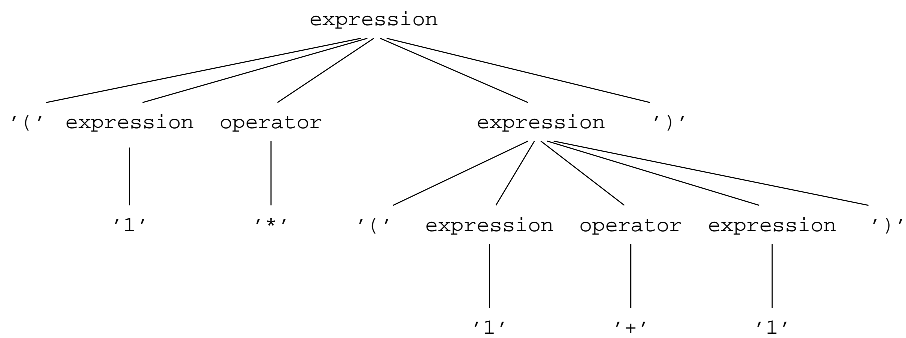


Figure 1.27 Parse tree of the derivation in Figure 1.26.

```
void P() { ... Q(); .... S(); ... }
void Q() { ... R(); .... T(); ... }
void R() { ... P(); }
void S() { ... }
void T() { ... }
```

Figure 1.28 Sample C program used in the construction of a calling graph.

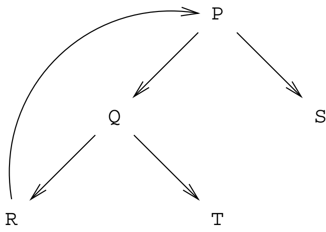


Figure 1.29 Initial (direct) calling graph of the code in Figure 1.28.

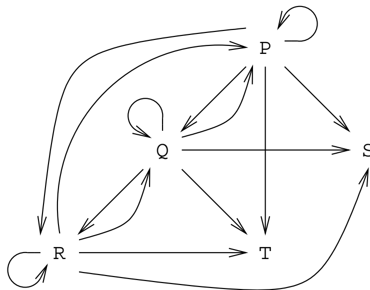


Figure 1.30 Calling graph of the code in Figure 1.28.

Data definitions:

1. Let G be a directed graph with one node for each routine. The information items are arrows in G.
2. An arrow from a node A to a node B means that routine A calls routine B directly or indirectly.

Initializations:

If the body of a routine A contains a call to routine B, an arrow from A to B must be present.

Inference rules:

If there is an arrow from node A to node B and one from B to C, an arrow from A to C must be present.

Figure 1.31 Recursion detection as a closure algorithm.

```
calls(A, C) :- calls(A, B), calls(B, C).
calls(a, b).
calls(b, a).
:-? calls(a, a).
```

Figure 1.32 A Prolog program corresponding to the closure algorithm of Figure 1.31.

```
SET the flag Something changed TO True;

WHILE Something changed:
    SET Something changed TO False;

    FOR EACH Node 1 IN Graph:
        FOR EACH Node 2 IN Descendants of Node 1:
            FOR EACH Node 3 IN Descendants of Node 2:
                IF there is no arrow from Node 1 to Node 3:
                    Add an arrow from Node 1 to Node 3;
                    SET Something changed TO True;
```

Figure 1.33 Outline of a bottom-up algorithm for transitive closure.

2

From program text to abstract syntax tree

```
expression → product | factor
product → expression '*' factor
factor → number | identifier
```

Figure 2.1 A very simple grammar for `expression`.

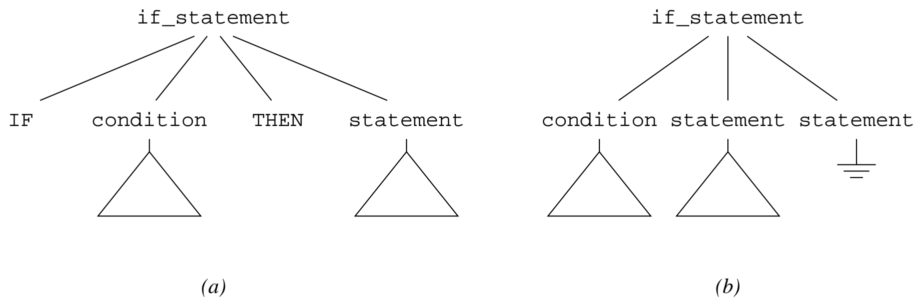


Figure 2.2 Syntax tree (a) and abstract syntax tree (b) of an if-then statement.

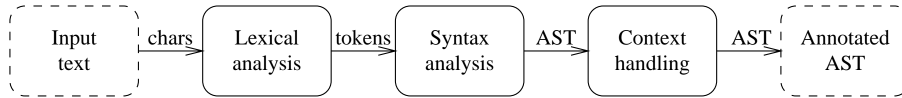


Figure 2.3 Pipeline from input to annotated syntax tree.

Basic patterns:	*Matching:*
x	The character x
.	Any character, usually except a newline
[xyz...]	Any of the characters x, y, z, ...
Repetition operators:	
$R^?$	An R or nothing (= optionally an R)
R^*	Zero or more occurrences of R
R^+	One or more occurrences of R
Composition operators:	
R_1R_2	An R_1 followed by an R_2
$R_1 \vert R_2$	Either an R_1 or an R_2
Grouping:	
(R)	R itself

Figure 2.4 Components of regular expressions.

```
/* Define class constants; 0-255 reserved for ASCII characters: */
#define EOF             256
#define IDENTIFIER      257
#define INTEGER         258
#define ERRONEOUS       259

typedef struct {
    char *file_name;
    int line_number;
    int char_number;
} Position_in_File;

typedef struct {
    int class;
    char *repr;
    Position_in_File pos;
} Token_Type;

extern Token_Type Token;

extern void start_lex(void);
extern void get_next_token(void);
```

Figure 2.5 Header file `lex.h` of the handwritten lexical analyzer.

```
#include    "input.h"           /* for get_input() */
#include    "lex.h"

/* PRIVATE */
static char *input;
static int dot;                 /* dot position in input */
static int input_char;          /* character at dot position */

#define next_char()             (input_char = input[++dot])

/* PUBLIC */
Token_Type Token;

void start_lex(void) {
    input = get_input();
    dot = 0; input_char = input[dot];
}
```

Figure 2.6 Data and start-up of the handwritten lexical analyzer.

```
void get_next_token(void) {
    int start_dot;

    skip_layout_and_comment();
    /* now we are at the start of a token or at end-of-file, so: */
    note_token_position();

    /* split on first character of the token */
    start_dot = dot;
    if (is_end_of_input(input_char)) {
        Token.class = EOF; Token.repr = "<EOF>"; return;
    }
    if (is_letter(input_char)) {recognize_identifier();}
    else
    if (is_digit(input_char)) {recognize_integer();}
    else
    if (is_operator(input_char) || is_separator(input_char)) {
        Token.class = input_char; next_char();
    }
    else {Token.class = ERRONEOUS; next_char();}

    Token.repr = input_to_zstring(start_dot, dot-start_dot);
}
```

Figure 2.7 Main reading routine of the handwritten lexical analyzer.

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

Figure 2.8 Skipping layout and comment in the handwritten lexical analyzer.

```
void recognize_identifier(void) {
    Token.class = IDENTIFIER; next_char();

    while (is_letter_or_digit(input_char)) {next_char();}

    while (is_underscore(input_char)
    &&     is_letter_or_digit(input[dot+1])
    ) {
        next_char();
        while (is_letter_or_digit(input_char)) {next_char();}
    }
}
```

Figure 2.9 Recognizing an identifier in the handwritten lexical analyzer.

```
void recognize_integer(void) {
    Token.class = INTEGER; next_char();
    while (is_digit(input_char)) {next_char();}
}
```

Figure 2.10 Recognizing an integer in the handwritten lexical analyzer.

```
#define is_end_of_input(ch)     ((ch) == '\0')
#define is_layout(ch)           (!is_end_of_input(ch) && (ch) <= ' ')
#define is_comment_starter(ch)  ((ch) == '#')
#define is_comment_stopper(ch)  ((ch) == '#' || (ch) == '\n')

#define is_uc_letter(ch)        ('A' <= (ch) && (ch) <= 'Z')
#define is_lc_letter(ch)        ('a' <= (ch) && (ch) <= 'z')
#define is_letter(ch)           (is_uc_letter(ch) || is_lc_letter(ch))
#define is_digit(ch)            ('0' <= (ch) && (ch) <= '9')
#define is_letter_or_digit(ch)  (is_letter(ch) || is_digit(ch))
#define is_underscore(ch)       ((ch) == '_')

#define is_operator(ch)         (strchr("+-*/", (ch)) != 0)
#define is_separator(ch)        (strchr(";,(){}", (ch)) != 0)
```

Figure 2.11 Character classification in the handwritten lexical analyzer.

```
#include    "lex.h"      /* for start_lex(), get_next_token() */

int main(void) {
    start_lex();

    do {
        get_next_token();
        switch (Token.class) {
        case IDENTIFIER:    printf("Identifier"); break;
        case INTEGER:       printf("Integer"); break;
        case ERRONEOUS:     printf("Erroneous token"); break;
        case EOF:           printf("End-of-file pseudo-token"); break;
        default:            printf("Operator or separator"); break;
        }
        printf(": %s\n", Token.repr);
    } while (Token.class != EOF);
    return 0;
}
```

Figure 2.12 Driver for the handwritten lexical analyzer.

```
Integer: 8
Operator or separator: ;
Identifier: abc
Erroneous token: _
Erroneous token: _
Identifier: dd_8
Operator or separator: ;
Identifier: zz
Erroneous token: _
End-of-file pseudo-token: <EOF>
```

Figure 2.13 Sample results of the hand-written lexical analyzer.

```
#define is_operator(ch)         (is_operator_bit[(ch)&0377])

static const char is_operator_bit[256] = {
    0,          /* position 0 */
    0, 0, ...   /* another 41 zeroes */
    1,          /* '*', position 42 */
    1,          /* '+' */
    0,
    1,          /* '-' */
    0,
    1,          /* '/', position 47 */
    0, 0, ...   /* 208 more zeroes */
};
```

Figure 2.14 A naive table implementation of `is_operator()`.

```
#define UC_LETTER_MASK      (1<<1)  /* a 1 bit, shifted left 1 pos. */
#define LC_LETTER_MASK      (1<<2)  /* a 1 bit, shifted left 2 pos. */
#define OPERATOR_MASK       (1<<5)

#define LETTER_MASK         (UC_LETTER_MASK | LC_LETTER_MASK)

#define bits_of(ch)         (charbits[(ch)&0377])

#define is_end_of_input(ch) ((ch) == '\0')

#define is_uc_letter(ch)    (bits_of(ch) & UC_LETTER_MASK)
#define is_lc_letter(ch)    (bits_of(ch) & LC_LETTER_MASK)
#define is_letter(ch)       (bits_of(ch) & LETTER_MASK)

#define is_operator(ch)     (bits_of(ch) & OPERATOR_MASK)

static const char charbits[256] = {
    0000,       /* position 0 */
    ...
    0040,       /* '*', position 42 */
    0040,       /* '+' */
    ...
    0000,       /* position 64 */
    0002,       /* 'A' */
    0002,       /* 'B' */

    0000,       /* position 96 */
    0004,       /* 'a' */
    0004,       /* 'b' */
    ...
    0000        /* position 255 */
};
```

Figure 2.15 Efficient classification of characters (excerpt).

```
SET the global token (Token .class, Token .length) TO (0, 0);

// Try to match token description T1→R1:
FOR EACH Length SUCH THAT the input matches T1→R1 over Length:
    IF Length > Token .length:
        SET (Token .class, Token .length) TO (T1, Length);

// Try to match token description T2→R2:
FOR EACH Length SUCH THAT the input matches T2→R2 over Length:
    IF Length > Token .length:
        SET (Token .class, Token .length) TO (T2, Length);

...

FOR EACH Length SUCH THAT the input matches Tn→Rn over Length:
    IF Length > Token .length:
        SET (Token .class, Token .length) TO (Tn, Length);

IF Token .length = 0:
    Handle non-matching character;
```

Figure 2.16 Outline of a naive generated lexical analyzer.

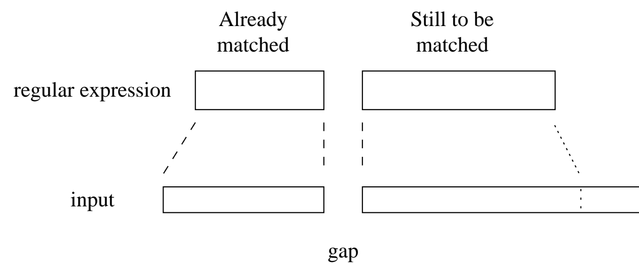


Figure 2.17 Components of a token description and components of the input.

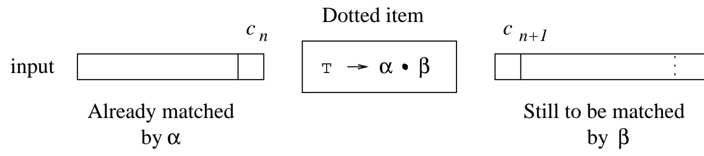


Figure 2.18 The relation between a dotted item and the input.

$$T \rightarrow \alpha \bullet (R)^* \beta \quad \Rightarrow \quad T \rightarrow \alpha (R)^* \bullet \beta, \quad T \rightarrow \alpha (\bullet R)^* \beta$$

$$T \rightarrow \alpha (R \bullet)^* \beta \quad \Rightarrow \quad T \rightarrow \alpha (R)^* \bullet \beta, \quad T \rightarrow \alpha (\bullet R)^* \beta$$

$$T \rightarrow \alpha \bullet (R)^+ \beta \quad \Rightarrow \quad T \rightarrow \alpha (\bullet R)^+ \beta$$

$$T \rightarrow \alpha (R \bullet)^+ \beta \quad \Rightarrow \quad T \rightarrow \alpha (R)^+ \bullet \beta, \quad T \rightarrow \alpha (\bullet R)^+ \beta$$

$$T \rightarrow \alpha \bullet (R)^? \beta \quad \Rightarrow \quad T \rightarrow \alpha (R)^? \bullet \beta, \quad T \rightarrow \alpha (\bullet R)^? \beta$$

$$T \rightarrow \alpha (R \bullet)^? \beta \quad \Rightarrow \quad T \rightarrow \alpha (R)^? \bullet \beta$$

$$T \rightarrow \alpha \bullet (R_1 | R_2 | ...) \beta \quad \Rightarrow \quad T \rightarrow \alpha (\bullet R_1 | R_2 | ...) \beta, \quad T \rightarrow \alpha (R_1 | \bullet R_2 | ...) \beta, \quad ...$$

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

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

$$... \quad ... \quad ...$$

Figure 2.19 ε move rules for the regular operators.

```
integral_number → [0-9]+
fixed_point_number → [0-9]*'.'[0-9]+
```

Figure 2.20 A simple set of regular expressions.

```
IMPORT Input char [1..];        // as from the previous module
SET Read index TO 1;            // the read index into Input char [ ]

PROCEDURE Get next token:
    SET Start of token TO Read index;
    SET End of last token TO Uninitialized;
    SET Class of last token TO Uninitialized;

    SET Item set TO Initial item set ();
    WHILE Item set /= Empty:
        SET Ch TO Input char [Read index];
        SET Item set TO Next item set (Item set, Ch);
        SET Class TO Class of token recognized in (Item set);
        IF Class /= No class:
            SET Class of last token TO Class;
            SET End of last token TO Read index;
        SET Read index TO Read index + 1;

    SET Token .class TO Class of last token;
    SET Token .repr TO Input char [Start of token .. End of last token];
    SET Read index TO End of last token + 1;
```

Figure 2.21 Outline of a linear-time lexical analyzer.

```
FUNCTION Initial item set RETURNING an item set:
    SET New item set TO Empty;

    // Initial contents - obtain from the language specification:
    FOR EACH token description T→R IN the language specification:
        SET New item set TO New item set + item T→•R;

    RETURN ε closure (New item set);
```

Figure 2.22 The function `Initial item set` for a lexical analyzer.

```
FUNCTION Next item set (Item set, Ch) RETURNING an item set:
    SET New item set TO Empty;

    // Initial contents - obtain from character moves:
    FOR EACH item T→α•Bβ IN Item set:
        IF B is a basic pattern AND B matches Ch:
            SET New item set TO New item set + item T→αB•β;

    RETURN ε closure (New item set);
```

Figure 2.23 The function `Next item set()` for a lexical analyzer.

```
FUNCTION ε closure (Item set) RETURNING an item set:
    SET Closure set TO the Closure set produced by the
        closure algorithm 2.25, passing the Item set to it;

    // Filter out the interesting items:
    SET New item set TO Empty set;
    FOR EACH item I IN Closure set:
        IF I is a basic item:
            Insert I in New item set;

    RETURN New item set;
```

Figure 2.24 The function ε `closure()` for a lexical analyzer.

Data definitions:
Let `Closure set` be a set of dotted items.

Initializations:
Put each item in `Item set` in `Closure set`.

Inference rules:
If an item in `Closure set` matches the left-hand side of one of the ε moves in Figure 2.19, the corresponding right-hand side must be present in `Closure set`.

Figure 2.25 Closure algorithm for dotted items.

Data definitions:
1a. A ‘state’ is a set of items.
1b. Let `States` be a set of states.
2a. A ‘state transition’ is a triple (start state, character, end state).
2b. Let `Transitions` be a set of state transitions.

Initializations:
1. Set `States` to contain a single state, `Initial item set()`.
2. Set `Transitions` to the empty set.

Inference rules:
If `States` contains a state `S`, `States` must contain the state `E` and `Transitions` must contain the state transition (`S`, `Ch`, `E`) for each character `Ch` in the input character set, where `E = Next item set (S, Ch)`.

Figure 2.26 The subset algorithm for lexical analyzers.

State	Next state [ ]			Class of token recognized in[ ]
	Ch			
	digit	point	other	
S_0	S_1	S_2	-	-
S_1	S_1	S_2	-	integral_number
S_2	S_3	-	-	-
S_3	S_3	-	-	fixed_point_number

Figure 2.27 Transition table and recognition table for the regular expressions from Figure 2.20.

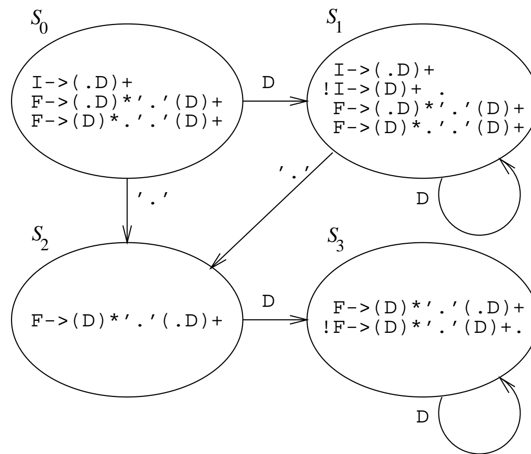


Figure 2.28 Transition diagram of the states and transitions for Figure 2.20.

```
PROCEDURE Get next token:
    SET Start of token TO Read index;
    SET End of last token TO Uninitialized;
    SET Class of last token TO Uninitialized;

    SET Item set TO Initial item set;
    WHILE Item set /= Empty:
        SET Ch TO Input char [Read index];
        SET Item set TO Next item set [Item set, Ch];
        SET Class TO Class of token recognized in [Item set];
        IF Class /= No class:
            SET Class of last token TO Class;
            SET End of last token TO Read index;
        SET Read index TO Read index + 1;

    SET Token .class TO Class of last token;
    SET Token .repr TO Input char [Start of token .. End of last token];
    SET Read index TO End of last token + 1;
```

Figure 2.29 Outline of an efficient linear-time routine `Get next token()`.

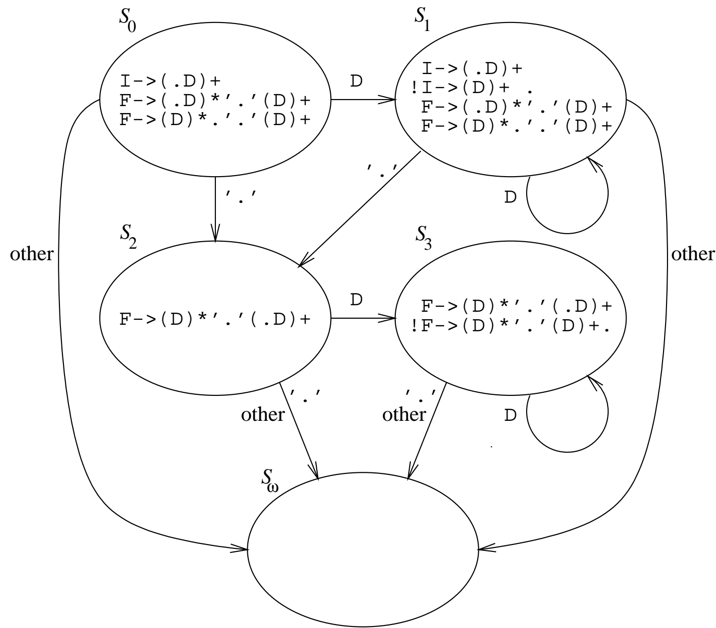


Figure 2.30 Transition diagram of *all* states and transitions for Figure 2.20.

state	digit=1	other=2	point=3
0	1	-	2
1	1	-	2
2	3	-	-
3	3	-	-

Figure 2.31 The transition matrix from Figure 2.27 in reduced form.

```
0     1  -  2
1     1  -  2
2        3  -  -
3        3  -  -
      ----------
      1  3  2  -
```

Figure 2.32 Fitting the strips into one array.

```
Empty state [0..3][1..3] =
    ((0, 1, 0), (0, 1, 0), (0, 1, 1), (0, 1, 1));
Displacement [0..3] = (0, 0, 1, 1);
Entry [1..3] = (1, 3, 2);
```

Figure 2.33 The transition matrix from Figure 2.27 in compressed form.

```
IF Empty state [State][Ch]:
    SET New state TO No state;
ELSE entry in Entry [ ] is valid:
    SET New state TO Entry [Displacement [State] + Ch];
```

Figure 2.34 Code for `SET New state TO Next state[State, Ch]`.

```
0  (1, 1)     -     (2, 3)
1          (1, 1)     -     (2, 3)
2                                  (3, 1)     -     -
3                                          (3, 1)   -   -
   ---------------------------------------------------------
   (1, 1)  (1, 1)  (2, 3)  (2, 3)  (3, 1)  (3, 1)   -   -
```

Figure 2.35 Fitting the strips with entries marked by character.

```
Displacement [0..3] = (0, 1, 4, 5);
Mark [1..8] = (1, 1, 3, 3, 1, 1, 0, 0);
Entry [1..6] = (1, 1, 2, 2, 3, 3);
```

Figure 2.36 The transition matrix compressed with marking by character.

```
IF Mark [Displacement [State] + Ch] /= Ch:
    SET New state TO No state;
ELSE entry in Entry [ ] is valid:
    SET New state TO Entry [Displacement [State] + Ch];
```

Figure 2.37 Code for `SET New state TO Next state[State, Ch]` for marking by character.

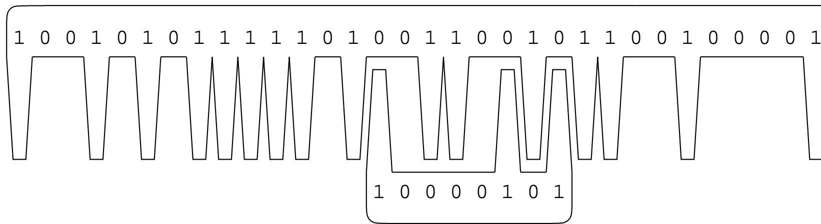


Figure 2.38 A damaged comb finding room for its teeth.

(a)

	w	x	y	z
0	1	2	–	–
1	3	–	4	–
2	1	–	–	6
3	–	2	–	–
4	–	–	–	5
5	1	–	4	–
6	–	7	–	–
7	–	–	–	–

(b)

	w	x	y	z	w	x	y	z
0	1	2	–	–				
1					3	–	4	–
2	1	–	–	6				
3	–	2	–	–				
4					–	–	–	5
5	1	–	4	–				
6					–	7	–	–
7					–	–	–	–
	1	2	4	6	3	7	4	5

Figure 2.39 A transition table (a) and its compressed form packed by graph coloring (b).

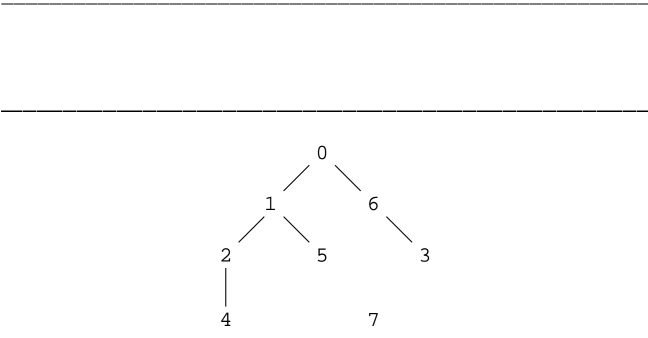


Figure 2.40 Interference graph for the automaton of Figure 2.39a.

```
%{
#include    "lex.h"

Token_Type Token;
int line_number = 1;
%}

whitespace          [ \t]

letter              [a-zA-Z]
digit               [0-9]
underscore          "_"
letter_or_digit     ({letter}|{digit})
underscored_tail    ({underscore}{letter_or_digit}+)
identifier          ({letter}{letter_or_digit}*{underscored_tail}*)

operator            [-+*/]
separator           [;,(){}]

%%

{digit}+                {return INTEGER;}
{identifier}            {return IDENTIFIER;}
{operator}|{separator}  {return yytext[0];}
#[^#\n]*#?              {/* ignore comment */}
{whitespace}            {/* ignore whitespace */}
\n                      {line_number++;}
.                       {return ERRONEOUS;}

%%

void start_lex(void) {}

void get_next_token(void) {
    Token.class = yylex();
    if (Token.class == 0) {
        Token.class = EOF; Token.repr = "<EOF>"; return;
    }
    Token.pos.line_number = line_number;
    strcpy(Token.repr = (char*)malloc(strlen(yytext)+1), yytext);
}

int yywrap(void) {return 1;}
```

Figure 2.41 *Lex* input for the token set from Figure 2.20.

```
FUNCTION Get next token () RETURNING a token:
    SET Simple token TO Get next simple token ();
    IF Class of Simple token = Identifier:
        SET Simple token TO Identify in symbol table (Simple token);
        // See if this has reset Class of Simple token:
        IF Class of Simple token = Macro:
            Switch to macro (Simple token);
            RETURN Get next token ();
        ELSE Class of Simple token /= Macro:
            // Identifier or Type identifier or Keyword:
            RETURN Simple token;
    ELSE Class of Simple token /= Identifier:
        RETURN Simple token;
```

Figure 2.42 A `Get next token()` that does lexical identification.

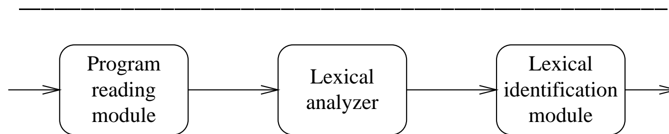


Figure 2.43 Pipeline from input to lexical identification.

```
FUNCTION Identify (Identifier name)
    RETURNING a pointer to Identifier info:
    SET Bucket number TO Hash value (Identifier name);
    IF there is no Identifier info record for Identifier name
        in Bucket [Bucket number]:
        Insert an empty Identifier info record for Identifier name
            in Bucket [Bucket number];
    RETURN Pointer to Identifier info record for Identifier name
        in Bucket [Bucket number];
```

Figure 2.44 Outline of the function `Identify`.

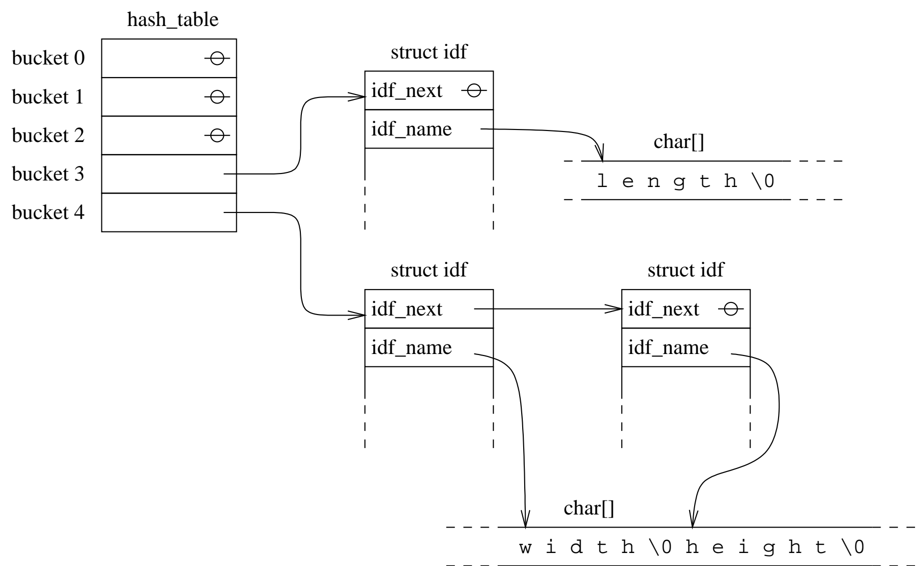


Figure 2.45 A hash table of size 5 containing three identifiers.

```
FUNCTION Identify in symbol table (Token) RETURNING a token:
    SET Identifier info pointer TO Identify (Representation of Token);
    IF Identifier info indicates macro:
        SET Class of Token TO Macro;
    ELSE IF Identifier info indicates keyword:
        SET Class of Token TO Keyword;
    ELSE IF Identifier info indicates type identifier:
        SET Class of Token TO Type identifier;
    ELSE Identifier info indicates
        variable, struct, union, etc. identifier:
        // Do not change Class of Token;
    Append Identifier info pointer to Token;
    RETURN Token;
```

Figure 2.46 A possible `Identify in symbol table(Token)` for C.

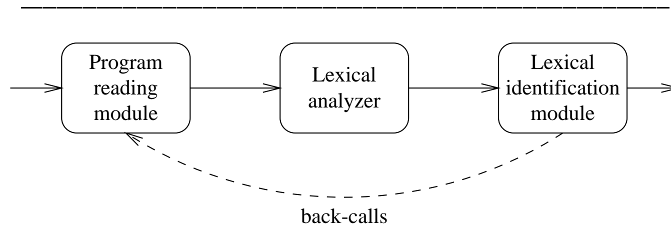


Figure 2.47 Pipeline from input to lexical identification, with feedback.

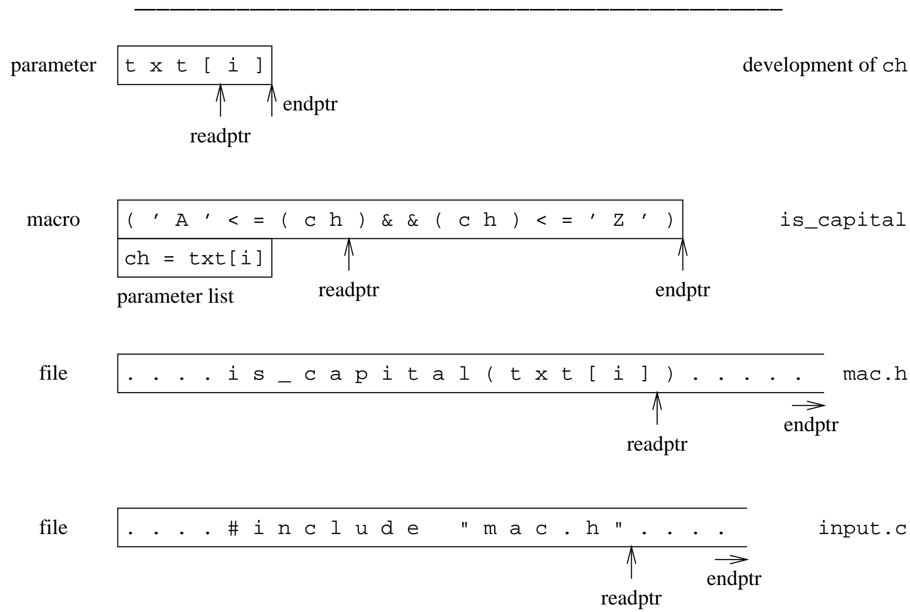


Figure 2.48 An input buffer stack of include files, macro calls, and macro parameters.

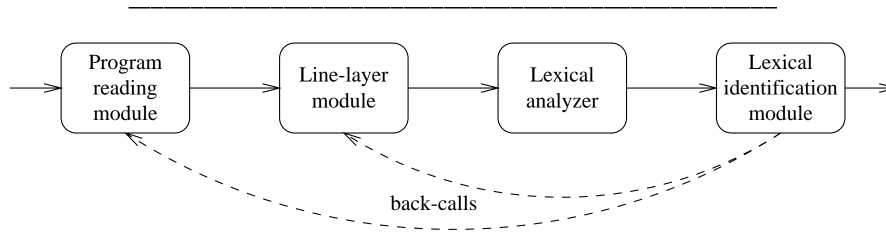


Figure 2.49 Input pipeline, with line layer and feedback.

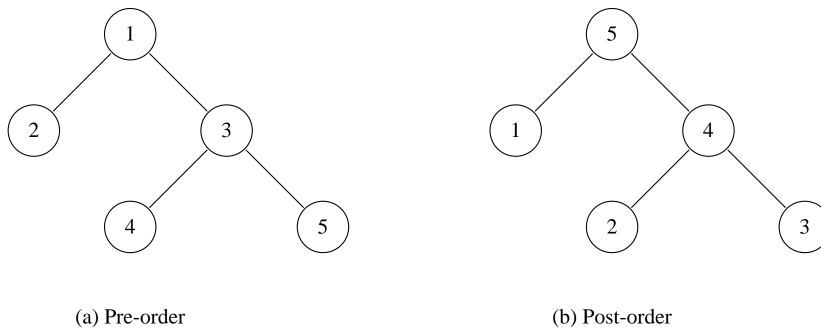


Figure 2.50 A tree with its nodes numbered in pre-order and post-order.

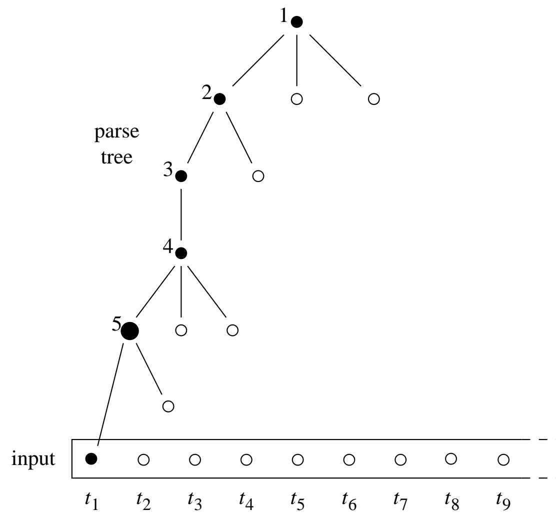


Figure 2.51 A top-down parser recognizing the first token in the input.

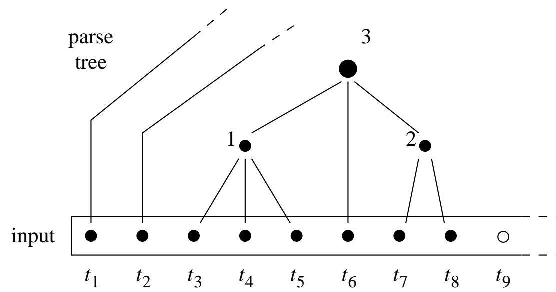


Figure 2.52 A bottom-up parser constructing its first, second, and third nodes.

```
input → expression EOF
expression → term rest_expression
term → IDENTIFIER | parenthesized_expression
parenthesized_expression → '(' expression ')'
rest_expression → '+' expression | ε
```

Figure 2.53 A simple grammar for demonstrating top-down parsing.

```
#include    "tokennumbers.h"

/* PARSER */
int input(void) {
    return expression() && require(token(EOF));
}

int expression(void) {
    return term() && require(rest_expression());
}

int term(void) {
    return token(IDENTIFIER) || parenthesized_expression();
}

int parenthesized_expression(void) {
    return token('(') && require(expression()) && require(token(')'));
}

int rest_expression(void) {
    return token('+') && require(expression()) || 1;
}

int token(int tk) {
    if (tk != Token.class) return 0;
    get_next_token(); return 1;
}

int require(int found) {
    if (!found) error();
    return 1;
}
```

Figure 2.54 A recursive descent parser/recognizer for the grammar of Figure 2.53.

```
#include    "lex.h"    /* for start_lex(), get_next_token(), Token */

/* DRIVER */
int main(void) {
    start_lex(); get_next_token();
    require(input());
    return 0;
}

void error(void) {
    printf("Error in expression\n"); exit(1);
}
```

Figure 2.55 Driver for the recursive descent parser/recognizer.

```
S → A 'a' 'b'
A → 'a' | ε
```

Figure 2.56 A simple grammar with a FIRST/FOLLOW conflict.

```
int S(void) {
    return A() && require(token('a')) && require(token('b'));
}

int A(void) {
    return token('a') || 1;
}
```

Figure 2.57 A faulty recursive recognizer for grammar of Figure 2.56.

Data definitions:

1. Token sets called FIRST sets for all terminals, non-terminals and alternatives of non-terminals in G.
2. A token set called FIRST for each alternative tail in G; an alternative tail is a sequence of zero or more grammar symbols α if $A\alpha$ is an alternative or alternative tail in G.

Initializations:

1. For all terminals T, set FIRST(T) to $\{T\}$.
2. For all non-terminals N, set FIRST(N) to the empty set.
3. For all non-empty alternatives and alternative tails α, set FIRST(α) to the empty set.
4. Set the FIRST set of all empty alternatives and alternative tails to $\{\varepsilon\}$.

Inference rules:

1. For each rule $N \rightarrow \alpha$ in G, FIRST(N) must contain all tokens in FIRST(α), including ε if FIRST(α) contains it.
2. For each alternative or alternative tail α of the form $A\beta$, FIRST(α) must contain all tokens in FIRST(A), excluding ε, should FIRST(A) contain it.
3. For each alternative or alternative tail α of the form $A\beta$ and FIRST(A) contains ε, FIRST(α) must contain all tokens in FIRST(β), including ε if FIRST(β) contains it.

Figure 2.58 Closure algorithm for computing the FIRST sets in a grammar G.

Rule/alternative (tail)	FIRST set
`input`	{ }
`    expression EOF`	{ }
`        EOF`	{ EOF }
`expression`	{ }
`    term rest_expression`	{ }
`        rest_expression`	{ }
`term`	{ }
`    IDENTIFIER`	{ IDENTIFIER }
`|`	
`    parenthesized_expression`	{ }
`parenthesized_expression`	{ }
`    '(' expression ')'`	{ '(' }
`        expression ')'`	{ }
`            ')'`	{ ')' }
`rest_expression`	{ }
`    '+' expression`	{ '+' }
`        expression`	{ }
`|   ε`	{ ε }

Figure 2.59 The initial FIRST sets.

```
Rule/alternative (tail)              FIRST set
input                                { IDENTIFIER '(' }
    expression EOF                   { IDENTIFIER '(' }
        EOF                          { EOF }

expression                           { IDENTIFIER '(' }
    term rest_expression             { IDENTIFIER '(' }
        rest_expression              { '+' ε }

term                                 { IDENTIFIER '(' }
    IDENTIFIER                       { IDENTIFIER }
|
    parenthesized_expression         { '(' }

parenthesized_expression             { '(' }
    '(' expression ')'               { '(' }
        expression ')'               { IDENTIFIER '(' }
            ')'                      { ')' }

rest_expression                      { '+' ε }
    '+' expression                   { '+' }
        expression                   { IDENTIFIER '(' }
|   ε                                { ε }
```

Figure 2.60 The final FIRST sets.

```
void input(void) {
    switch (Token.class) {
    case IDENTIFIER: case '(':
                        expression(); token(EOF); break;
    default:            error();
    }
}

void expression(void) {
    switch (Token.class) {
    case IDENTIFIER: case '(':
                        term(); rest_expression(); break;
    default:            error();
    }
}

void term(void) {
    switch (Token.class) {
    case IDENTIFIER:    token(IDENTIFIER); break;
    case '(':           parenthesized_expression(); break;
    default:            error();
    }
}

void parenthesized_expression(void) {
    switch (Token.class) {
    case '(':           token('('); expression(); token(')'); break;
    default:            error();
    }
}

void rest_expression(void) {
    switch (Token.class) {
    case '+':           token('+'); expression(); break;
    case EOF: case ')': break;
    default:            error();
    }
}

void token(int tk) {
    if (tk != Token.class) error();
    get_next_token();
}
```

Figure 2.61 A predictive parser for the grammar of Figure 2.53.

Data definitions:

1. Token sets called FOLLOW sets for all non-terminals in G.
2. Token sets called FIRST sets for all alternatives and alternative tails in G.

Initializations:

1. For all non-terminals N, set FOLLOW(N) to the empty set.
2. Set all FIRST sets to the values determined by the algorithm for FIRST sets.

Inference rules:

1. For each rule of the form $M \rightarrow \alpha N \beta$ in G, FOLLOW(N) must contain all tokens in FIRST(β), excluding ε, should FIRST(β) contain it.
2. For each rule of the form $M \rightarrow \alpha N \beta$ in G where FIRST(β) contains ε, FOLLOW(N) must contain all tokens in FOLLOW(M).

Figure 2.62 Closure algorithm for computing the FOLLOW sets in grammar G.

Rule	FIRST set	FOLLOW set
`input`	{ `IDENTIFIER '('` }	{ }
`expression`	{ `IDENTIFIER '('` }	{ `EOF ')'` }
`term`	{ `IDENTIFIER '('` }	{ `'+' EOF ')'` }
`parenthesized_expression`	{ `'('` }	{ `'+' EOF ')'` }
`rest_expression`	{ `'+'` ε }	{ `EOF ')'` }

Figure 2.63 The FIRST and FOLLOW sets for the grammar from Figure 2.53.

```
void S(void) {
    switch (Token.class) {
    case 'a':   A(); token('a'); token('b'); break;
    default:    error();
    }
}

void A(void) {
    switch (Token.class) {
    case 'a':   token('a'); break;
    case 'a':   break;
    default:    error();
    }
}
```

Figure 2.64 A predictive parser for the grammar of Figure 2.56.

Top of stack/state:	Look-ahead token				
	IDENTIFIER	+	(	)	EOF
input	expression		expression		
expression	term rest_-expression		term rest_-expression		
term	IDENTIFIER		parenthesized_-expression		
parenthesized_-expression			(expression)		
rest_expression		+ expression		ε	ε

Figure 2.65 Transition table for an LL(1) parser for the grammar of Figure 2.53.

```
Prediction stack:   expression ')' rest_expression EOF
Present input:      IDENTIFIER '+' IDENTIFIER ')' '+' IDENTIFIER EOF
```

Figure 2.66 Prediction stack and present input in a push-down automaton.

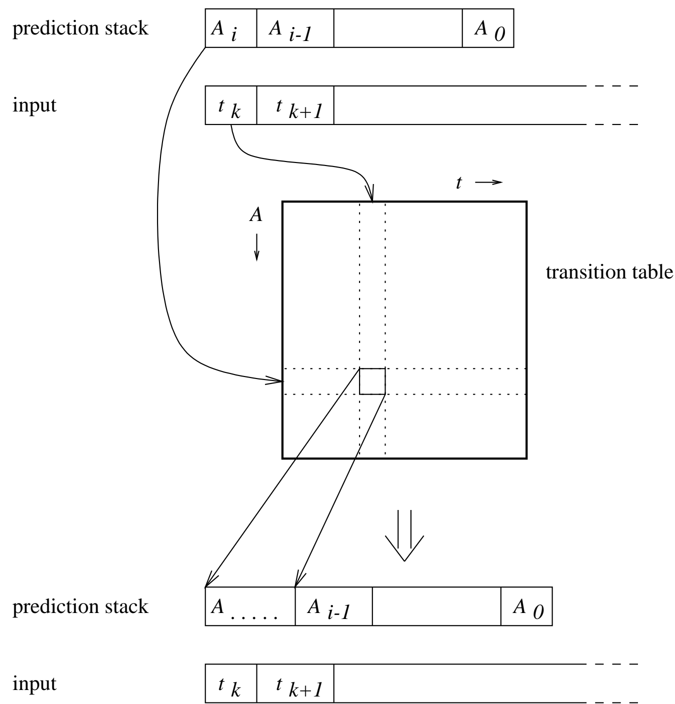


Figure 2.67 Prediction move in an LL(1) push-down automaton.

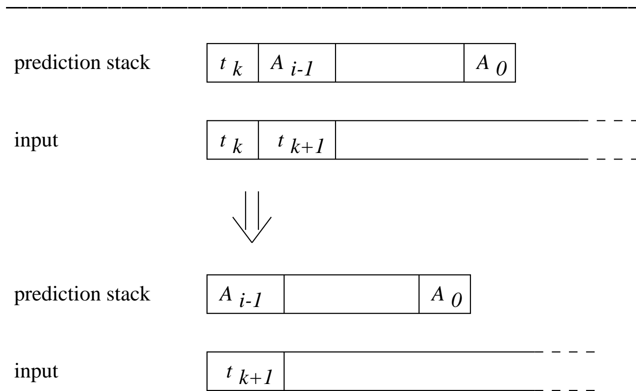


Figure 2.68 Match move in an LL(1) push-down automaton.

```
IMPORT Input token [1..];        // from lexical analyzer

SET Input token index TO 1;
SET Prediction stack TO Empty stack;
PUSH Start symbol ON Prediction stack;

WHILE Prediction stack /= Empty stack:
    Pop top of Prediction stack into Predicted;
    IF Predicted is a Token class:
        // Try a match move:
        IF Predicted = Input token [Input token index] .class:
            SET Input token index TO Input token index + 1; // matched
        ELSE Predicted /= Input token:
            ERROR "Expected token not found: ", Predicted;
    ELSE Predicted is a Non-terminal:
        // Try a prediction move, using the input token as look-ahead:
        SET Prediction TO
            Prediction table [Predicted, Input token [Input token index]];
        IF Prediction = Empty:
            ERROR "Token not expected: ", Input token [Input token index]];
        ELSE Prediction /= Empty:
            PUSH The symbols of Prediction ON Prediction stack;
```

Figure 2.69 Predictive parsing with an LL(1) push-down automaton.

```
Initial situation:
Prediction stack:   input
Input:              '(' IDENTIFIER '+' IDENTIFIER ')' '+' IDENTIFIER EOF

Prediction moves:
Prediction stack:   expression EOF
Prediction stack:   term rest_expression EOF
Prediction stack:   parenthesized_expression rest_expression EOF
Prediction stack:   '(' expression ')' rest_expression EOF
Input:              '(' IDENTIFIER '+' IDENTIFIER ')' '+' IDENTIFIER EOF

Match move on '(':
Prediction stack:   expression ')' rest_expression EOF
Input:              IDENTIFIER '+' IDENTIFIER ')' '+' IDENTIFIER EOF
```

Figure 2.70 The first few parsing moves for `(i+i)+i`.

```
// Step 1: construct acceptable set:
SET Acceptable set TO Acceptable set of (Prediction stack);

// Step 2: skip unacceptable tokens:
WHILE Input token [Input token index] IS NOT IN Acceptable set:
    MESSAGE "Token skipped: ", Input token [Input token index];
    SET Input token index TO Input token index + 1;

// Step 3: resynchronize the parser:
SET the flag Resynchronized TO False;
WHILE NOT Resynchronized:
    Pop top of Prediction stack into Predicted;
    IF Predicted is a Token class:
        // Try a match move:
        IF Predicted = Input token [Input token index] .class:
            SET Input token index TO Input token index + 1; // matched
            SET Resynchronized TO True;          // resynchronized!
        ELSE Predicted /= Input token:
            Insert a token of class Predicted, including representation;
            MESSAGE "Token inserted of class ", Predicted;
    ELSE Predicted is a Non-terminal:
        // Do a prediction move:
        SET Prediction TO
            Prediction table [Predicted, Input token [Input token index]];
        IF Prediction = Empty:
            SET Prediction TO Shortest production table [Predicted];
        // Now Prediction /= Empty:
        PUSH The symbols of Prediction ON Prediction stack;
```

Figure 2.71 Acceptable-set error recovery in a predictive parser.

Data definitions:

1. A set of pairs of the form (production rule, integer).
2a. A set of pairs of the form (non-terminal, integer).
2b. A set of pairs of the form (terminal, integer).

Initializations:

1a. For each production rule $N \rightarrow A_1 ... A_n$ with $n > 0$ there is a pair $(N \rightarrow A_1 ... A_n, \infty)$.
1b. For each production rule $N \rightarrow \varepsilon$ there is a pair $(N \rightarrow \varepsilon, 0)$.
2a. For each non-terminal N there is a pair (N, ∞).
2b. For each terminal T there is a pair $(T, 1)$.

Inference rules:

1. For each production rule $N \rightarrow A_1 ... A_n$ with $n > 0$, if there are pairs (A_1, l_1) to (A_n, l_n) with all $l_i < \infty$, the pair $(N \rightarrow A_1 ... A_n, l_N)$ must be replaced by a pair $(N \rightarrow A_1 ... A_n, l_{new})$ where $l_{new} = \sum_{i=1}^{n} l_i$ provided $l_{new} < l_N$.
2. For each non-terminal N, if there are one or more pairs of the form $(N \rightarrow \alpha, l_i)$ with $l_i < \infty$, the pair (N, l_N) must be replaced by (N, l_{new}) where l_{new} is the minimum of the l_is, provided $l_{new} < l_N$.

Figure 2.72 Closure algorithm for computing lengths of shortest productions.

Non-terminal	Alternative	Shortest length
`input`	`expression EOF`	2
`expression`	`term rest_expression`	1
`term`	`IDENTIFIER`	1
`parenthesized_expression`	`'(' expression ')'`	3
`rest_expression`	ε	0

Figure 2.73 Shortest production table for the grammar of Figure 2.53.

Error detected, since `Prediction table [expression, '+']` is empty:
Prediction stack: `expression ')' rest_expression EOF`
Input: `'+' IDENTIFIER ')' '+' IDENTIFIER EOF`

Shortest production for expression:
Prediction stack: `term rest_expression ')' rest_expression EOF`
Input: `'+' IDENTIFIER ')' '+' IDENTIFIER EOF`

Shortest production for term:
Prediction stack: `IDENTIFIER rest_expression ')' rest_expression EOF`
Input: `'+' IDENTIFIER ')' '+' IDENTIFIER EOF`

Token IDENTIFIER inserted in the input and matched:
Prediction stack: `rest_expression ')' rest_expression EOF`
Input: `'+' IDENTIFIER ')' '+' IDENTIFIER EOF`

Normal prediction for rest_expression, resynchronized:
Prediction stack: `'+' expression ')' rest_expression EOF`
Input: `'+' IDENTIFIER ')' '+' IDENTIFIER EOF`

Figure 2.74 Some steps in parsing `(i++i)+i`.

```
expression(int *e) →
    term(int *t)                {*e = *t;}
    expression_tail_option(int *e)
;

expression_tail_option(int *e) →
    '-' term(int *t)            {*e -= *t;}
    expression_tail_option(int *e)
|   ε
;

term(int *t) → IDENTIFIER {*t = 1;};
```

Figure 2.75 Minimal non-left-recursive grammar for expressions.

```
%lexical get_next_token_class;
%token IDENTIFIER;

%start Main_Program, main;

main {int result;}:
    expression(&result)          {printf("result = %d\n", result);}
;

expression(int *e) {int t;}:
    term(&t)                     {*e = t;}
    expression_tail_option(e)
;

expression_tail_option(int *e) {int t;}:
    '-' term(&t)                 {*e -= t;}
    expression_tail_option(e)
|
;

term(int *t):
    IDENTIFIER                   {*t = 1;}
;
```

Figure 2.76 *LLgen* code for a parser for simple expressions.

```
{
#include    "lex.h"

int main(void) {
    start_lex(); Main_Program(); return 0;
}

Token_Type Last_Token;          /* error recovery support */
int Reissue_Last_Token = 0;     /* idem */

int get_next_token_class(void) {
    if (Reissue_Last_Token) {
        Token = Last_Token;
        Reissue_Last_Token = 0;
    }
    else get_next_token();
    return Token.class;
}

void insert_token(int token_class) {
    Last_Token = Token; Reissue_Last_Token = 1;
    Token.class = token_class;
    /* and set the attributes of Token, if any */
}

void print_token(int token_class) {
    switch (token_class) {
    case IDENTIFIER: printf("IDENTIFIER"); break;
    case EOFILE    : printf("<EOF>"); break;
    default        : printf("%c", token_class); break;
    }
}

#include    "../../fg2/LLgen/LLmessage.i"
}
```

Figure 2.77 Auxiliary C code for a parser for simple expressions.

```
void LLmessage(int class) {
    switch (class) {
    default:
        insert_token(class);
        printf("Missing token ");
        print_token(class);
        printf(" inserted in front of token ");
        print_token(LLsymb); printf("\n");
        break;
    case 0:
        printf("Token deleted: ");
        print_token(LLsymb); printf("\n");
        break;
    case -1:
        printf("End of input expected, but found token ");
        print_token(LLsymb); printf("\n");
        break;
    }
}
```

Figure 2.78 The routine `LLmessage()` required by *LLgen*.

```
expression(struct expr **ep):
    {(*ep) = new_expr(); (*ep)->type = '-';}
    expression(&(*ep)->expr) '-' term(&(*ep)->term)
|   {(*ep) = new_expr(); (*ep)->type = 'T';}
    term(&(*ep)->term)
;

term(struct term **tp):
    {(*tp) = new_term(); (*tp)->type = 'I';}
    IDENTIFIER
;
```

Figure 2.79 Original grammar with code for constructing a parse tree.

```
struct expr {
    int type;           /* '-' or 'T' */
    struct expr *expr;  /* for '-' */
    struct term *term;  /* for '-' and 'T' */
};
#define new_expr()  ((struct expr *)malloc(sizeof(struct expr)))

struct term {
    int type;           /* 'I' only */
};
#define new_term()  ((struct term *)malloc(sizeof(struct term)))

extern void print_expr(struct expr *e);
extern void print_term(struct term *t);
```

Figure 2.80 Data structures for the parse tree.

```
expression(struct expr **ep):
    expression(ep)
    {   struct expr *e_aux = (*ep);
        (*ep) = new_expr(); (*ep)->type = '-'; (*ep)->expr = e_aux;
    }
    '-' term(&(*ep)->term)
|   {(*ep) = new_expr(); (*ep)->type = 'T';}
    term(&(*ep)->term)
;
```

Figure 2.81 Visibly left-recursive grammar with code for constructing a parse tree.

```
expression(struct expr **ep):
    {(*ep) = new_expr(); (*ep)->type = 'T';}
    term(&(*ep)->term)
    expression_tail_option(ep)
;

expression_tail_option(struct expr **ep):
    {   struct expr *e_aux = (*ep);
        (*ep) = new_expr(); (*ep)->type = '-'; (*ep)->expr = e_aux;
    }
    '-' term(&(*ep)->term)
    expression_tail_option(ep)
|
;
```

Figure 2.82 Adapted *LLgen* grammar with code for constructing a parse tree.

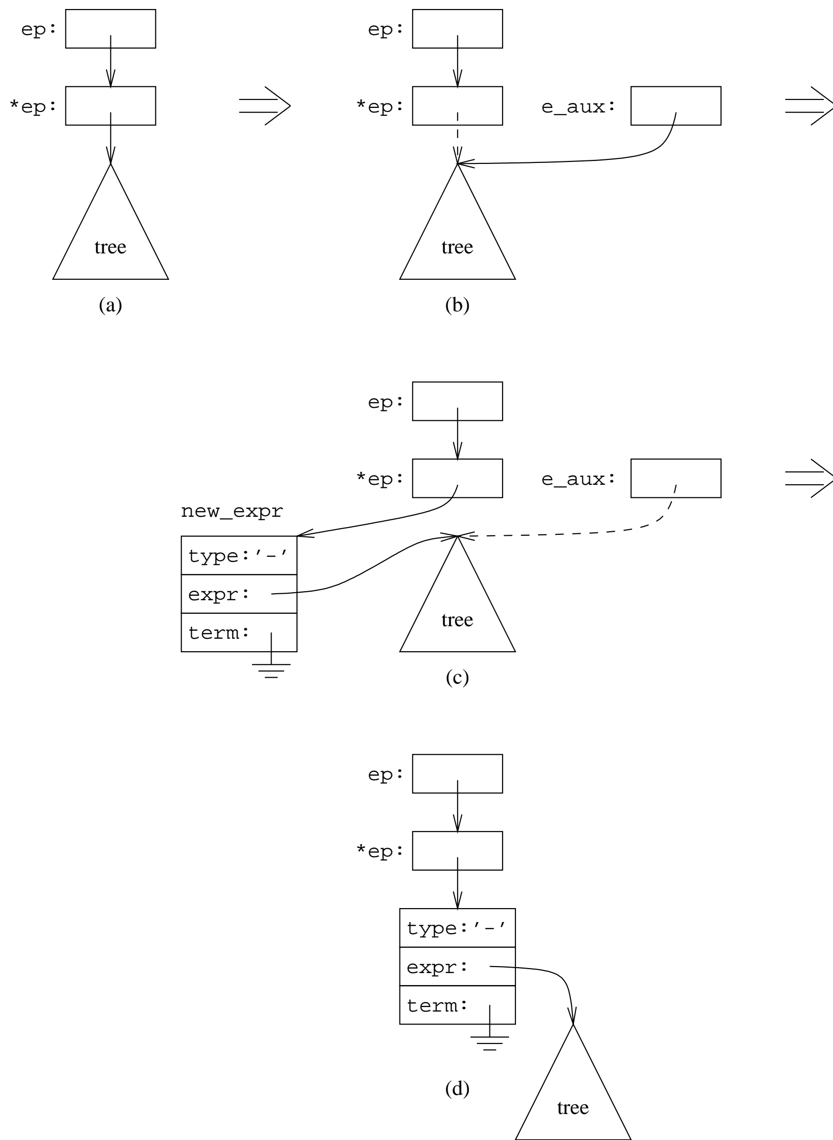


Figure 2.83 Tree transformation performed by `expression_tail_option`.

```
input → expression EOF
expression → term | expression '+' term
term → IDENTIFIER | '(' expression ')'
```

Figure 2.84 A simple grammar for demonstrating bottom-up parsing.

```
Z → E $
E → T | E + T
T → i | ( E )
```

Figure 2.85 An abbreviated form of the simple grammar for bottom-up parsing.

Data definitions:
A set S of LR(0) items.

Initializations:
S is prefilled externally with one or more LR(0) items.

Inference rules:
If S holds an item of the form $P \to \alpha \bullet N \beta$, then for each production rule $N \to \gamma$ in G, S must also contain the item $N \to \bullet \gamma$.

Figure 2.86 ε closure algorithm for LR(0) item sets for a grammar G.

```
FUNCTION Initial item set RETURNING an item set:
    SET New item set TO Empty;

    // Initial contents - obtain from the start symbol:
    FOR EACH production rule S→α for the start symbol S:
        SET New item set TO New item set + item S→•α;

    RETURN ε closure (New item set);
```

Figure 2.87 The routine `Initial item set` for an LR(0) parser.

```
FUNCTION Next item set (Item set, Symbol) RETURNING an item set:
    SET New item set TO Empty;

    // Initial contents - obtain from token moves:
    FOR EACH item N→α•Sβ IN Item set:
        IF S = Symbol:
            SET New item set TO New item set + item N→αS•β;

    RETURN ε closure (New item set);
```

Figure 2.88 The routine `Next item set()` for an LR(0) parser.

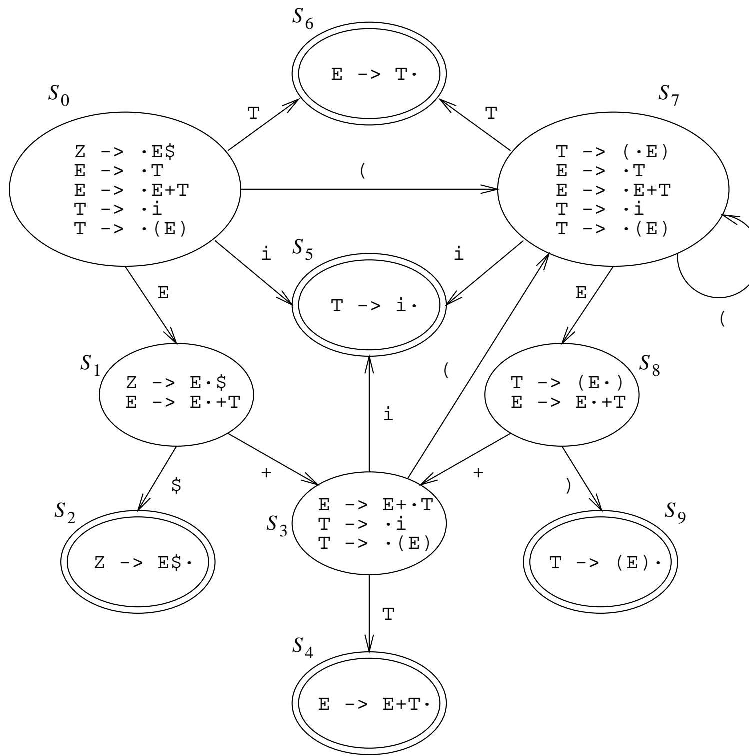


Figure 2.89 Transition diagram for the LR(0) automaton for the grammar of Figure 2.85.

	GOTO table symbol							ACTION table
state	i	+	(	)	$	E	T	
0	5		7			1	6	shift
1		3			2			shift
2								Z→E$
3	5		7				4	shift
4								E→E+T
5								T→i
6								E→T
7	5		7			8	6	shift
8		3		9				shift
9								T→(E)

Figure 2.90 GOTO and ACTION tables for the LR(0) automaton for the grammar of Figure 2.85.

Stack	Input	Action
S_0	i + i $	shift
S_0 i S_5	+ i $	reduce by T→i
S_0 T S_6	+ i $	reduce by E→T
S_0 E S_1	+ i $	shift
S_0 E S_1 + S_3	i $	shift
S_0 E S_1 + S_3 i S_5	$	reduce by T→i
S_0 E S_1 + S_3 T S_4	$	reduce by E→E+T
S_0 E S_1	$	shift
S_0 E S_1 $ S_2		reduce by Z→E$
S_0 Z		stop

Figure 2.91 LR(0) parsing of the input `i+i`.

```
IMPORT Input token [1..];        // from the lexical analyzer

SET Input token index TO 1;
SET Reduction stack TO Empty stack;
PUSH Start state ON Reduction stack;

WHILE Reduction stack /= {Start state, Start symbol, End state}
    SET State TO Top of Reduction stack;
    SET Action TO Action table [State];
    IF Action = "shift":
        // Do a shift move:
        SET Shifted token TO Input token [Input token index];
        SET Input token index TO Input token index + 1; // shifted
        PUSH Shifted token ON Reduction stack;
        SET New state TO Goto table [State, Shifted token .class];
        PUSH New state ON Reduction stack;      // can be Empty
    ELSE IF Action = ("reduce", N→α):
        // Do a reduction move:
        Pop the symbols of α from Reduction stack;
        SET State TO Top of Reduction stack;    // update State
        PUSH N ON Reduction stack;
        SET New state TO Goto table [State, N];
        PUSH New state ON Reduction stack;      // cannot be Empty
    ELSE Action = Empty:
        ERROR "Error at token ", Input token [Input token index];
```

Figure 2.92 LR(0) parsing with a push-down automaton.

state	look-ahead token i	+	(	)	$
0	shift				
1		shift			shift
2					Z→E$
3	shift		shift		
4		E→E+T		E→E+T	E→E+T
5		T→i		T→i	T→i
6		E→T		E→T	E→T
7	shift		shift		
8		shift		shift	
9		T→(E)		T→(E)	T→(E)

Figure 2.93 ACTION table for the SLR(1) automaton for the grammar of Figure 2.85.

state	stack symbol/look-ahead token i	+	(	)	$	E	T
0	s5					s1	s6
1		s3			s2		
2					r1		
3	s5		s7				s4
4		r3		r3	r3		
5		r4		r4	r4		
6		r2		r2	r2		
7	s5		s7			s8	s6
8		s3		s9			
9		r5		r5	r5		

Figure 2.94 ACTION/GOTO table for the SLR(1) automaton for the grammar of Figure 2.85.

```
S → A | xb
A → aAb | B
B → x
```

Figure 2.95 Grammar for demonstrating the LR(1) technique.

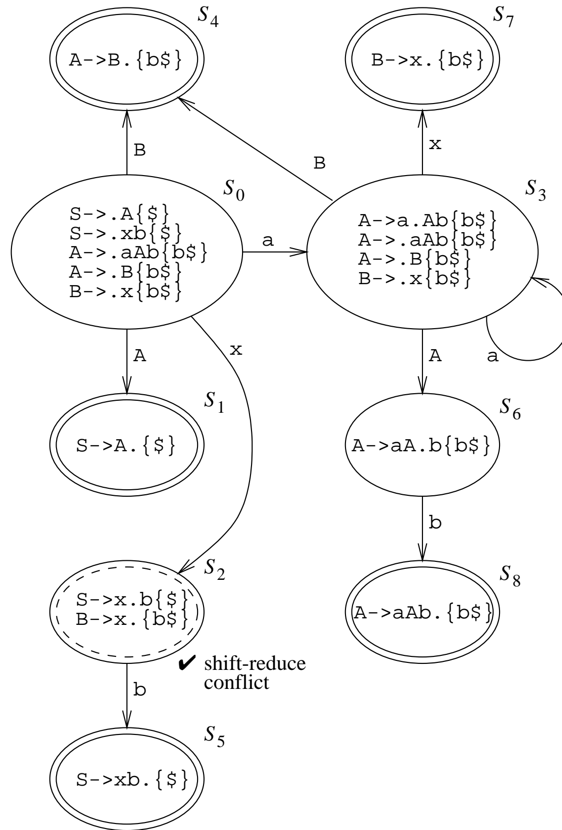


Figure 2.96 The SLR(1) automaton for the grammar of Figure 2.95.

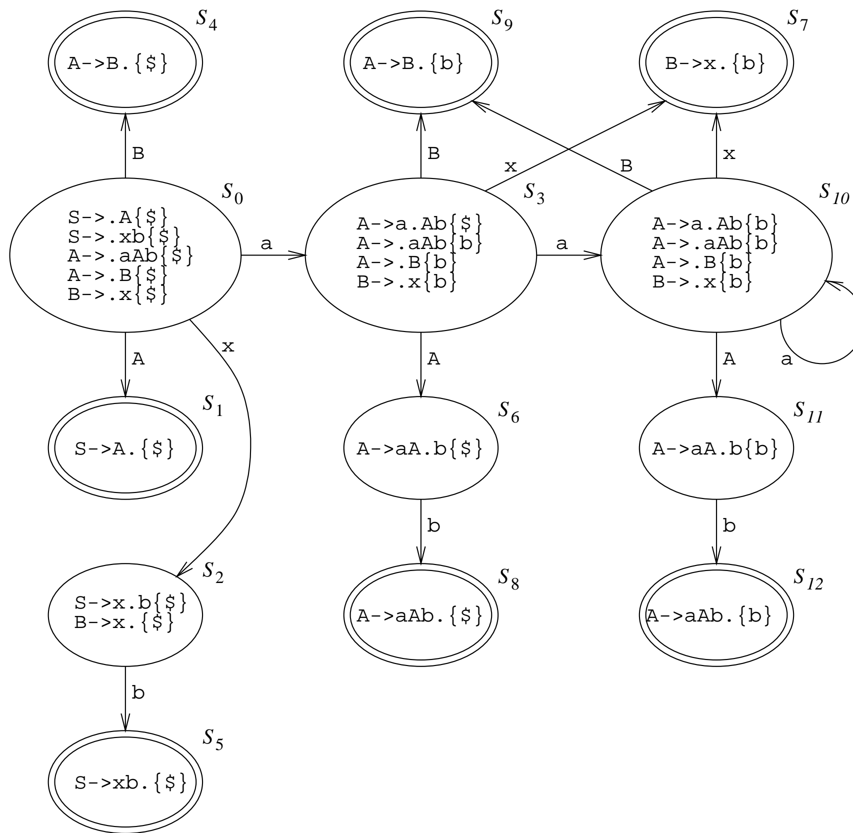


Figure 2.97 The LR(1) automaton for the grammar of Figure 2.95.

Data definitions:
A set S of LR(1) items of the form $N \rightarrow \alpha \bullet \beta\{\sigma\}$.

Initializations:
S is prefilled externally with one or more LR(1) items.

Inference rules:
If S holds an item of the form $P \rightarrow \alpha \bullet N\beta\{\sigma\}$, then for each production rule $N \rightarrow \gamma$ in G, S must also contain the item $N \rightarrow \bullet\gamma\{\tau\}$, where τ = FIRST($\beta\{\sigma\}$).

Figure 2.98 ε closure algorithm for LR(1) item sets for a grammar G.

```
IMPORT Input token [1..];       // from the lexical analyzer

SET Input token index TO 1;
SET Reduction stack TO Empty stack;
PUSH Start state ON Reduction stack;

WHILE Reduction stack /= {Start state, Start symbol, End state}
    SET State TO Top of Reduction stack;
    SET Look ahead TO Input token [Input token index] .class;
    SET Action TO Action table [State, Look ahead];
    IF Action = "shift":
        // Do a shift move:
        SET Shifted token TO Input token [Input token index];
        SET Input token index TO Input token index + 1; // shifted
        PUSH Shifted token ON Reduction stack;
        SET New state TO Goto table [State, Shifted token .class];
        PUSH New state ON Reduction stack;      // cannot be Empty
    ELSE IF Action = ("reduce", N→α):
        // Do a reduction move:
        Pop the symbols of α from Reduction stack;
        SET State TO Top of Reduction stack;    // update State
        PUSH N ON Reduction stack;
        SET New state TO Goto table [State, N];
        PUSH New state ON Reduction stack;      // cannot be Empty
    ELSE Action = Empty:
        ERROR "Error at token ", Input token [Input token index];
```

Figure 2.99 LR(1) parsing with a push-down automaton.

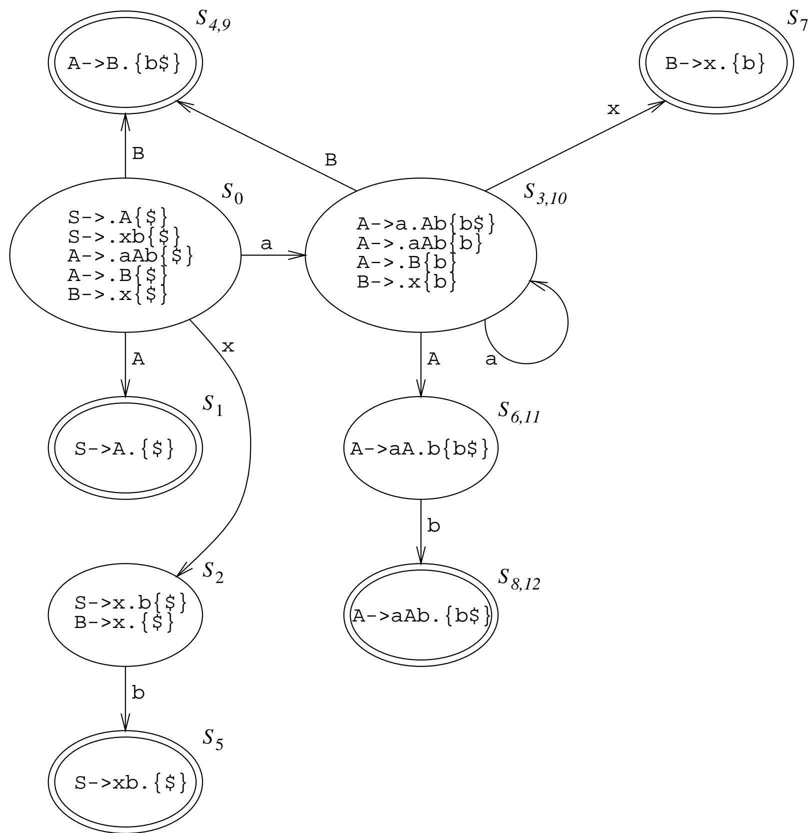


Figure 2.100 The LALR(1) automaton for the grammar of Figure 2.95.

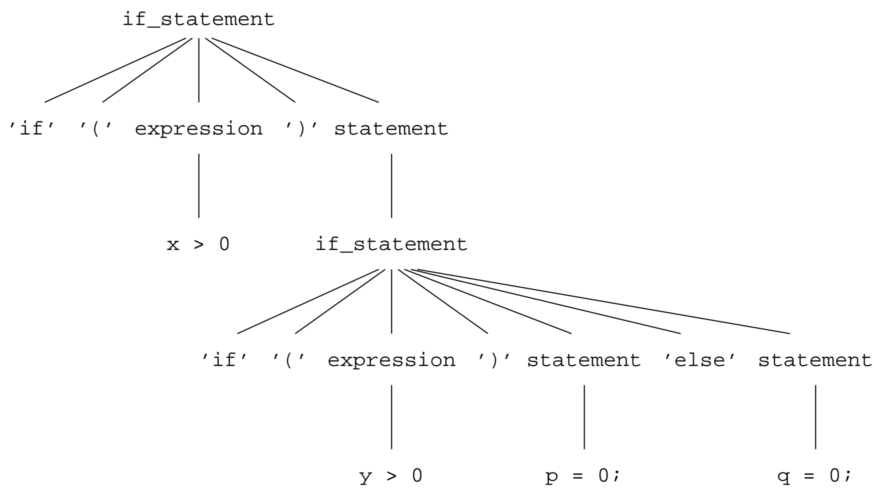


Figure 2.101 A possible syntax tree for a nested if-statement.

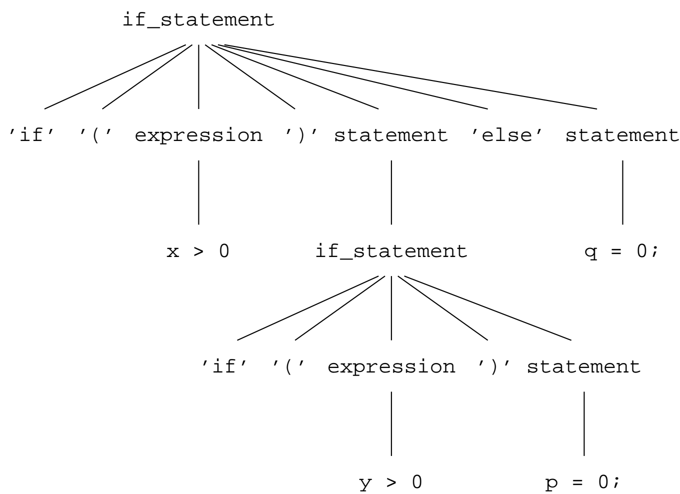


Figure 2.102 An alternative syntax tree for the same nested if-statement.

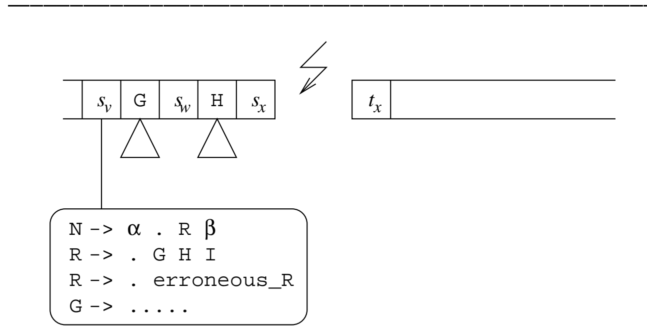


Figure 2.103 LR error recovery – detecting the error.

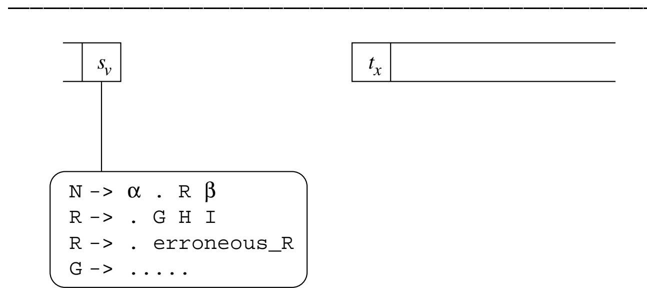


Figure 2.104 LR error recovery – finding an error recovery state.

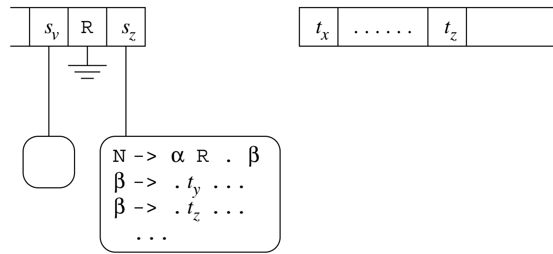


Figure 2.105 LR error recovery – repairing the stack.

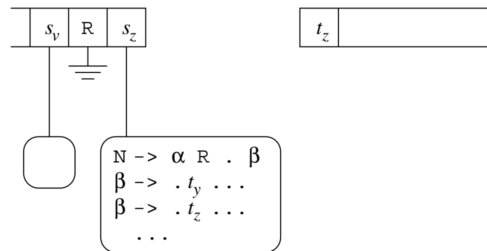


Figure 2.106 LR error recovery – repairing the input.

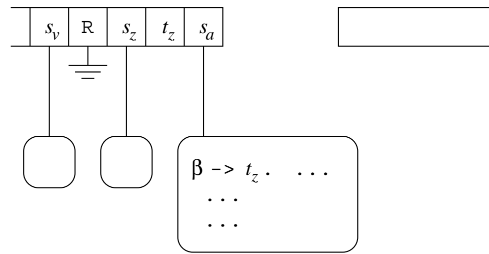


Figure 2.107 LR error recovery – restarting the parser.

```
%{
#include    "tree.h"
%}

%union {
    struct expr *expr;
    struct term *term;
}

%type <expr> expression;
%type <term> term;

%token IDENTIFIER

%start main

%%

main:
    expression    {print_expr($1); printf("\n");}
;

expression:
    expression '-' term
    {$$ = new_expr(); $$->type = '-'; $$->expr = $1; $$->term = $3;}
|
    term
    {$$ = new_expr(); $$->type = 'T'; $$->term = $1;}
;

term:
    IDENTIFIER
    {$$ = new_term(); $$->type = 'I';}
;

%%
```

Figure 2.108 *Yacc* code for constructing parse trees.

```
#include    "lex.h"

int main(void) {
    start_lex();
    yyparse();          /* routine generated by yacc */
    return 0;
}

int yylex(void) {
    get_next_token();
    return Token.class;
}

int
yyerror(const char *msg) {
    fprintf(stderr, "%s\n", msg);
    return 0;
}
```

Figure 2.109 Auxiliary code for the *yacc* parser for simple expressions.

	Lexical analysis	Syntax analysis
Top-down	Decision on first character: manual method	Decision on first token: LL(1) method
Bottom-up	Decision on reduce items: finite-state automata	Decision on reduce items: LR techniques

Figure 2.110 A very high-level view of text analysis techniques.

```
declaration: decl_specifiers init_declarator? ';'
decl_specifiers: type_specifier decl_specifiers?
type_specifier: 'int' | 'long'
init_declarator: declarator initializer?
declarator: IDENTIFIER | declarator '(' ')' | declarator '[' ']'
initializer:
    '=' expression
    | '=' '{' initializer_list '}' | '=' '{' initializer_list ',' '}'
initializer_list:
    expression
    | initializer_list ',' initializer_list | '{' initializer_list '}'
```

Figure 2.111 A simplified grammar for declarations in C.

```
type → actual_type | virtual_type
actual_type → actual_basic_type actual_size
virtual_type → virtual_basic_type virtual_size
actual_basic_type → 'int' | 'char'
actual_size → '[' NUMBER ']'
virtual_basic_type → 'int' | 'char' | 'void'
virtual_size → '[' ']'
```

Figure 2.112 Sample grammar for `type`.

3

Annotating the abstract syntax tree – the context

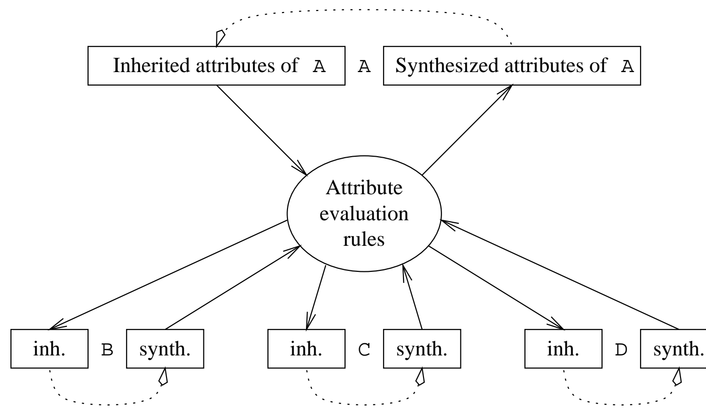


Figure 3.1 Data flow in a node with attributes.

```
Constant_definition (INH old symbol table, SYN new symbol table) →
    'CONST'  Defined_identifier  '='  Expression  ';'
    ATTRIBUTE RULES:
        SET Expression .symbol table TO
            Constant_definition .old symbol table;
        SET Constant_definition .new symbol table TO
            Updated symbol table (
                Constant_definition .old symbol table,
                Defined_identifier .name,
                Checked type of Constant_definition (Expression .type),
                Expression .value
            );

Defined_identifier (SYN name) → ...

Expression (INH symbol table, SYN type, SYN value) → ...
```

Figure 3.2 A simple attribute rule for `Constant_definition`.

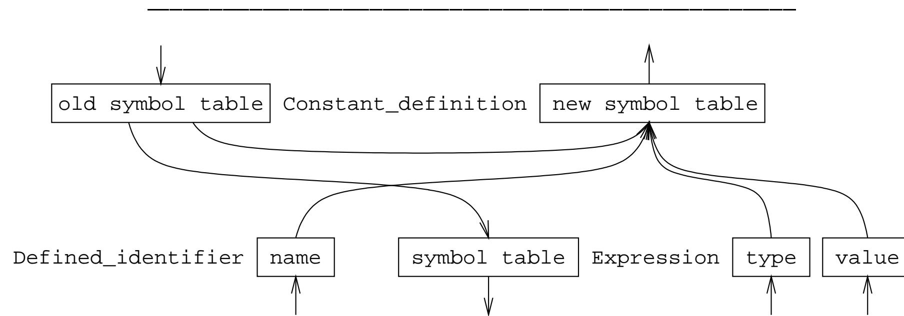


Figure 3.3 Dependency graph of the rule for `Constant_definition` from Figure 3.1.

```
Expression (INH symbol table, SYN type, SYN value) →
    Number
    ATTRIBUTE RULES:
        SET Expression .type TO Number .type;
        SET Expression .value TO Number .value;
```

Figure 3.4 Trivial attribute grammar for `Expression`.

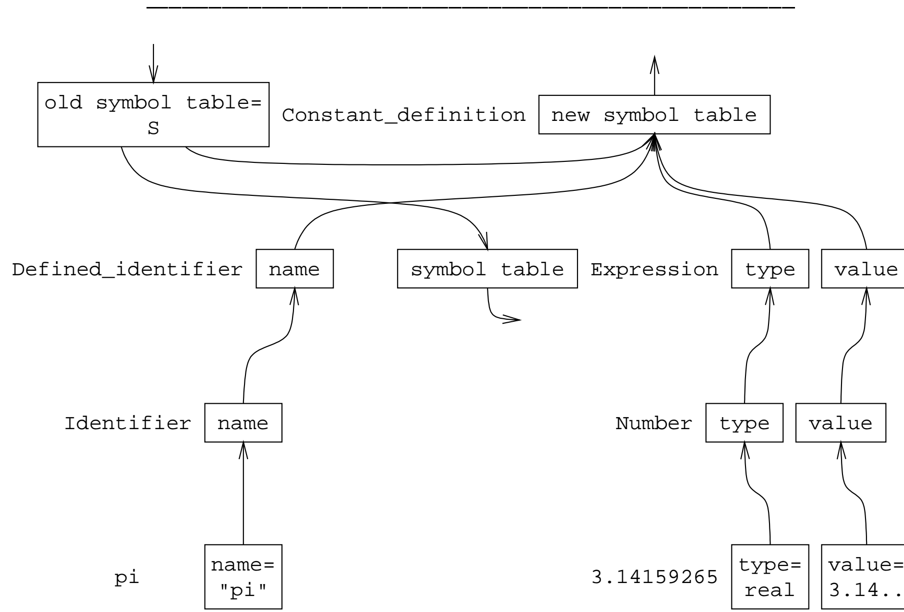


Figure 3.5 Sample attributed syntax tree with data flow.

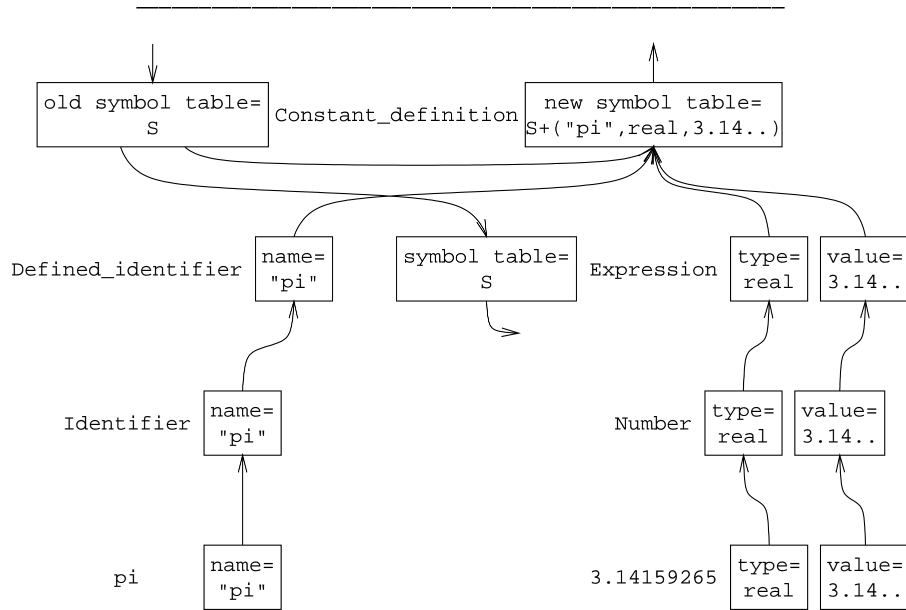


Figure 3.6 The attributed syntax tree from Figure 3.5 after attribute evaluation.

```
Number → Digit_Seq Base_Tag
Digit_Seq → Digit_Seq Digit | Digit
Digit → Digit_Token   // 0 1 2 3 4 5 6 7 8 9
Base_Tag → 'B' | 'D'
```

Figure 3.7 A context-free grammar for octal and decimal numbers.

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

Digit_Seq(INH base, SYN value) →
    Digit_Seq[1]  Digit
    ATTRIBUTE RULES
        SET Digit_Seq[1] .base TO Digit_Seq .base;
        SET Digit .base TO Digit_Seq .base;
        SET Digit_Seq .value TO
            Digit_Seq[1] .value * Digit_Seq .base + Digit .value;
|
    Digit
    ATTRIBUTE RULES
        SET Digit .base TO Digit_Seq .base;
        SET Digit_Seq .value TO Digit .value;

Digit(INH base, SYN value) →
    Digit_Token
    ATTRIBUTE RULES
        SET Digit .value TO Checked digit value (
            Value_of (Digit_Token .repr [0]) - Value_of ('0'),
            base
        );

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

Figure 3.8 An attribute grammar for octal and decimal numbers.

```
FUNCTION Checked digit value (Token value, Base) RETURNING an integer:
    IF Token value < Base: RETURN Token value;
    ELSE Token value >= Base:
        ERROR "Token " Token value " cannot be a digit in base " Base;
        RETURN Base - 1;
```

Figure 3.9 The function `Checked digit value`.

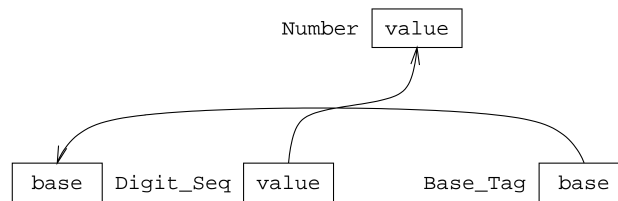


Figure 3.10 The dependency graph of `Number`.

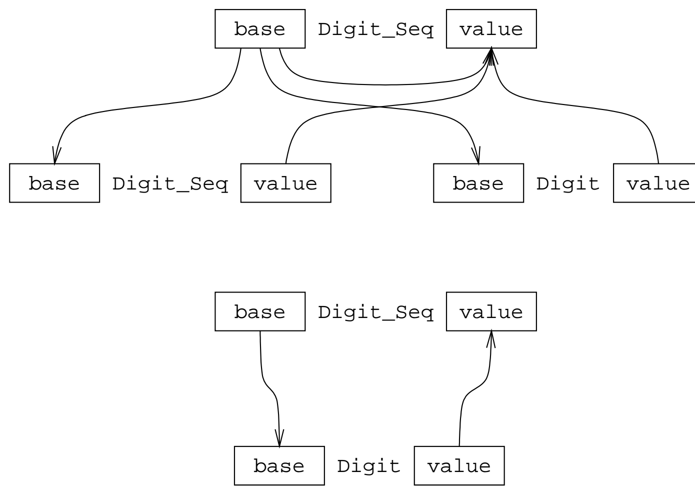


Figure 3.11 The two dependency graphs of `Digit_Seq`.

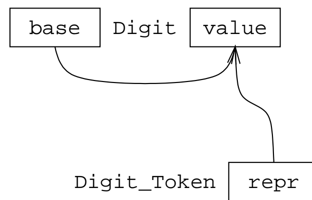


Figure 3.12 The dependency graph of `Digit`.

Figure 3.13 The two dependency graphs of Base_Tag.

```
PROCEDURE Evaluate for Digit_Seq alternative_1 (
    pointer to digit_seq node Digit_Seq,
    pointer to digit_seq alt_1 node Digit_Seq alt_1
):
    // Propagate attributes:
    Propagate for Digit_Seq alternative_1 (Digit_Seq, Digit_Seq alt_1);

    // Traverse subtrees:
    Evaluate for Digit_Seq (Digit_Seq alt_1 .Digit_Seq);
    Evaluate for Digit (Digit_Seq alt_1 .Digit);

    // Propagate attributes:
    Propagate for Digit_Seq alternative_1 (Digit_Seq, Digit_Seq alt_1);

PROCEDURE Propagate for Digit_Seq alternative_1 (
    pointer to digit_seq node Digit_Seq,
    pointer to digit_seq alt_1 node Digit_Seq alt_1
):
    IF Digit_Seq alt_1 .Digit_Seq .base is not set
        AND Digit_Seq .base is set:
        SET Digit_Seq alt_1 .Digit_Seq .base TO Digit_Seq .base;

    IF Digit_Seq alt_1 .Digit .base is not set
        AND Digit_Seq .base is set:
        SET Digit_Seq alt_1 .Digit .base TO Digit_Seq .base;

    IF Digit_Seq .value is not set
        AND Digit_Seq alt_1 .Digit_Seq .value is set
        AND Digit_Seq .base is set
        AND Digit_Seq alt_1 .Digit .value is set:
        SET Digit_Seq .value TO
            Digit_Seq alt_1 .Digit_Seq .value * Digit_Seq .base
            + Digit_Seq alt_1 .Digit .value;
```

Figure 3.14 Data-flow code for the first alternative of `Digit_Seq`.

```
PROCEDURE Run the data-flow attribute evaluator on node Number:
    WHILE Number .value is not set:
        PRINT "Evaluate for Number called";    // report progress
        Evaluate for Number (Number);

    // Print one attribute:
    PRINT "Number .value = ", Number .value;
```

Figure 3.15 Driver for the data-flow code.

```
FUNCTION Topological sort of (a set Set) RETURNING a list:
    SET 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:
    // First 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 3.16 Outline code for a simple implementation of topological sort.

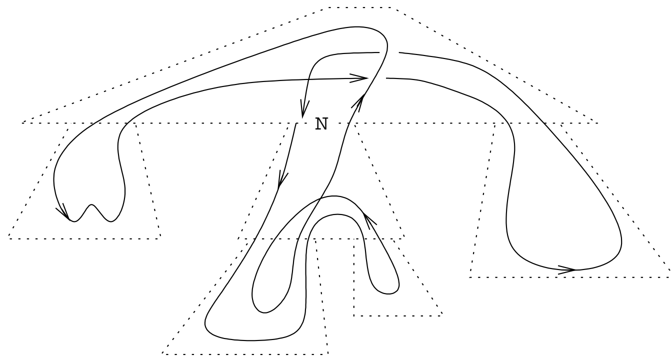


Figure 3.17 A fairly long, possibly circular, data-flow path.

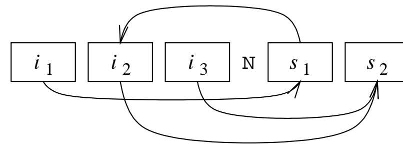


Figure 3.18 An example of an IS-SI graph.

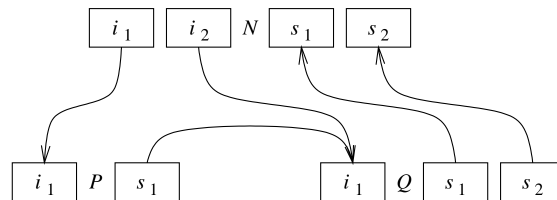


Figure 3.19 The dependency graph for the production rule $N \rightarrow PQ$.

```
SET the flag Something was changed TO True;

// Initialization step:
FOR EACH terminal T IN Attribute grammar:
    SET the IS-SI graph of T TO T's dependency graph;

FOR EACH non-terminal N IN Attribute grammar:
    SET the IS-SI graph of N TO the empty set;

WHILE Something was changed:
    SET Something was changed TO False;

    FOR EACH production rule P = M0→M1...Mn IN Attribute grammar:

        // Construct the dependency graph copy D:
        SET the dependency graph D TO a copy of the dependency graph of P;
        // Add the dependencies already found for Mi i=0...n:
        FOR EACH M IN M0...Mn:
            FOR EACH dependency d IN the IS-SI graph of M:
                Insert d in D;

        // Use the dependency graph D:
        Compute all induced dependencies in D by transitive closure;
        IF D contains a cycle: ERROR "Cycle found in production", P
        // Propagate the newly discovered dependencies:
        FOR EACH M IN M0...Mn:
            FOR EACH dependency d IN D
                SUCH THAT the attributes in d are attributes of M:
                IF there is no dependency d in the IS-SI graph of M:
                    Enter a dependency d into the IS-SI graph of M;
                    SET Something was changed TO True;
```

Figure 3.20 Outline of the strong-cyclicity test for an attribute grammar.

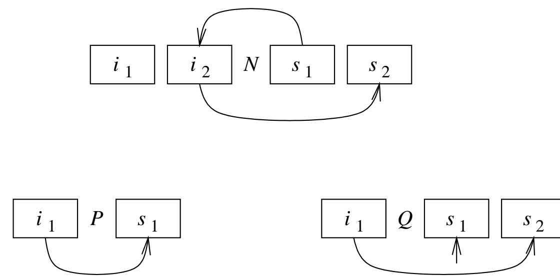


Figure 3.21 The IS-SI graphs of *N*, *P*, and *Q* collected so far.

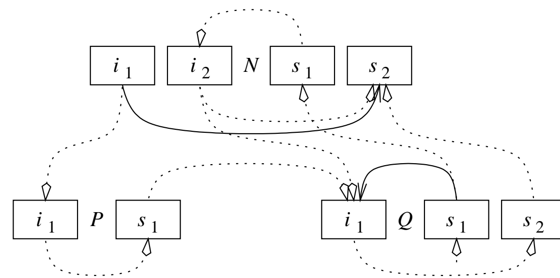


Figure 3.22 Transitive closure over the dependencies of *N*, *P*, *Q* and *D*.

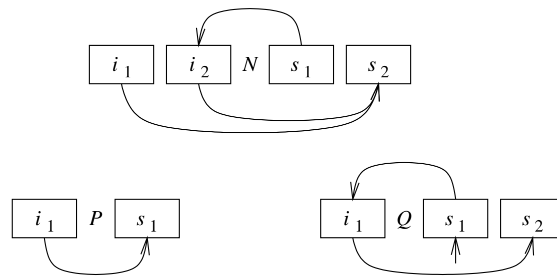


Figure 3.23 The new IS-SI graphs of *N*, *P*, and *Q*.

```
// Visit 1 from the parent: flow of control from parent enters here.
// The parent has set some inherited attributes, the set IN_1.

    // Visit some children M_k, M_l, ...:

    Compute some inherited attributes of M_k, the set (IM_k)_1;
    Visit M_k for the first time;
    // M_k returns with some of its synthesized attributes evaluated.

    Compute some inherited attributes of M_l, the set (IM_l)_1;
    Visit M_l for the first time;
    // M_l returns with some of its synthesized attributes evaluated.

    ... // Perhaps visit some more children, including possibly M_k or
        // M_l again, while supplying the proper inherited attributes
        // and obtaining synthesized attributes in return.

    // End of the visits to children.

    Compute some of N's synthesized attributes, the set SN_1;
    Leave to the parent;
// End of visit 1 from the parent.

// Visit 2 from the parent: flow of control re-enters here.
// The parent has set some inherited attributes, the set IN_2.

    ... // Again visit some children while supplying inherited
        // attributes and obtaining synthesized attributes in return.

    Compute some of N's synthesized attributes, the set SN_2;
    Leave to the parent;
// End of visit 2 from the parent.

... // Perhaps code for some more visits 3..n from the parent,
    // supplying sets IN_3 to IN_n and yielding
    // sets SN_3 to SN_n.
```

Figure 3.24 Outline code for multi-visit attribute evaluation.

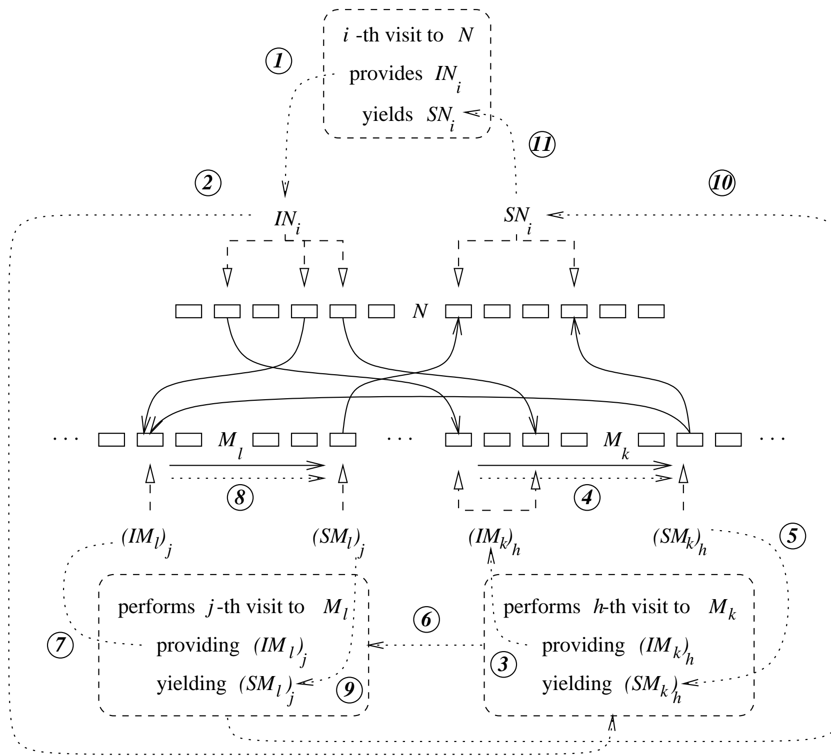


Figure 3.25 The i-th visit to a node N, visiting two children, M_k and M_l.

```
PROCEDURE Visit_i to N (pointer to an N node Node):
    SELECT Node .type:
        CASE alternative_1:
            Visit_i to N alternative_1 (Node);
        ...
        CASE alternative_k:
            Visit_i to N alternative_k (Node);
        ...
```

Figure 3.26 Structure of an i-th visit routine for N.

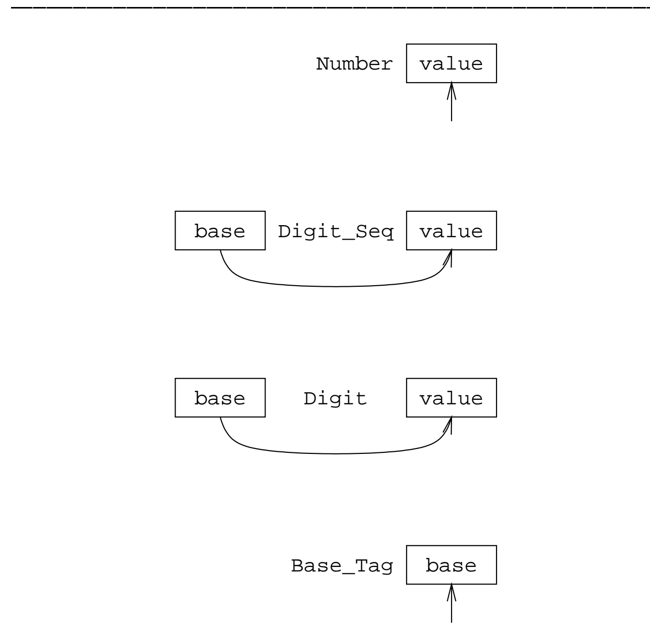


Figure 3.27 The IS-SI graphs of the non-terminals from grammar 3.8.

	IN_1	SN_1
Number		value
Digit_Seq	base	value
Digit	base	value
Base_Tag		base

Figure 3.28 Partitionings of the attributes of grammar 3.8.

```
PROCEDURE Visit_1 to Number alternative_1 (
    pointer to number node Number,
    pointer to number alt_1 node Number alt_1
):
// Visit 1 from the parent: flow of control from the parent enters here.
// The parent has set the attributes in IN1 of Number, the set { }.

    // Visit some children:

    // Compute the attributes in IN1 of Base_Tag (), the set { }:
    // Visit Base_Tag for the first time:
    Visit_1 to Base_Tag (Number alt_1 .Base_Tag);
    // Base_Tag returns with its SN1, the set { base }, evaluated.

    // Compute the attributes in IN1 of Digit_Seq (), the set { base }:
    SET Number alt_1 .Digit_Seq .base TO Number alt_1 .Base_Tag .base;
    // Visit Digit_Seq for the first time:
    Visit_1 to Digit_Seq (Number alt_1 .Digit_Seq);
    // Digit_Seq returns with its SN1, the set { value }, evaluated.

    // End of the visits to children.

    // Compute the attributes in SN1 of Number, the set { value }:
    SET Number .value TO Number alt_1 .Digit_Seq .value;
```

Figure 3.29 Visiting code for `Number` nodes.

```
PROCEDURE Visit_1 to Digit_Seq alternative_1 (
    pointer to digit_seq node Digit_Seq,
    pointer to digit_seq alt_1 node Digit_Seq alt_1
):
// Visit 1 from the parent: flow of control from the parent enters here.
// The parent has set the attributes in IN1 of Digit_Seq, the set { base }.

    // Visit some children:

    // Compute the attributes in IN1 of Digit_Seq (), the set { base }:
    SET Digit_Seq alt_1 .Digit_Seq .base TO Digit_Seq .base;
    // Visit Digit_Seq for the first time:
    Visit_1 to Digit_Seq (Digit_Seq alt_1 .Digit_Seq);
    // Digit_Seq returns with its SN1, the set { value }, evaluated.

    // Compute the attributes in IN1 of Digit (), the set { base }:
    SET Digit_Seq alt_1 .Digit .base TO Digit_Seq .base;
    // Visit Digit for the first time:
    Visit_1 to Digit (Digit_Seq alt_1 .Digit);
    // Digit returns with its SN1, the set { value }, evaluated.

    // End of the visits to children.

    // Compute the attributes in SN1 of Digit_Seq, the set { value }:
    SET Digit_Seq .value TO
        Digit_Seq alt_1 .Digit_Seq .value * Digit_Seq .base +
        Digit_Seq alt_1 .Digit .value;
```

Figure 3.30 Visiting code for `Digit_Seq alternative_1` nodes.

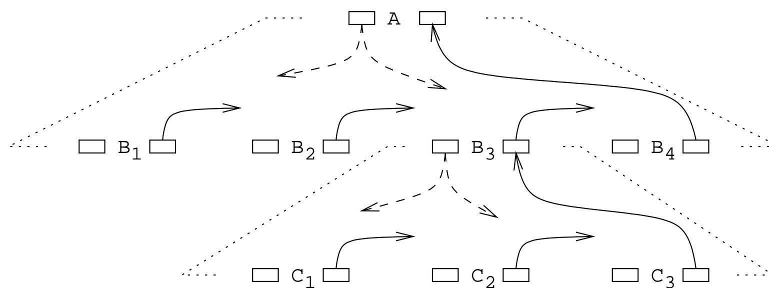


Figure 3.31 Data flow in part of a parse tree for an L-attributed grammar.

```
{
#include    "symbol_table.h"
}

%lexical get_next_token_class;
%token IDENTIFIER;
%token DIGIT;

%start Main_Program, main;

main {symbol_table sym_tab; int result;}:
    {init_symbol_table(&sym_tab);}
    declarations(sym_tab)
    expression(sym_tab, &result)
    {printf("result = %d\n", result);}
;

declarations(symbol_table sym_tab):
    declaration(sym_tab) [ ',' declaration(sym_tab) ]* ';'
;

declaration(symbol_table sym_tab) {symbol_entry *sym_ent;}:
    IDENTIFIER {sym_ent = look_up(sym_tab, Token.repr);}
    '=' DIGIT  {sym_ent->value = Token.repr - '0';}
;

expression(symbol_table sym_tab; int *e) {int t;}:
    term(sym_tab, &t)              {*e = t;}
    expression_tail_option(sym_tab, e)
;

expression_tail_option(symbol_table sym_tab; int *e) {int t;}:
    '-' term(sym_tab, &t)          {*e -= t;}
    expression_tail_option(sym_tab, e)
|
;

term(symbol_table sym_tab; int *t):
    IDENTIFIER      {*t = look_up(sym_tab, Token.repr)->value;}
;
```

Figure 3.32 *LLgen* code for an L-attributed grammar for simple expressions.

```
Declaration →
    Type_Declarator(type)  Declared_Idf_Sequence(type)  ';'
;

Declared_Idf_Sequence(INH type) →
    Declared_Idf(type)
|
    Declared_Idf_Sequence(type)  ','  Declared_Idf(type)
;

Declared_Idf(INH type) →
    Idf(repr)
    ATTRIBUTE RULES:
        Add to symbol table (repr, type);
;
```

Figure 3.33 Sketch of an L-attributed grammar for `Declaration`.

```
Declaration →
    Type_Declarator(type)  Declared_Idf_Sequence(repr list)  ';'
    ATTRIBUTE RULES:
        FOR EACH repr IN repr list:
            Add to symbol table(repr, type);

Declared_Idf_Sequence(SYN repr list) →
    Declared_Idf(repr)
    ATTRIBUTE RULES:
        SET repr list TO Convert to list (repr);
|
    Declared_Idf_Sequence(old repr list)  ','  Declared_Idf(repr)
    ATTRIBUTE RULES:
        SET repr list TO Append to list (old repr list, repr);
;

Declared_Idf(SYN repr) →
    Idf(repr)
;
```

Figure 3.34 Sketch of an S-attributed grammar for `Declaration`.

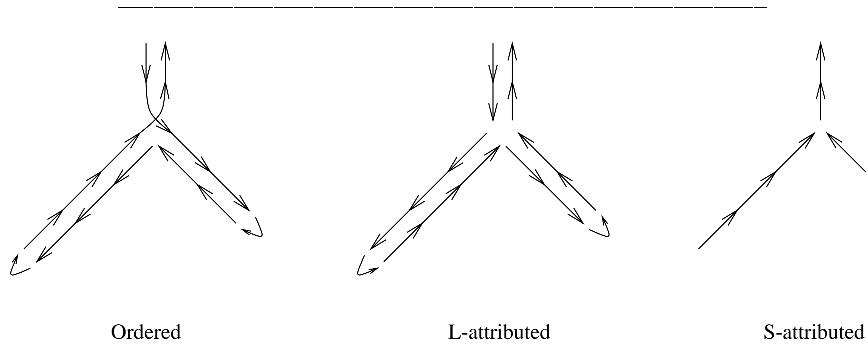
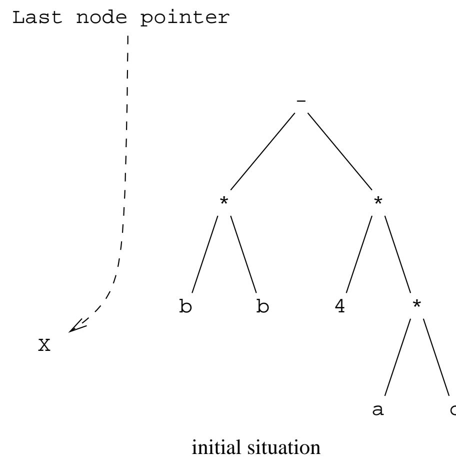
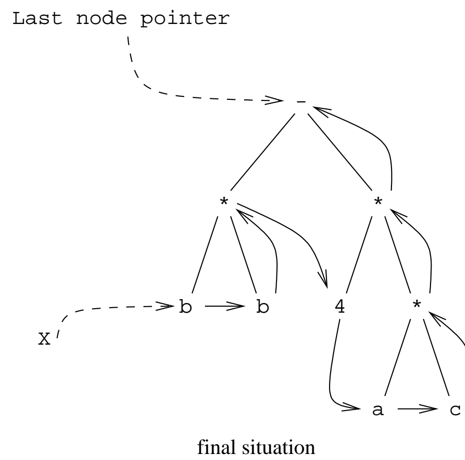


Figure 3.35 Pictorial comparison of three types of attribute grammars.



initial situation



final situation

Figure 3.36 Control flow graph for the expression `b*b - 4*a*c`.

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

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

void Thread_AST(AST_node *icode) {
    AST_node Dummy_node;

    Last_node = &Dummy_node; Thread_expression(icode);
    Last_node->successor = (AST_node *)0;
    Thread_start = Dummy_node.successor;
}
```

Figure 3.37 Threading code for the demo compiler from Section 1.2.

```
PROCEDURE Thread if statement (If node pointer):
    Thread expression (If node pointer .condition);
    SET Last node pointer .successor TO If node pointer;

    SET End if join node TO Generate join node ();

    SET Last node pointer TO address of a local node Aux last node;
    Thread block (If node pointer .then part);
    SET If node pointer .true successor TO Aux last node .successor;
    SET Last node pointer .successor TO address of End if join node;

    SET Last node pointer TO address of Aux last node;
    Thread block (If node pointer .else part);
    SET If node pointer .false successor TO Aux last node .successor;
    SET Last node pointer .successor TO address of End if join node;

    SET Last node pointer TO address of End if join node;
```

Figure 3.38 Sample threading routine for if-statements.

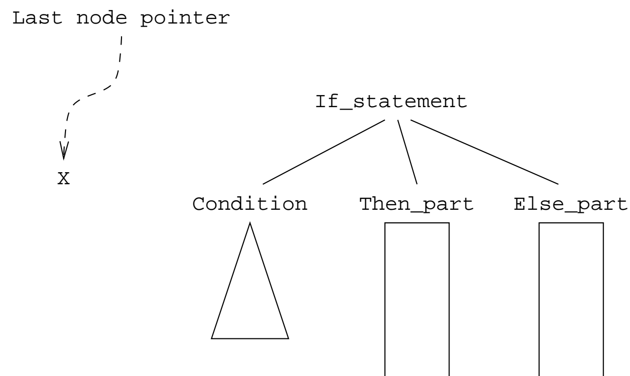


Figure 3.39 AST of an if-statement before threading.

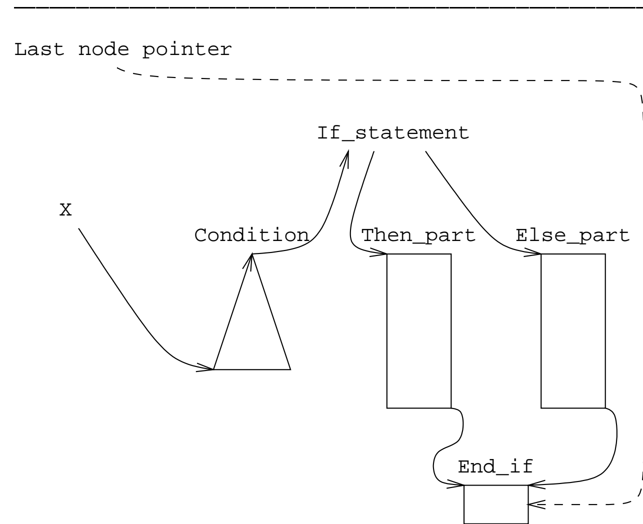


Figure 3.40 AST and control flow graph of an if-statement after threading.

```
If_statement(INH successor, SYN first) →
    'IF'  Condition  'THEN'  Then_part  'ELSE'  Else_part  'END'  'IF'
    ATTRIBUTE RULES:
        SET If_statement .first TO Condition .first;
        SET Condition .true successor TO Then_part .first;
        SET Condition .false successor TO Else_part .first;
        SET Then_part .successor TO If_statement .successor;
        SET Else_part .successor TO If_statement .successor;
```

Figure 3.41 Threading an if-statement using attribute rules.

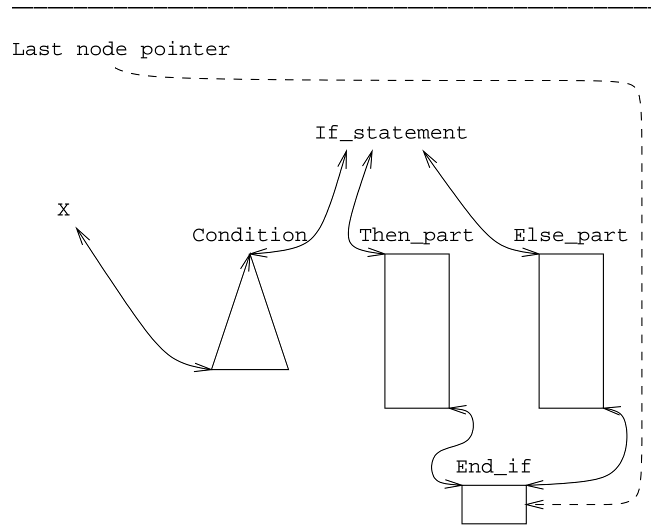


Figure 3.42 AST and doubly-linked control flow graph of an if-statement.

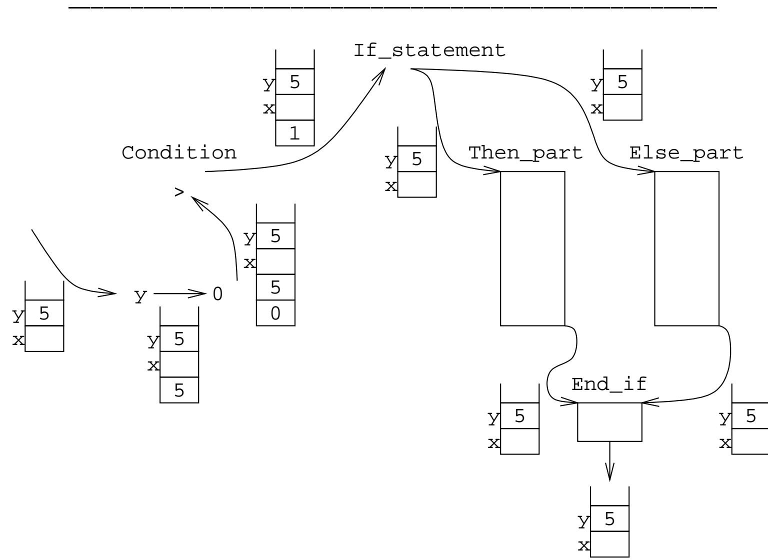


Figure 3.43 Stack representations in the control flow graph of an if-statement.

```
FUNCTION Symbolically interpret an if statement (
    Stack representation, If node
) RETURNING a stack representation:
    SET New stack representation TO
        Symbolically interpret a condition (
            Stack representation, If node .condition
        );
    Discard top entry from New stack representation;
    RETURN Merge stack representations (
        Symbolically interpret a statement sequence (
            New stack representation, If node .then part
        ),
        Symbolically interpret a statement sequence (
            New stack representation, If node .else part
        )
    );
```

Figure 3.44 Outline of a routine `Symbolically interpret an if statement`.

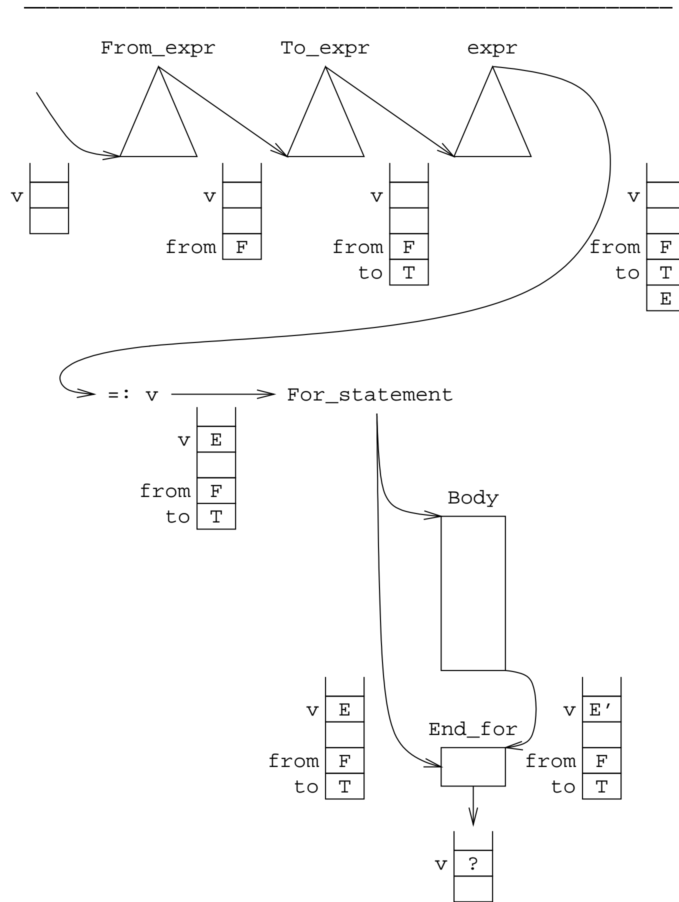


Figure 3.45 Stack representations in the control flow graph of a for-statement.

```
int i = 0;

while (some condition) {
    if (i > 0) printf("Loop reentered: i = %d\n", i);
    i++;
}
```

Figure 3.46 Value set analysis in the presence of a loop statement.

Data definitions:
Stack representations, with entries for every item we are interested in.

Initializations:
1. Empty stack representations are attached to all arrows in the control flow graph residing in the threaded AST.
2. Some stack representations at strategic points are initialized in accordance with properties of the source language; for example, the stack representations of input parameters are initialized to `Initialized`.

Inference rules:
For each node type, source language dependent rules allow inferences to be made, adding information to the stack representation on the outgoing arrows based on those on the incoming arrows and the node itself, and vice versa.

Figure 3.47 Full symbolic interpretation as a closure algorithm.

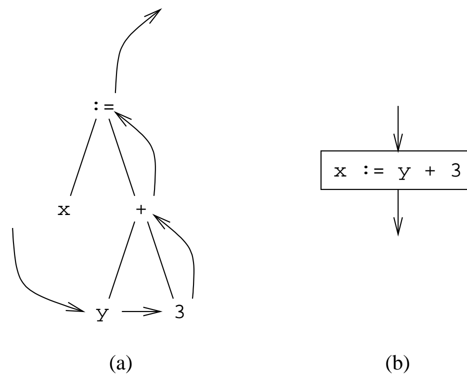


Figure 3.48 An assignment as a full control flow graph and as a single node.

$$IN(N) = \bigcup_{M=\text{dynamic predecessor of } N} OUT(M)$$

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

Figure 3.49 Data-flow equations for a node *N*.

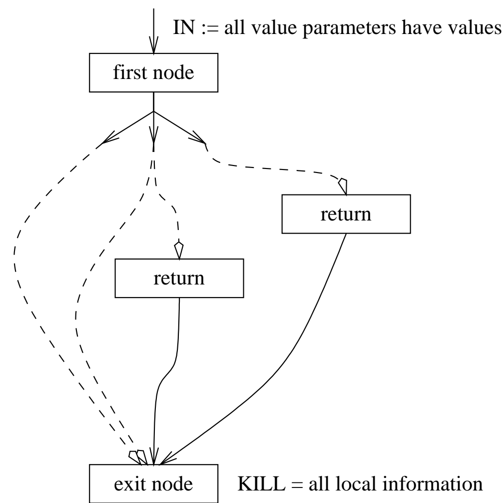


Figure 3.50 Data-flow details at routine entry and exit.

Data definitions:

1. Constant *KILL* and *GEN* sets for each node.
2. Variable *IN* and *OUT* sets for each node.

Initializations:

1. The *IN* set of the top node is initialized with information established externally.
2. For all other nodes N, $IN(N)$ and $OUT(N)$ are set to empty.

Inference rules:

1. For any node N, $IN(N)$ should contain

$$\bigcup_{M=dynamic\ predecessor\ of\ N} OUT(M).$$

2. For any node N, $OUT(N)$ should contain

$$(IN(N) \setminus KILL(N)) \cup GEN(N).$$

Figure 3.51 Closure algorithm for solving the data-flow equations.

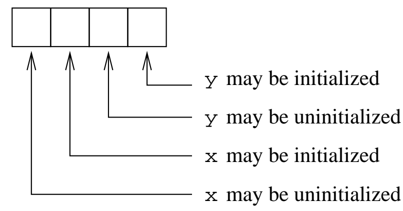


Figure 3.52 Bit patterns for properties of the variables x and y.

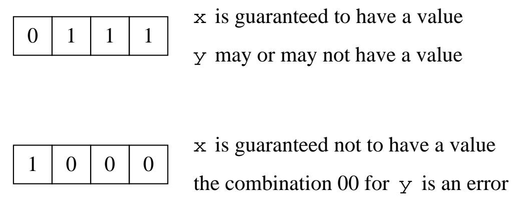


Figure 3.53 Examples of bit patterns for properties of the variables x and y.

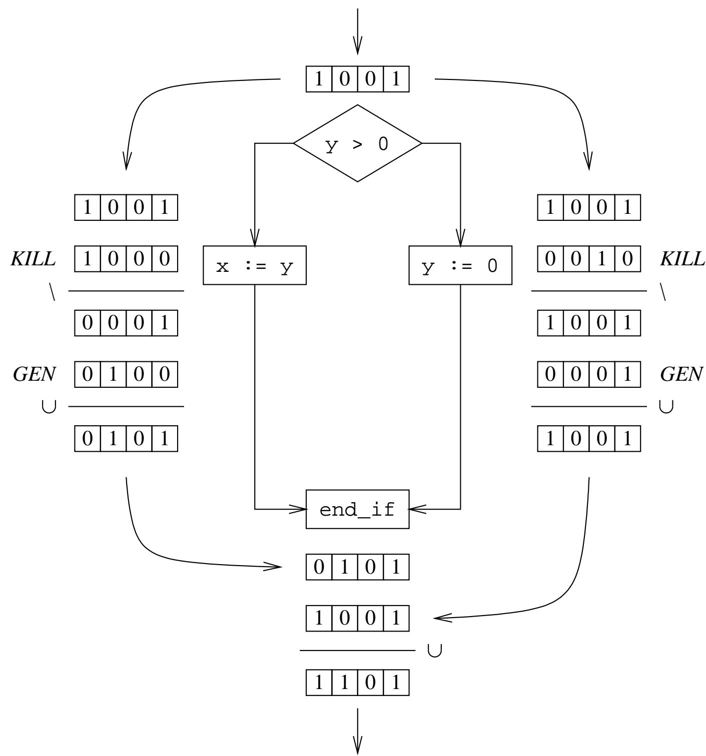


Figure 3.54 Data-flow propagation through an if-statement.

```
{   int x;
    x = ...;              /* code fragment  1, does not use x */
    if (...) {
        ...               /* code fragment  2, does not use x */
        print(x);         /* code fragment  3, uses x */
        ...               /* code fragment  4, does not use x */
    } else {
        int y;
        ...               /* code fragment  5, does not use x,y */
        print(x);         /* code fragment  6, uses x, but not y */
        ...               /* code fragment  7, does not use x,y */
        y = ...;          /* code fragment  8, does not use x,y */
        ...               /* code fragment  9, does not use x,y */
        print(y);         /* code fragment 10, uses y but not x */
        ...               /* code fragment 11, does not use x,y */
    }
    x = ...;              /* code fragment 12, does not use x */
    ...                   /* code fragment 13, does not use x */
    print(x);             /* code fragment 14, uses x */
    ...                   /* code fragment 15, does not use x */
}
```

Figure 3.55 A segment of C code to demonstrate live analysis.

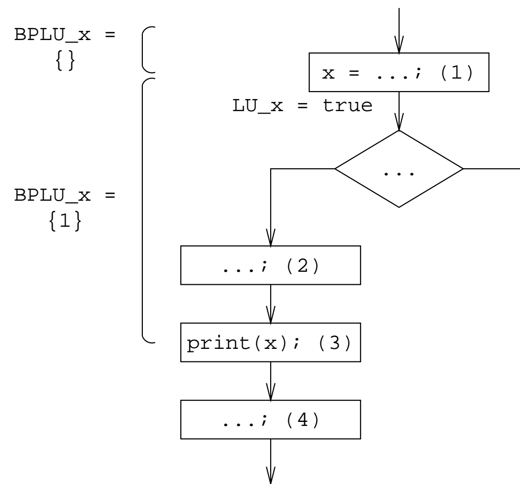


Figure 3.56 The first few steps in live analysis for Figure 3.55 using backpatch lists.

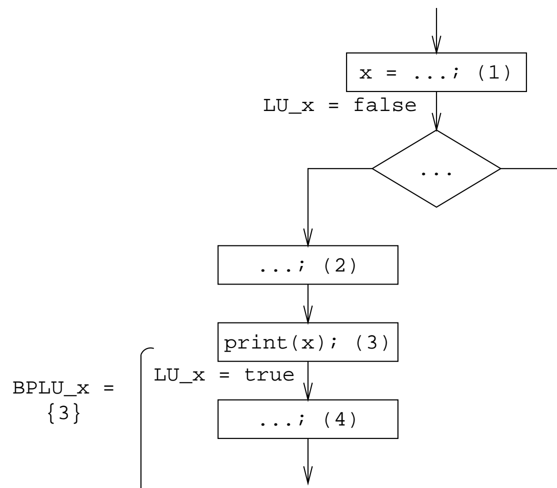


Figure 3.57 Live analysis for Figure 3.55, after a few steps.

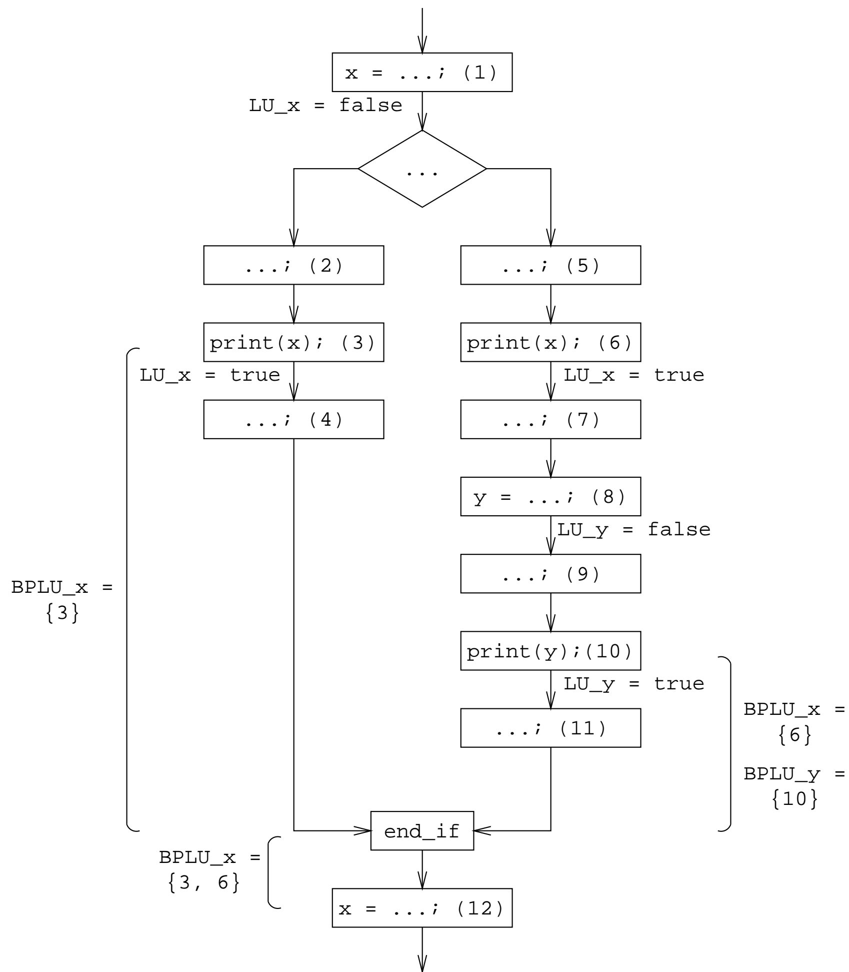


Figure 3.58 Live analysis for Figure 3.55, merging at the end-if node.

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

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

Figure 3.59 Backwards data-flow equations for a node *N*.

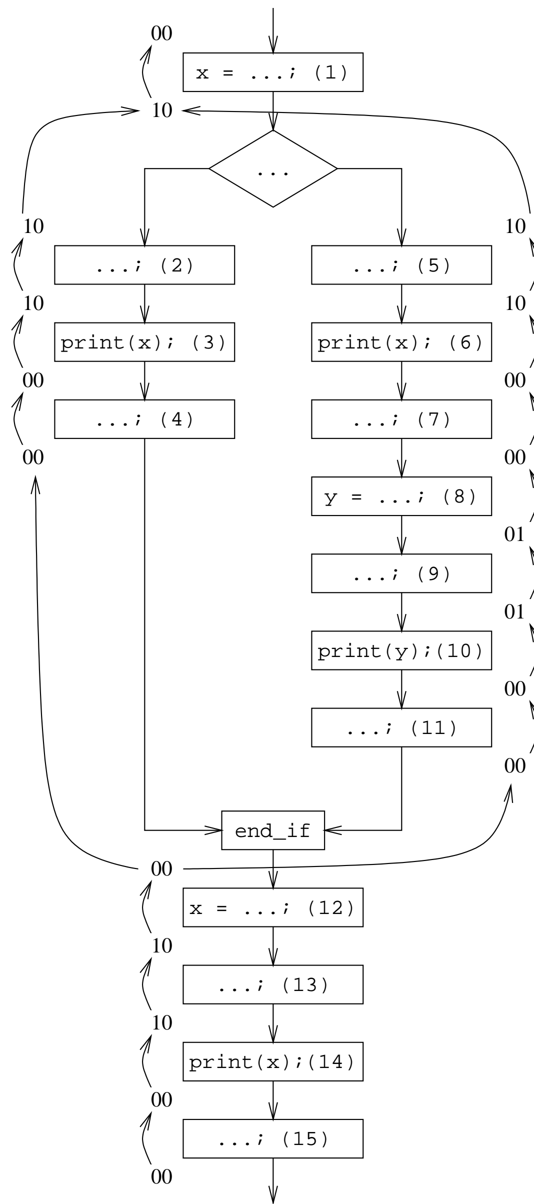


Figure 3.60 Live analysis for Figure 3.55 using backward data-flow equations.

```
S(SYN s) →
    A(i1, s1)
    ATTRIBUTE RULES:
        SET i1 TO s1;
        SET s TO s1;

A(INH i1, SYN s1) →
    A(i2, s2)  'a'
    ATTRIBUTE RULES:
        SET i2 TO i1;
        SET s1 TO s2;
|
    B(i2, s2)
    ATTRIBUTE RULES:
        SET i2 TO i1;
        SET s1 TO s2;

B(INH i, SYN s) →
    'b'
    ATTRIBUTE RULES: SET s TO i;
```

Figure 3.61 Attribute grammar for Exercise {#ShowCycle}.

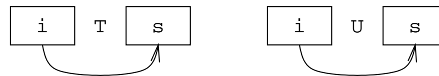


Figure 3.62 IS-SI graphs for T and U.

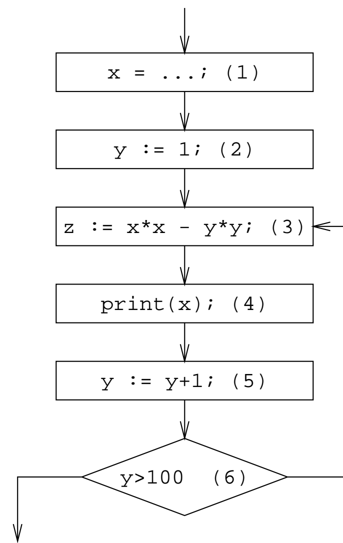


Figure 3.63 Example program for a very busy expression.

4

Processing the intermediate code

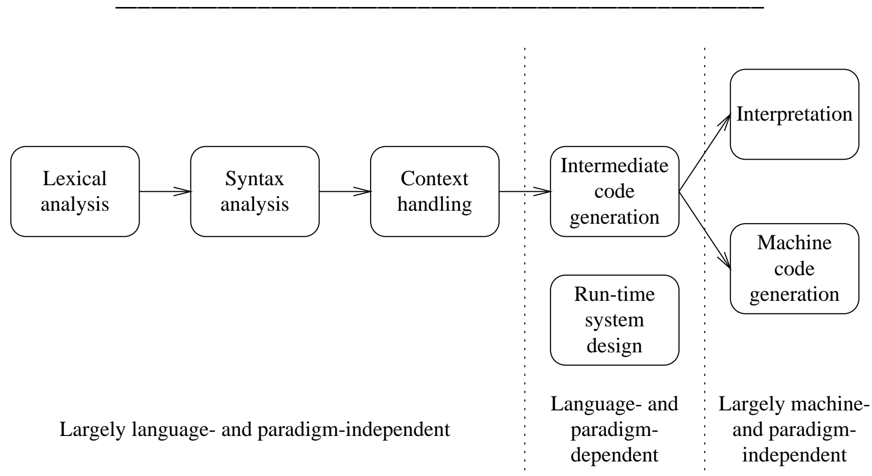


Figure 4.1 The status of the various modules in compiler construction.

`re:`	`3.0`
`im:`	`4.0`

Figure 4.2 A value of type `Complex_Number` as the programmer sees it.

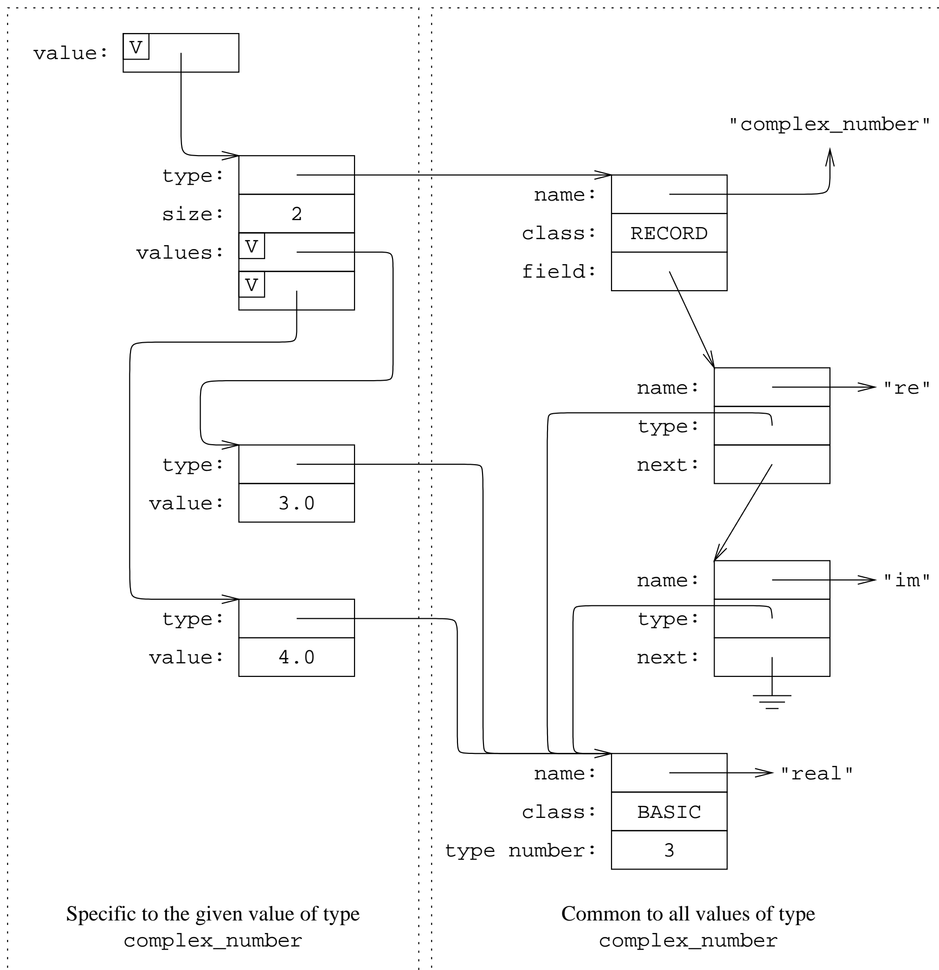


Figure 4.3 A representation of the value of Figure 4.2 in a recursive interpreter.

```
PROCEDURE Elaborate if statement (If node):
    SET Result TO Evaluate condition (If node .condition);
    IF Status .mode /= Normal mode: RETURN;
    IF Result .type /= Boolean:
        ERROR "Condition in if-statement is not of type Boolean";
        RETURN;
    IF Result .boolean .value = True:
        Elaborate statement (If node .then part);
    ELSE Result .boolean .value = False:
        // Check if there is an else-part at all:
        IF If node .else part /= No node:
            Elaborate statement (If node .else part);
        ELSE If node .else part = No node:
            SET Status .mode TO Normal mode;
```

Figure 4.4 Outline of a routine for recursively interpreting an if-statement.

```
WHILE Active node .type /= End of program type:
    SELECT Active node .type:
        CASE ...
        CASE If type:
            // We arrive here after the condition has been evaluated;
            // the Boolean result is on the working stack.
            SET Value TO Pop working stack ();
            IF Value .boolean .value = True:
                SET Active node TO Active node .true successor;
            ELSE Value .boolean .value = False:
                IF Active node .false successor /= No node:
                    SET Active node TO Active node .false successor;
                ELSE Active node .false successor = No node:
                    SET Active node TO Active node .successor;
        CASE ...
```

Figure 4.5 Sketch of the main loop of an iterative interpreter, showing the code for an if-statement.

```
#include    "parser.h"      /* for types AST_node and Expression */
#include    "thread.h"      /* for Thread_AST() and Thread_start */
#include    "stack.h"       /* for Push() and Pop() */
#include    "backend.h"     /* for self check */
                                        /* PRIVATE */
static AST_node *Active_node_pointer;

static void Interpret_iteratively(void) {
    while (Active_node_pointer != 0) {
        /* there is only one node type, Expression: */
        Expression *expr = Active_node_pointer;
        switch (expr->type) {
        case 'D':
            Push(expr->value);
            break;
        case 'P': {
            int e_left = Pop(); int e_right = Pop();
            switch (expr->oper) {
            case '+': Push(e_left + e_right); break;
            case '*': Push(e_left * e_right); break;
            }}
            break;
        }
        Active_node_pointer = Active_node_pointer->successor;
    }
    printf("%d\n", Pop());      /* print the result */
}
                                        /* PUBLIC */
void Process(AST_node *icode) {
    Thread_AST(icode); Active_node_pointer = Thread_start;
    Interpret_iteratively();
}
```

Figure 4.6 An iterative interpreter for the demo compiler of Section 1.2.

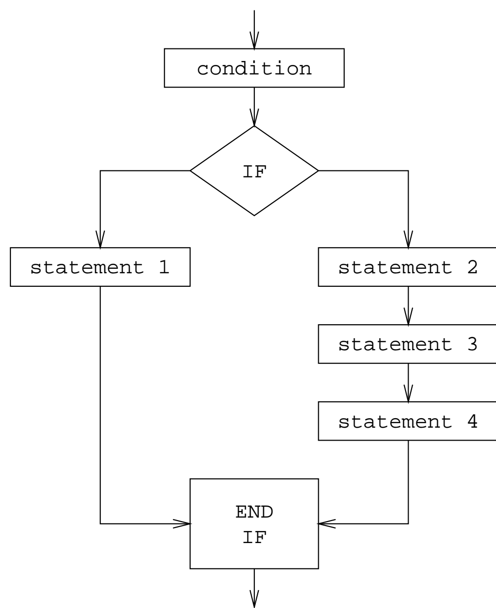


Figure 4.7 An AST stored as a graph.

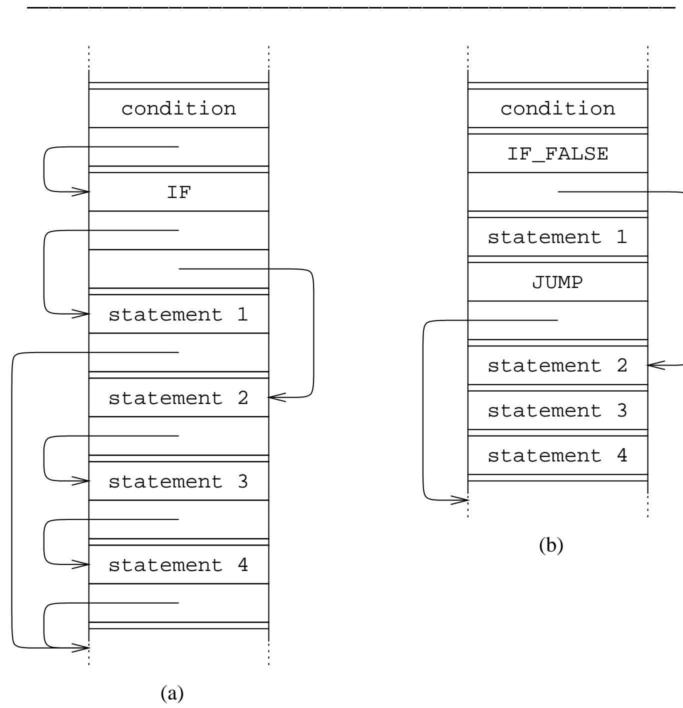


Figure 4.8 Storing the AST in an array (a) and as pseudo-instructions (b).

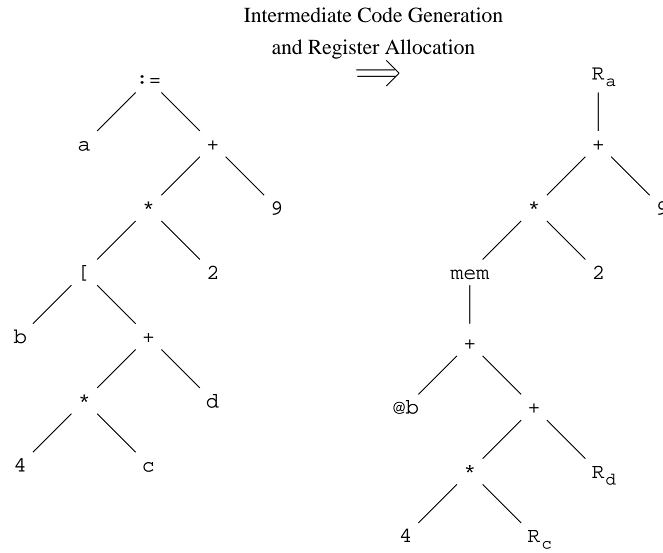


Figure 4.9 Two ASTs for the expression `a := (b[4*c + d] * 2) + 9.`

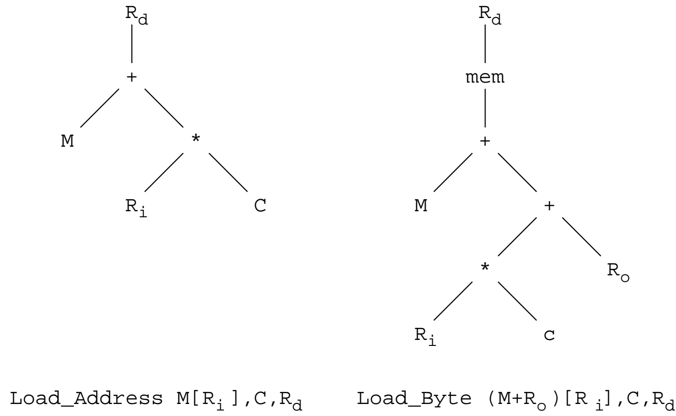


Figure 4.10 Two sample instructions with their ASTs.

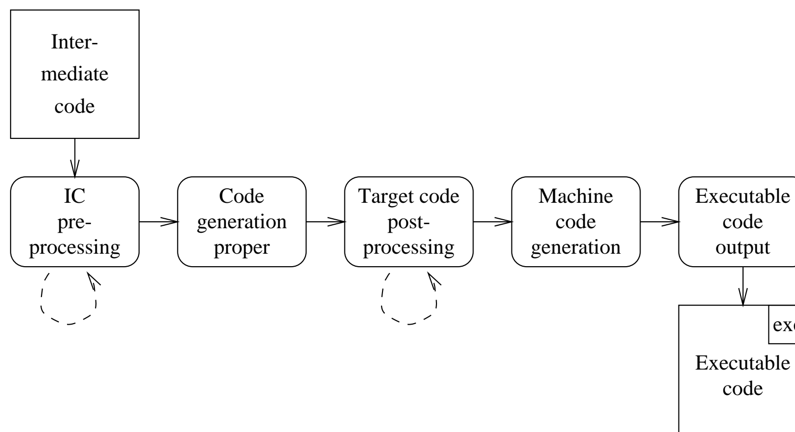


Figure 4.11 Overall structure of a code generator.

```
#include    "parser.h"      /* for types AST_node and Expression */
#include    "thread.h"      /* for Thread_AST() and Thread_start */
#include    "backend.h"     /* for self check */
                                        /* PRIVATE */
static AST_node *Active_node_pointer;

static void Trivial_code_generation(void) {
    printf("#include    \"stack.h\"\nint main(void) {\n");
    while (Active_node_pointer != 0) {
        /* there is only one node type, Expression: */
        Expression *expr = Active_node_pointer;
        switch (expr->type) {
        case 'D':
            printf("Push(%d);\n", expr->value);
            break;
        case 'P':
            printf("{\n\
                int e_left = Pop(); int e_right = Pop();\n\
                switch (%d) {\n\
                case '+': Push(e_left + e_right); break;\n\
                case '*': Push(e_left * e_right); break;\n\
                }}\n",
                expr->oper
            );
            break;
        }
        Active_node_pointer = Active_node_pointer->successor;
    }
    printf("printf(\"%%d\\n\", Pop()); /* print the result */\n");
    printf("return 0;}\n");
}
                                        /* PUBLIC */
void Process(AST_node *icode) {
    Thread_AST(icode); Active_node_pointer = Thread_start;
    Trivial_code_generation();
}
```

Figure 4.12 A trivial code generator for the demo compiler of Section 1.2.

```
#include    "stack.h"
int main(void) {
Push(7);
Push(1);
Push(5);
{
    int e_left = Pop(); int e_right = Pop();
    switch (43) {
    case '+': Push(e_left + e_right); break;
    case '*': Push(e_left * e_right); break;
    }}
{
    int e_left = Pop(); int e_right = Pop();
    switch (42) {
    case '+': Push(e_left + e_right); break;
    case '*': Push(e_left * e_right); break;
    }}
printf("%d\n", Pop()); /* print the result */
return 0;}
```

Figure 4.13 Code for `(7*(1+5))` generated by the code generator of Figure 4.12.

```
int main(void) {
    Expression_D(7);
    Expression_D(1);
    Expression_D(5);
    Expression_P(43);  /* 43 = ASCII value of '+' */
    Expression_P(42);  /* 42 = ASCII value of '*' */
    Print();
    return 0;
}
```

Figure 4.14 Possible threaded code for `(7*(1+5))`.

```
#include    "stack.h"

void Expression_D(int digit) {
    Push(digit);
}

void Expression_P(int oper) {
    int e_left = Pop(); int e_right = Pop();
    switch (oper) {
    case '+': Push(e_left + e_right); break;
    case '*': Push(e_left * e_right); break;
    }
}

void Print(void) {
    printf("%d\n", Pop());
}
```

Figure 4.15 Routines for the threaded code for `(7*(1+5))`.

```
case 'P':
    printf("{\n\
        int e_left = Pop(); int e_right = Pop();\n"
    );
    switch (expr->oper) {
    case '+': printf("Push(e_left + e_right);\n"); break;
    case '*': printf("Push(e_left * e_right);\n"); break;
    }
    printf("}\n");
    break;
```

Figure 4.16 Partial evaluation in a segment of the code generator.

```
#include    "stack.h"
int main(void) {
Push(7);
Push(1);
Push(5);
{int e_left = Pop(); int e_right = Pop(); Push(e_left + e_right);}
{int e_left = Pop(); int e_right = Pop(); Push(e_left * e_right);}
printf("%d\n", Pop()); /* print the result */
return 0;}
```

Figure 4.17 Code for `(7*(1+5))` generated by the code generator of Figure 4.16.

```
case 'P':
    printf("{\n\
        int e_left = Pop(); int e_right = Pop();\n"
    );
    switch (expr->oper) {
    case '+': printf("Push(e_left + e_right);\n"); break;
    case '*': printf("Push(e_left * e_right);\n"); break;
    }
    printf("}\n");
    break;
```

Figure 4.18 Foreground (run-now) view of partially evaluating code.

```
case 'P':
    printf("{\n\
        int e_left = Pop(); int e_right = Pop();\n"
    );
    switch (expr->oper) {
    case '+': printf("Push(e_left + e_right);\n"); break;
    case '*': printf("Push(e_left * e_right);\n"); break;
    }
    printf("}\n");
    break;
```

Figure 4.19 Background (run-later) view of partially evaluating code.

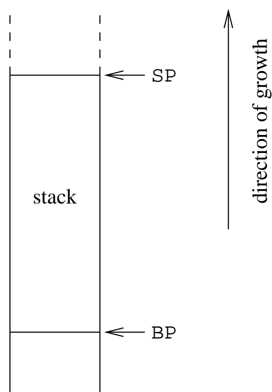


Figure 4.20 Data administration in a simple stack machine.

Instruction		Actions
`Push_Const`	c	`SP:=SP+1; stack[SP]:=`c`;`
`Push_Local`	i	`SP:=SP+1; stack[SP]:=stack[BP+`i`];`
`Store_Local`	i	`stack[BP+`i`]:=stack[SP]; SP:=SP-1;`
`Add_Top2`		`stack[SP-1]:=stack[SP-1]+stack[SP]; SP:=SP-1;`
`Subtr_Top2`		`stack[SP-1]:=stack[SP-1]-stack[SP]; SP:=SP-1;`
`Mult_Top2`		`stack[SP-1]:=stack[SP-1]*stack[SP]; SP:=SP-1;`

Figure 4.21 Stack machine instructions.

Instruction		Actions
`Load_Const`	c, R_n	R_n `:=`c`;`
`Load_Mem`	x, R_n	R_n `:=`x`;`
`Store_Reg`	R_n, x	x`:=`R_n`;`
`Add_Reg`	R_m, R_n	R_n`:=`R_n`+`R_m`;`
`Subtr_Reg`	R_m, R_n	R_n`:=`R_n`-`R_m`;`
`Mult_Reg`	R_m, R_n	R_n`:=`R_n`*`R_m`;`

Figure 4.22 Register machine instructions.

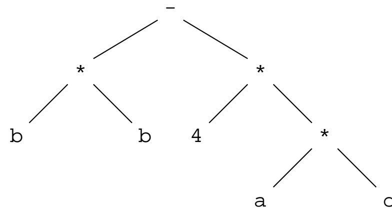


Figure 4.23 The abstract syntax tree for `b*b - 4*(a*c)`.

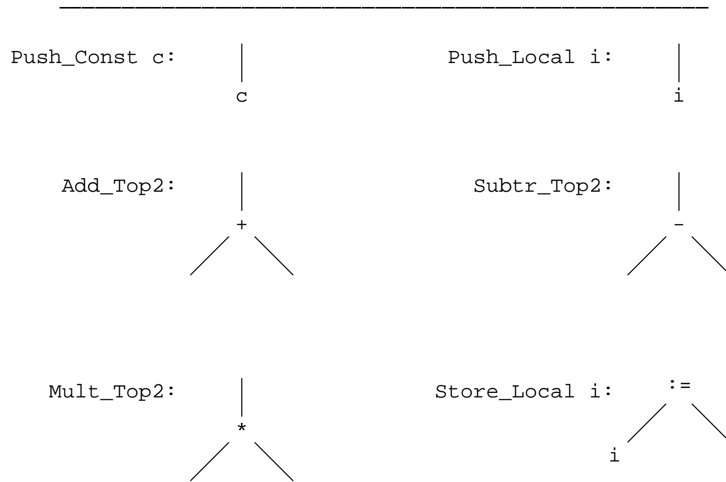


Figure 4.24 The abstract syntax trees for the stack machine instructions.

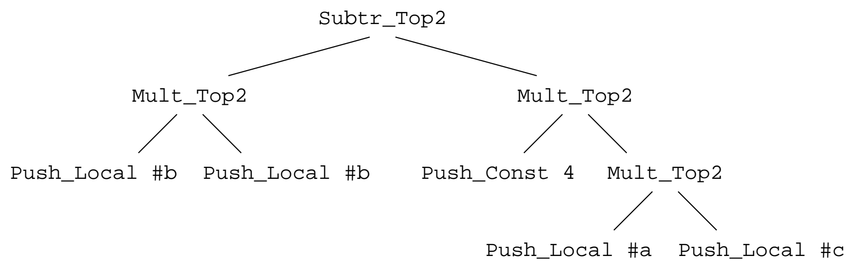


Figure 4.25 The abstract syntax tree for `b*b - 4*(a*c)` rewritten.

```
PROCEDURE Generate code (Node):
    SELECT Node .type:
        CASE Constant type:   Emit ("Push_Const" Node .value);
        CASE LocalVar type:   Emit ("Push_Local" Node .number);
        CASE StoreLocal type: Emit ("Store_Local" Node .number);
        CASE Add type:
            Generate code (Node .left); Generate code (Node .right);
            Emit ("Add_Top2");
        CASE Subtract type:
            Generate code (Node .left); Generate code (Node .right);
            Emit ("Subtr_Top2");
        CASE Multiply type:
            Generate code (Node .left); Generate code (Node .right);
            Emit ("Mult_Top2");
```

Figure 4.26 Depth-first code generation for a stack machine.

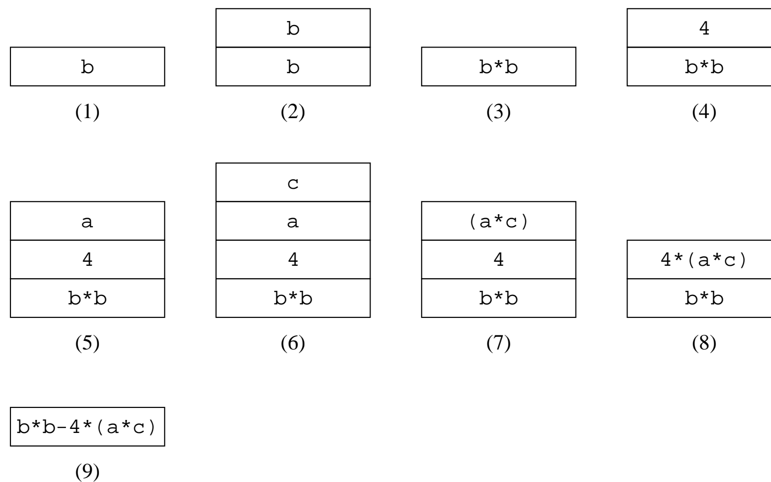


Figure 4.27 Successive stack configurations for `b*b - 4*(a*c)`.

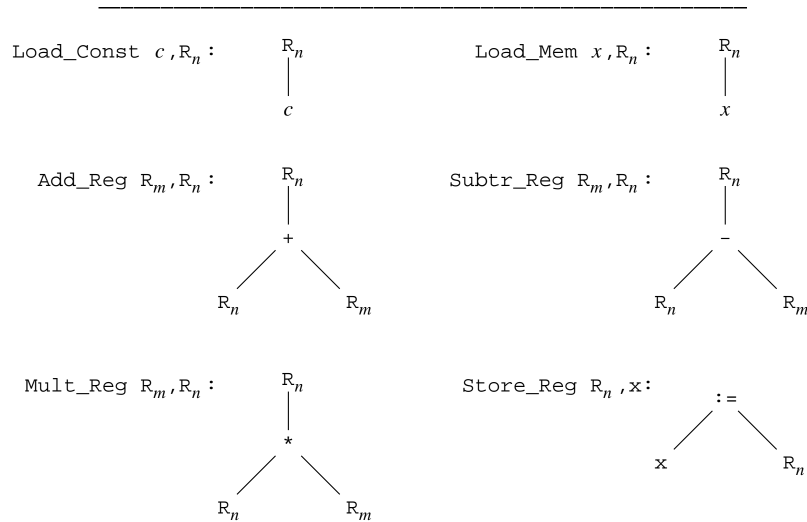


Figure 4.28 The abstract syntax trees for the register machine instructions.

```
PROCEDURE Generate code (Node, a register Target, a register set Aux):
    SELECT Node .type:
        CASE Constant type:
            Emit ("Load_Const " Node .value ",R" Target);
        CASE Variable type:
            Emit ("Load_Mem " Node .address ",R" Target);
        CASE ...
        CASE Add type:
            Generate code (Node .left, Target, Aux);
            SET Target 2 TO An arbitrary element of Aux;
            SET Aux 2 TO Aux \ Target 2;
                // the \ denotes the set difference operation
            Generate code (Node .right, Target 2, Aux 2);
            Emit ("Add_Reg R" Target 2 ",R" Target);
        CASE ...
```

Figure 4.29 Simple code generation with register allocation.

```
PROCEDURE Generate code (Node, a register number Target):
    SELECT Node .type:
        CASE Constant type:
            Emit ("Load_Const " Node .value ",R" Target);
        CASE Variable type:
            Emit ("Load_Mem " Node .address ",R" Target);
        CASE ...
        CASE Add type:
            Generate code (Node .left, Target);
            Generate code (Node .right, Target+1);
            Emit ("Add_Reg R" Target+1 ",R" Target);
        CASE ...
```

Figure 4.30 Simple code generation with register numbering.

```
Load_Mem     b,R1
Load_Mem     b,R2
Mult_Reg     R2,R1
Load_Const   4,R2
Load_Mem     a,R3
Load_Mem     c,R4
Mult_Reg     R4,R3
Mult_Reg     R3,R2
Subtr_Reg    R2,R1
```

Figure 4.31 Register machine code for the expression `b*b - 4*(a*c)`.

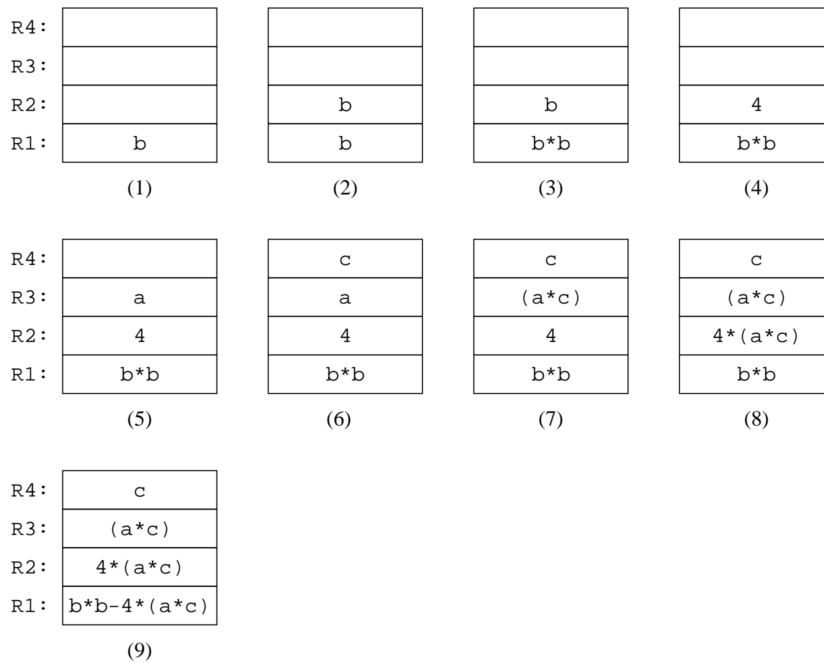


Figure 4.32 Successive register contents for `b*b - 4*(a*c)`.

```
Load_Mem     b,R1
Load_Mem     b,R2
Mult_Reg     R2,R1
Load_Mem     a,R2
Load_Mem     c,R3
Mult_Reg     R3,R2
Load_Const   4,R3
Mult_Reg     R3,R2
Subtr_Reg    R2,R1
```

Figure 4.33 Weighted register machine code for the expression `b*b - 4*(a*c)`.

```
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 ...
```

Figure 4.34 Register requirements (weight) of a node.

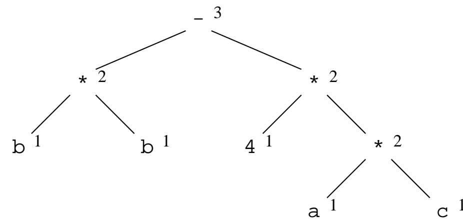


Figure 4.35 AST for `b*b - 4*(a*c)` with register weights.

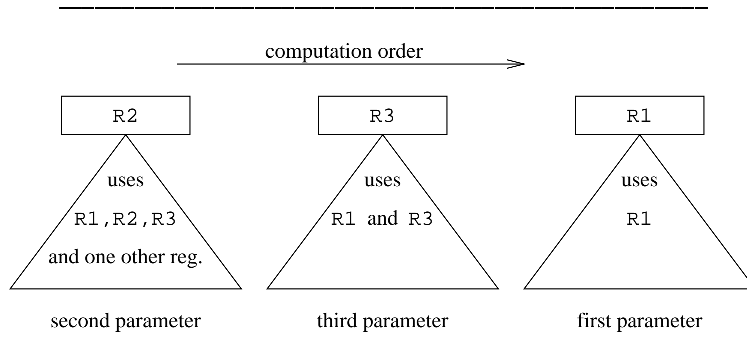


Figure 4.36 Evaluation order of three parameter trees.

```
PROCEDURE Generate code for large trees (Node, Target register):
    SET Auxiliary register set TO
        Available register set \ Target register;

    WHILE Node /= No node:
        Compute the weights of all nodes of the tree of Node;
        SET Tree node TO Maximal non_large tree (Node);
        Generate code
            (Tree node, Target register, Auxiliary register set);

        IF Tree node /= Node:
            SET Temporary location TO Next free temporary location();
            Emit ("Store R" Target register ",T" Temporary location);
            Replace Tree node by a reference to Temporary location;
            Return any temporary locations in the tree of Tree node
                to the pool of free temporary locations;
        ELSE Tree node = Node:
            Return any temporary locations in the tree of Node
                to the pool of free temporary locations;
            SET Node TO No node;

FUNCTION Maximal non_large tree (Node) RETURNING a node:
    IF Node .weight <= Size of Auxiliary register set: RETURN Node;
    IF Node .left .weight > Size of Auxiliary register set:
        RETURN Maximal non_large tree (Node .left);
    ELSE Node .right .weight >= Size of Auxiliary register set:
        RETURN Maximal non_large tree (Node .right);
```

Figure 4.37 Code generation for large trees.

```
Load_Mem     a,R1
Load_Mem     c,R2
Mult_Reg     R2,R1
Load_Const   4,R2
Mult_Reg     R2,R1
Store_Reg    R1,T1
Load_Mem     b,R1
Load_Mem     b,R2
Mult_Reg     R2,R1
Load_Mem     T1,R2
Subtr_Reg    R2,R1
```

Figure 4.38 Code generated for `b*b - 4*(a*c)` with only 2 registers.

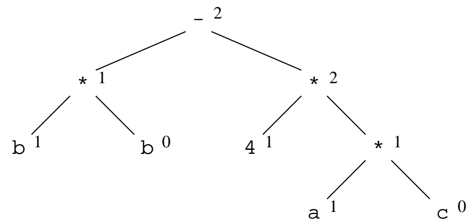


Figure 4.39 Register-weighted tree for a memory-register machine.

```
Load_Const   4,R2
Load_Mem     a,R1
Mult_Mem     c,R1
Mult_Reg     R1,R2
Load_Mem     b,R1
Mult_Mem     b,R1
Subtr_Reg    R2,R1
```

Figure 4.40 Code for the register-weighted tree for a memory-register machine.

```
{   int n;

    n = a + 1;
    x = b + n*n + c;
    n = n + 1;
    y = d * n;
}
```

Figure 4.41 Sample basic block in C.

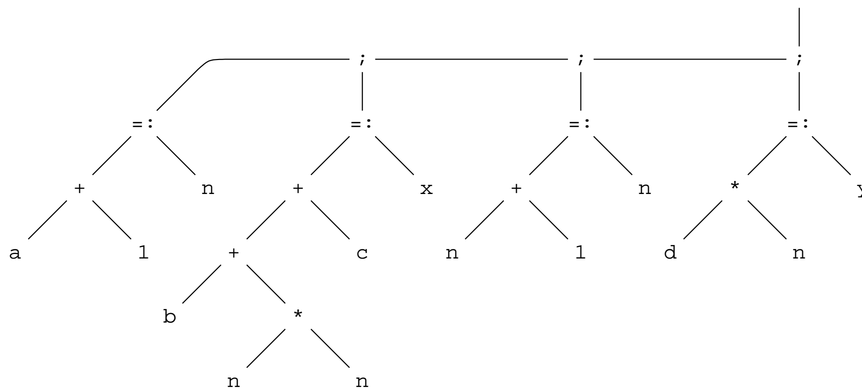


Figure 4.42 AST of the sample basic block.

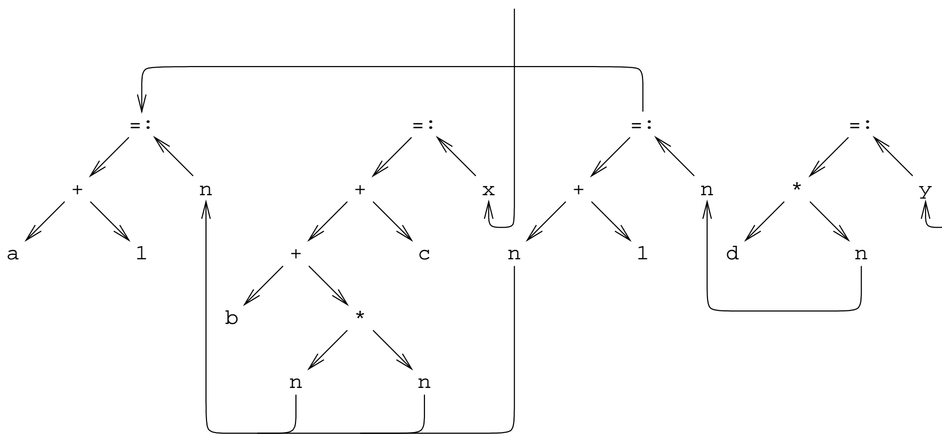


Figure 4.43 Data dependency graph for the sample basic block.

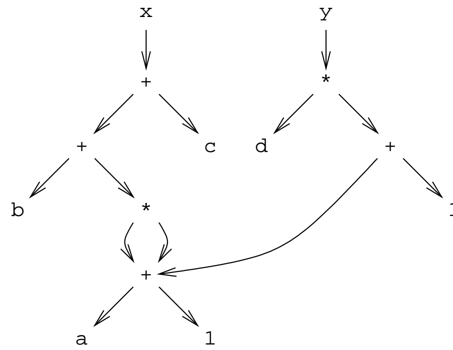


Figure 4.44 Cleaned-up data dependency graph for the sample basic block.

```
{   int n;

    n = a + 1;
    x = b + n*n + c;     /* subexpression n*n ... */
    n = n*n + 1;         /* ... in common */
    y = d * n;
}
```

Figure 4.45 Basic block in C with common subexpression.

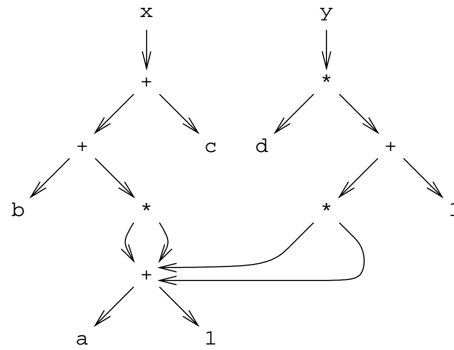


Figure 4.46 Data dependency graph with common subexpression.

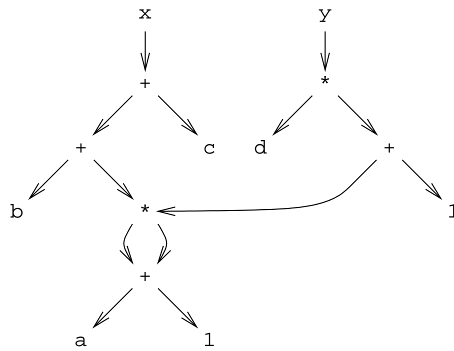


Figure 4.47 Cleaned-up data dependency graph with common subexpression eliminated.

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

Figure 4.48 The data dependency graph of Figure 4.44 as an array of triples.

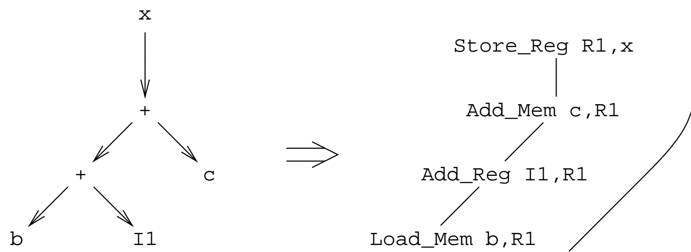


Figure 4.49 Rewriting and ordering a ladder sequence.

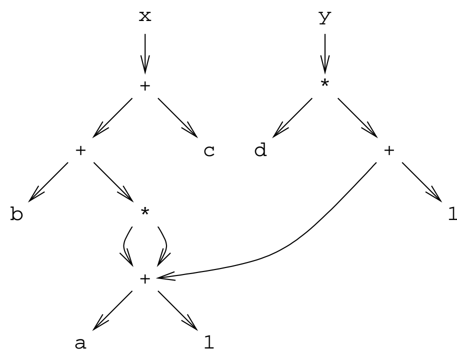


Figure 4.50 Cleaned-up data dependency graph for the sample basic block.

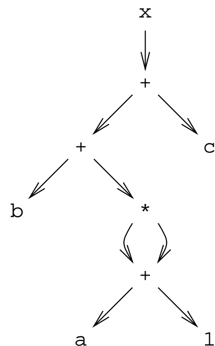


Figure 4.51 Data dependency graph after removal of the first ladder sequence.

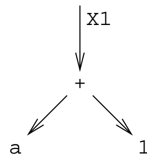


Figure 4.52 Data dependency graph after removal of the second ladder sequence.

```
Load_Mem    a,R1
Add_Const   1,R1
Load_Reg    R1,X1

Load_Reg    X1,R1
Mult_Reg    X1,R1
Add_Mem     b,R1
Add_Mem     c,R1
Store_Reg   R1,x

Load_Reg    X1,R1
Add_Const   1,R1
Mult_Mem    d,R1
Store_Reg   R1,y
```

Figure 4.53 Pseudo-register target code generated for the basic block.

```
Load_Mem    a,R1
Add_Const   1,R1
Load_Reg    R1,R2

Load_Reg    R2,R1
Mult_Reg    R2,R1
Add_Mem     b,R1
Add_Mem     c,R1
Store_Reg   R1,x

Load_Reg    R2,R1
Add_Const   1,R1
Mult_Mem    d,R1
Store_Reg   R1,y
```

Figure 4.54 Code generated for the program segment of Figure 4.41.

```
Load_Mem    a,R1
Add_Const   1,R1
Load_Reg    R1,R2

Mult_Reg    R1,R2
Add_Mem     b,R2
Add_Mem     c,R2
Store_Reg   R2,x

Add_Const   1,R1
Mult_Mem    d,R1
Store_Reg   R1,y
```

Figure 4.55 Code generated by the GNU C compiler, *gcc*.

```
{   int n;

    n = a + 1;
    *x = b + n*n + c;
    n = n + 1;
    y = d * n;
}
```

Figure 4.56 Sample basic block with an assignment under a pointer.

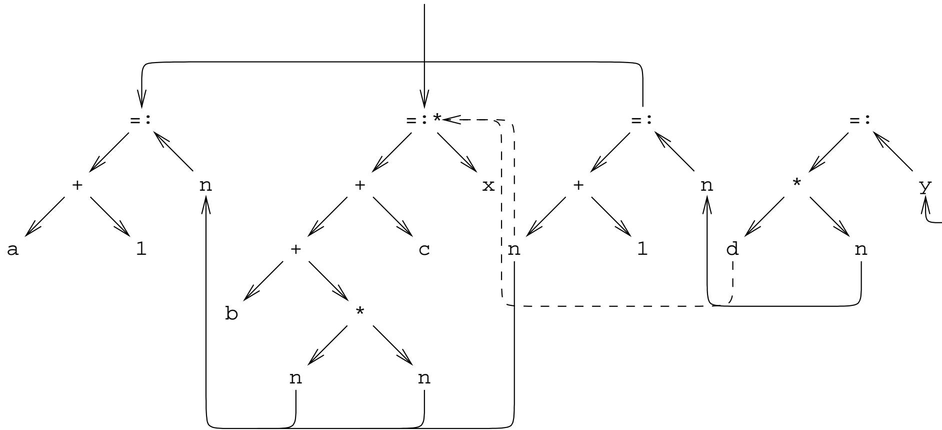


Figure 4.57 Data dependency graph with an assignment under a pointer.

Figure 4.58 Cleaned-up data dependency graph with an assignment under a pointer.

```
Load_Mem            a,R1
Add_Const           1,R1
Store_Reg           R1,n

Load_Mem            n,R1
Mult_Mem            n,R1
Add_Mem             b,R1
Add_Mem             c,R1
Store_Indirect_Mem  R1,x

Load_Mem            n,R1
Add_Const           1,R1
Mult_Mem            d,R1
Store_Reg           R1,y
```

Figure 4.59 Target code generated for the basic block of Figure 4.56.

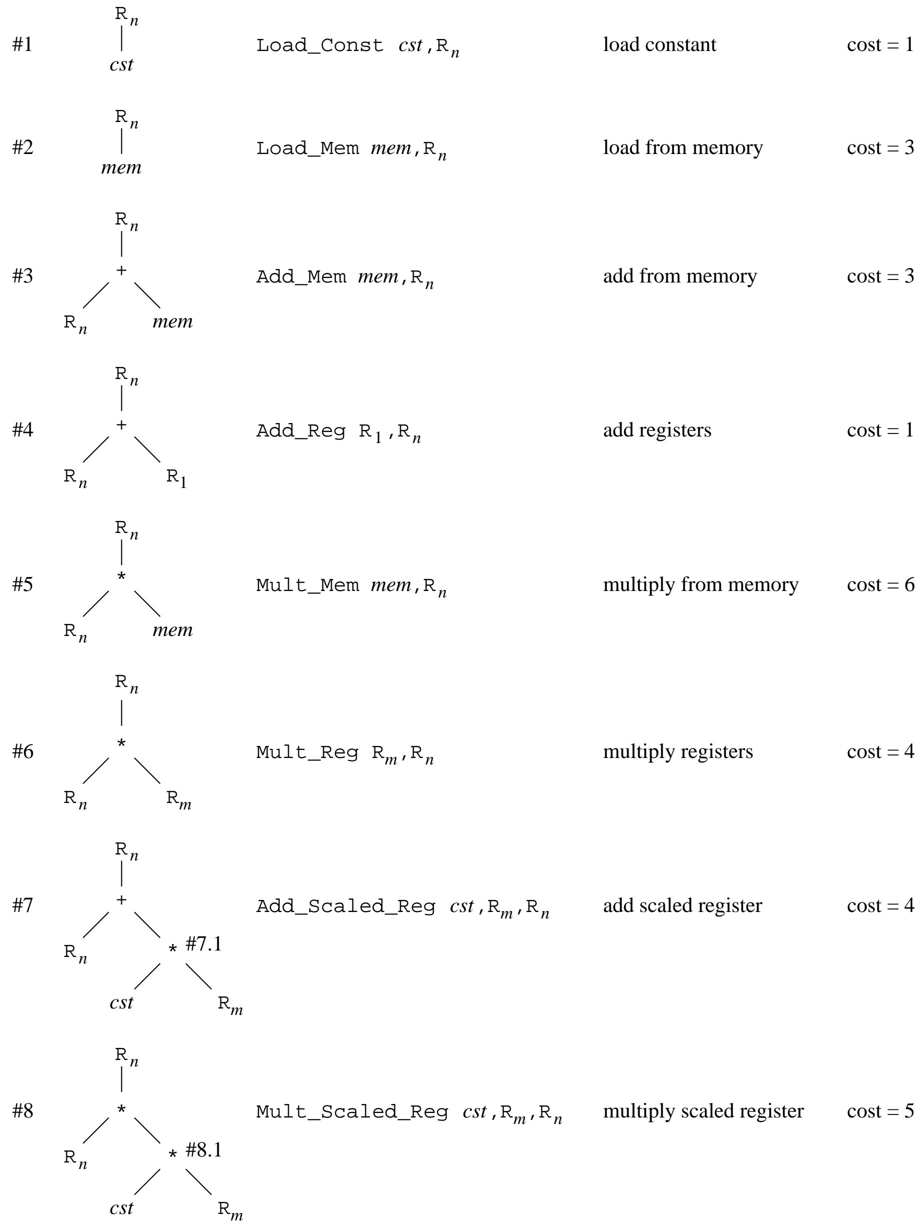


Figure 4.60 Sample instruction patterns for BURS code generation.

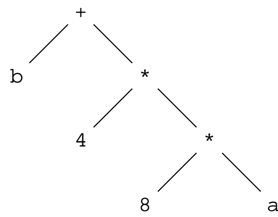


Figure 4.61 Input tree for the BURS code generation.

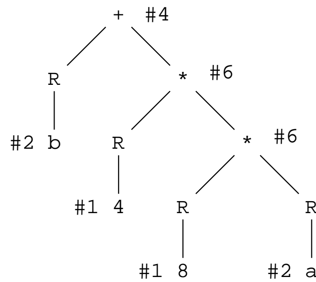


Figure 4.62 Naive rewrite of the input tree.

```
Load_Const   8,R1    ; 1 unit
Load_Mem     a,R2    ; 3 units
Mult_Reg     R2,R1   ; 4 units
Load_Const   4,R2    ; 1 unit
Mult_Reg     R1,R2   ; 4 units
Load_Mem     b,R1    ; 3 units
Add_Reg      R2,R1   ; 1 unit
             Total   = 17 units
```

Figure 4.63 Code resulting from the naive rewrite.

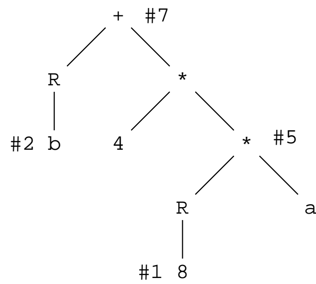


Figure 4.64 Top-down largest-fit rewrite of the input tree.

```
Load_Const       8,R1      ; 1 unit
Mult_Mem         a,R1      ; 6 units
Load_Mem         b,R2      ; 3 units
Add_Scaled_Reg   4,R1,R2   ; 4 units
                 Total     = 14 units
```

Figure 4.65 Code resulting from the top-down largest-fit rewrite.

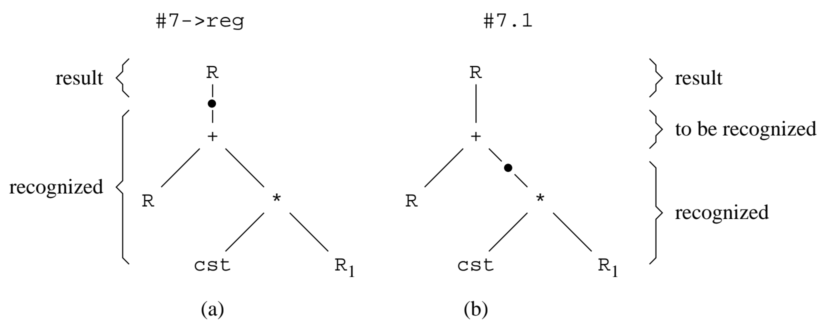


Figure 4.66 The dotted trees corresponding to `#7`→`reg` and to `#7.1`.

```
PROCEDURE Bottom_up pattern matching (Node):
    IF Node is an operation:
        Bottom_up pattern matching (Node .left);
        Bottom_up pattern matching (Node .right);
        SET Node .label set TO Label set for (Node);
    ELSE IF Node is a constant:
        SET Node .label set TO Label set for constant ();
    ELSE Node is a variable:
        SET Node .label set TO Label set for variable ();

FUNCTION Label set for (Node) RETURNING a label set:
    SET Label set TO the Empty set;
    FOR EACH Label IN the Machine label set:
        FOR EACH Left label IN Node .left .label set:
            FOR EACH Right label IN Node .right .label set:
                IF Label .operator = Node .operator
                    AND Label .operand = Left label .result
                    AND Label .second operand = Right label .result:
                    Insert Label into Label set;
    RETURN Label set;

FUNCTION Label set for constant () RETURNING a label set:
    SET Label set TO
        { (No operator, No location, No location, "Constant") };
    FOR EACH Label IN the Machine label set:
        IF Label .operator = "Load" AND Label .operand = "Constant":
            Insert Label into Label set;
    RETURN Label set;

FUNCTION Label set for variable () RETURNING a label set:
    SET Label set TO
        { (No operator, No location, No location, "Memory") };
    FOR EACH Label IN the Machine label set:
        IF Label .operator = "Load" AND Label .operand = "Memory":
            Insert Label into Label set;
    RETURN Label set;
```

Figure 4.67 Outline code for bottom-up pattern matching in trees.

```
TYPE operator: "Load", '+', '*';
TYPE location: "Constant", "Memory", "Register", a label;

TYPE label:                    // a node in a pattern tree
    an operator FIELD operator;
    a location FIELD operand;
    a location FIELD second operand;
    a location FIELD result;
```

Figure 4.68 Types for bottom-up pattern recognition in trees.

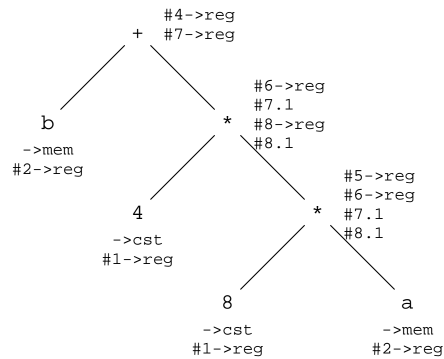


Figure 4.69 Label sets resulting from bottom-up pattern matching.

```
PROCEDURE Bottom_up pattern matching (Node):
    IF Node is an operation:
        Bottom_up pattern matching (Node .left);
        Bottom_up pattern matching (Node .right);
        SET Node .state TO Next state
            [Node .operand, Node .left .state, Node .right .state];
    ELSE IF Node is a constant:
        SET Node .state TO State for constant;
    ELSE Node is a variable:
        SET Node .state TO State for variable;
```

Figure 4.70 Outline code for efficient bottom-up pattern matching in trees.

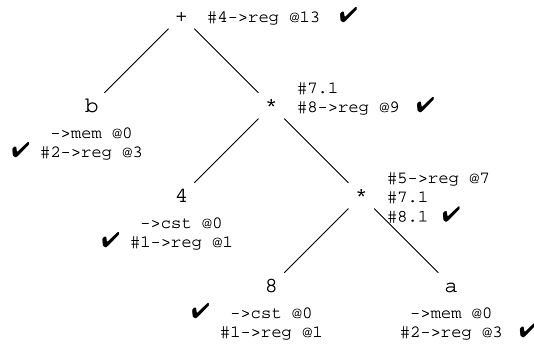


Figure 4.71 Bottom-up pattern matching with costs.

```
Load_Mem          a,R1      ; 3 units
Load_Const        4,R2      ; 1 unit
Mult_Scaled_Reg   8,R1,R2   ; 5 units
Load_Mem          b,R1      ; 3 units
Add_Reg           R2,R1     ; 1 unit
                  Total     = 13 units
```

Figure 4.72 Code generated by bottom-up pattern matching.

```
Load_Mem          a,R1      ; 3 units
Load_Const        4,R2      ; 1 unit
Mult_Scaled_Reg   8,R1,R2   ; 5 units
Add_Mem           b,R1      ; 3 units
                  Total     = 12 units
```

Figure 4.73 Code generated by bottom-up pattern matching, using commutativity.

S_{00} = { }
S_{01} = {→cst@0, #1→reg@1}
S_{02} = {→mem@0, #2→reg@3}
S_{03} = {#4→reg@0}
S_{04} = {#6→reg@5, #7.1@0, #8.1@0}
S_{05} = {#3→reg@0}
S_{06} = {#5→reg@4, #7.1@0, #8.1@0}
S_{07} = {#6→reg@0}
S_{08} = {#5→reg@0}
S_{09} = {#7→reg@0}
S_{10} = {#8→reg@1, #7.1@0, #8.1@0}
S_{11} = {#8→reg@0}
S_{12} = {#8→reg@2, #7.1@0, #8.1@0}
S_{13} = {#8→reg@4, #7.1@0, #8.1@0}

Figure 4.74 States of the BURS automaton for Figure 4.60.

```
cst: S01
mem: S02
```

Figure 4.75 Initial states for the basic operands.

'+'	S_{01}	S_{02}	S_{03}	S_{04}	S_{05}	S_{06}	S_{07}	S_{08}	S_{09}	S_{10}	S_{11}	S_{12}	S_{13}
S_{01} - S_{13}	S_{03}	S_{05}	S_{03}	S_{09}	S_{03}	S_{09}	S_{03}	S_{03}	S_{03}	S_{03}	S_{03}	S_{03}	S_{09}

'*'	S_{01}	S_{02}	S_{03}	S_{04}	S_{05}	S_{06}	S_{07}	S_{08}	S_{09}	S_{10}	S_{11}	S_{12}	S_{13}
S_{01}	S_{04}	S_{06}	S_{04}	S_{10}	S_{04}	S_{12}	S_{04}	S_{04}	S_{04}	S_{04}	S_{04}	S_{13}	S_{12}
S_{02} - S_{13}	S_{07}	S_{08}	S_{07}	S_{11}	S_{07}	S_{11}	S_{07}	S_{07}	S_{07}	S_{07}	S_{07}	S_{11}	S_{11}

Figure 4.76 The transition table `Cost conscious next state[ ]`.

	S_{01}	S_{02}	S_{03}	S_{04}	S_{05}	S_{06}	S_{07}	S_{08}	S_{09}	S_{10}	S_{11}	S_{12}	S_{13}
cst		--	--	--	--	--	--	--	--	--	--	--	--
mem	--		--	--	--	--	--	--	--	--	--	--	--
reg	#1	#2	#4	#6	#3	#5	#6	#5	#7	#8	#8	#8	#8

Figure 4.77 The code generation table.

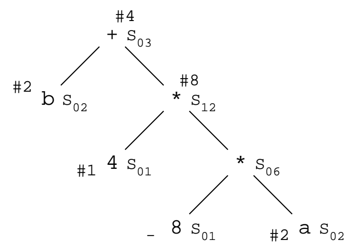


Figure 4.78 States and instructions used in BURS code generation.

```
a := read();
b := read();
c := read();
a := a + b + c;
if (a < 10) {
    d := c + 8;
    print(c);
} else if (a < 20) {
    e := 10;
    d := e + a;
    print(e);
} else {
    f := 12;
    d := f + a;
    print(f);
}
print(d);
```

Figure 4.79 A program segment for live analysis.

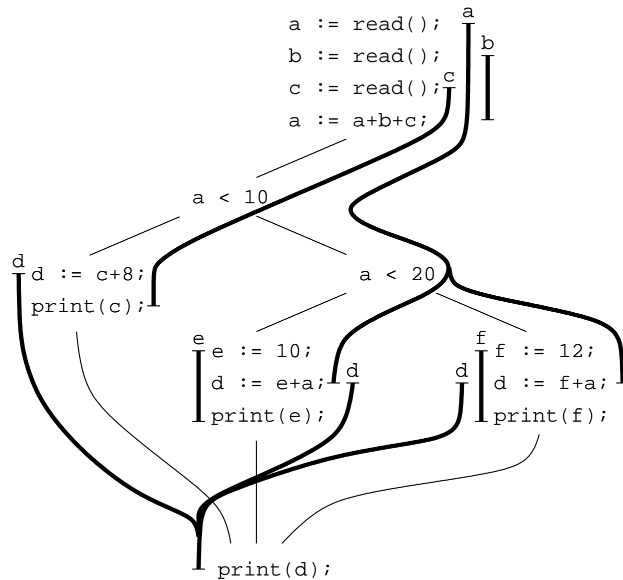


Figure 4.80 Live ranges of the variables from Figure 4.79.

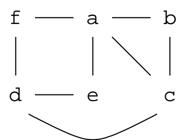


Figure 4.81 Register interference graph for the variables of Figure 4.79.

```
FUNCTION Color graph (Graph) RETURNING the Colors used set:
    IF Graph = the Empty graph: RETURN the Empty set;

    // Find the Least connected node:
    SET Least connected node TO No node;
    FOR EACH Node IN Graph .nodes:
        SET Degree TO 0;
        FOR EACH Arc IN Graph .arcs:
            IF Arc contains Node:
                Increment Degree;
        IF Least connected node = No node OR
            Degree < Minimum degree:
            SET Least connected node TO Node;
            SET Minimum degree TO Degree;

    // Remove Least connected node from Graph:
    SET Least connected node arc set TO the Empty set;
    FOR EACH Arc IN Graph .arcs:
        IF Arc contains Least connected node:
            Remove Arc from Graph .arcs;
            Insert Arc in Least connected node arc set;
    Remove Least connected node from Graph .nodes;

    // Color the reduced Graph recursively:
    SET Colors used set TO Color graph (Graph);

    // Color the Least connected node:
    SET Left over colors set TO Colors used set;
    FOR EACH Arc IN Least connected node arc set:
        FOR EACH End point IN Arc:
            IF End point /= Least connected node:
                Remove End point .color from Left over colors set;
    IF Left over colors set = Empty:
            SET Color TO New color;
            Insert Color in Colors used set;
            Insert Color in Left over colors set;
    SET Least connected node .color TO
        Arbitrary choice from Left over colors set;

    // Reattach the Least connected node:
    Insert Least connected node in Graph .nodes;
    FOR EACH Arc IN Least connected node arc set:
        Insert Arc in Graph .arcs;
    RETURN Colors used set;
```

Figure 4.82 Outline of a graph coloring algorithm.

```
Graph:
 f —— a —— b      f —— a            f —— a     f           f
 |    | \  |  ->  |    | \      ->  |    |  -> |       ->  |        ->
 d —— e    c      d —— e    c       d —— e     d —— e      d
   \_______/        \_______/
Stack:            b                 c b        a c b       e a c b

 f                         f1
          ->               ->            ->
 d e a c b     f d e a c b     d e a c b

 f1          f1          f1 —— a2    f1 —— a2           f1 —— a2 —— b3
 |      ->   |      ->   |     |  -> |     | \      ->  |     | \   |
 d2          d2 —— e1    d2 —— e1    d2 —— e1   c1      d2 —— e1    c1
                                       \________/         \_________/
 e a c b     a c b       c b         b
```

Figure 4.83 Coloring the interference graph for the variables of Figure 4.79.

```
; n in register %ax
    cwd             ; convert to double word:
                    ;     (%dx,%ax) = (extend_sign(%ax), %ax)
    negw %ax        ; negate: (%ax,cf) := (-%ax, %ax /= 0)
    adcw %dx,%dx    ; add with carry: %dx := %dx + %dx + cf
; sign(n) in %dx
```

Figure 4.84 Optimal code for the function `sign(n)`.

	Case n > 0			Case n = 0			Case n < 0		
	%dx	%ax	cf	%dx	%ax	cf	%dx	%ax	cf
initially:	–	n	–	–	0	–	–	n	–
cwd	0	n	–	0	0	–	-1	n	–
negw %ax	0	-n	1	0	0	0	-1	-n	1
adcw %dx,%dx	1	-n	1	0	0	0	-1	-n	1

Figure 4.85 Actions of the 80x86 code from Figure 4.84.

Problem	Technique	Quality
Expression trees, using register-register or memory-register instructions	Weighted trees; Figure 4.30	
with sufficient registers:		Optimal
with insufficient registers:		Optimal
Dependency graphs, using register-register or memory-register instructions	Ladder sequences; Section 4.2.5.2	Heuristic
Expression trees, using any instructions with cost function	Bottom-up tree rewriting; Section 4.2.6	
with sufficient registers:		Optimal
with insufficient registers:		Heuristic
Register allocation when all interferences are known	Graph coloring; Section 4.2.7	Heuristic

Figure 4.86 Comparison of some code generation techniques.

```
int i, A[10];

for (i = 0; i < 20; i++) {
    A[i] = 2*i;
}
```

Figure 4.87 Incorrect C program with compilation-dependent effect.

operation	$\Rightarrow$	replacement
`E * 2 ** n`	$\Rightarrow$	`E << n`
`2 * V`	$\Rightarrow$	`V + V`
`3 * V`	$\Rightarrow$	`(V << 1) + V`
`V ** 2`	$\Rightarrow$	`V * V`
`E + 0`	$\Rightarrow$	`E`
`E * 1`	$\Rightarrow$	`E`
`E ** 1`	$\Rightarrow$	`E`
`1 ** E`	$\Rightarrow$	`1`

Figure 4.88 Some transformations for arithmetic simplification.

```
void S {
    ...
    print_square(i++);
    ...
}

void print_square(int n) {
    printf("square = %d\n", n*n);
}
```

Figure 4.89 C code with a routine to be in-lined.

```
void S {
    ...
    {int n = i++; printf("square = %d\n", n*n);}
    ...
}
void print_square(int n) {
    printf("square = %d\n", n*n);
}
```

Figure 4.90 C code with the routine in-lined.

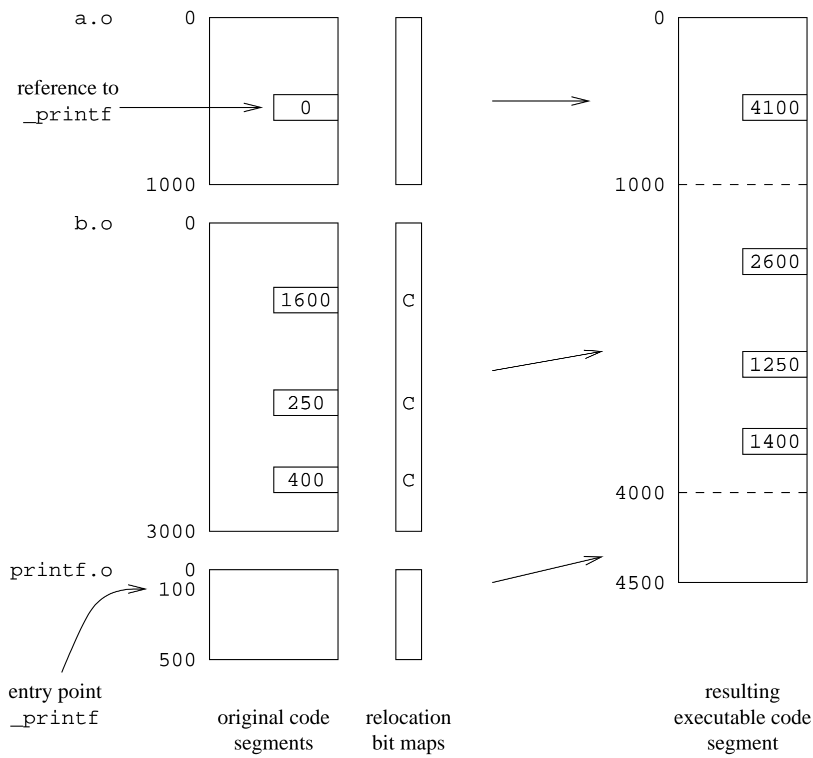


Figure 4.91 Linking three code segments.

```
.data
        ...
        .align 8
var1:
        .long 666
        ...
.code
        ...
        addl var1,%eax
        ...
        jmp label1
        ...
label1:
        ...
        ...
```

Figure 4.92 Assembly code fragment with internal symbols.

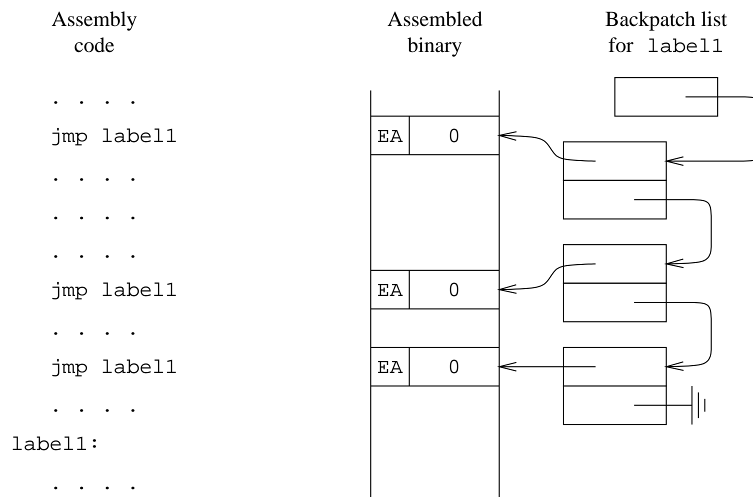


Figure 4.93 A backpatch list for labels.

External symbol	Type	Address
_options	entry point	50 data
__main	entry point	100 code
_printf	reference	500 code
_atoi	reference	600 code
_printf	reference	650 code
_exit	reference	700 code
_msg_list	entry point	300 data
_Out_Of_Memory	entry point	800 code
_fprintf	reference	900 code
_exit	reference	950 code
_file_list	reference	4 data

Figure 4.94 Example of an external symbol table.

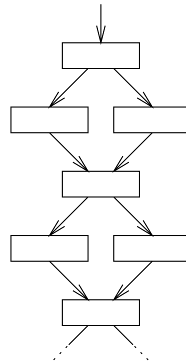


Figure 4.95 Test graph for recursive descent marking.

```
void Routine(void) {
    if (...) {
        while (...) {
            A;
        }
    }
    else {
        switch (...) {
        case: ...: B; break;
        case: ...: C; break;
        }
    }
}
```

Figure 4.96 Routine code for static profiling.

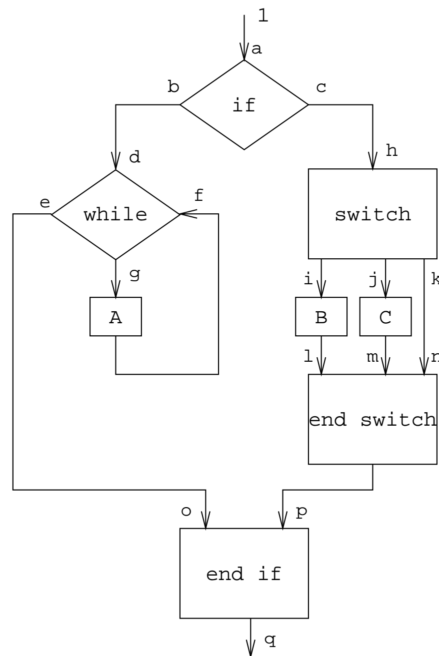


Figure 4.97 Flow graph for static profiling of Figure 4.96.

5

Memory management

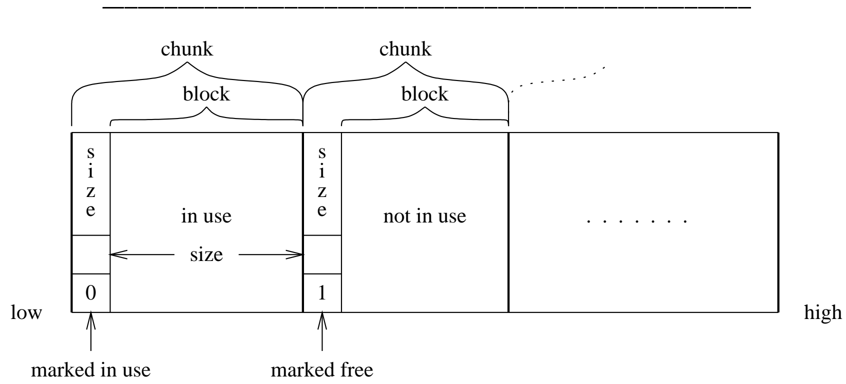


Figure 5.1 Memory structure used by the malloc/free mechanism.

```
SET the polymorphic chunk pointer First_chunk pointer TO
    Beginning of available memory;
SET the polymorphic chunk pointer One past available memory TO
    Beginning of available memory + Size of available memory;

SET First_chunk pointer .size TO Size of available memory;
SET First_chunk pointer .free TO True;

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;

    Coalesce free chunks;
    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);

PROCEDURE Free (Block pointer):
    SET Chunk pointer TO Block pointer - Administration size;
    SET Chunk pointer .free TO True;
```

Figure 5.2 A basic implementation of `Malloc(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 .free:
            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;

PROCEDURE Split chunk (Chunk pointer, Requested chunk size):
    SET Left_over size TO Chunk pointer .size - Requested chunk size;
    IF Left_over size > Administration size:
        // there is a non-empty left-over chunk
        SET Chunk pointer .size TO Requested chunk size;
        SET Left_over chunk pointer TO
            Chunk pointer + Requested chunk size;
        SET Left_over chunk pointer .size TO Left_over size;
        SET Left_over chunk pointer .free TO True;

PROCEDURE Coalesce free chunks:
    SET Chunk pointer TO First_chunk pointer;

    WHILE Chunk pointer /= One past available memory:
        IF Chunk pointer .free:
            Coalesce with all following free chunks (Chunk pointer);
        SET Chunk 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;
    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;
```

Figure 5.3 Auxiliary routines for the basic `Malloc(Block size)`.

```
SET Free list for T TO No T;

FUNCTION New T () RETURNING a pointer to an T:
    IF Free list for T = No T:
        // Acquire a new block of records:
        SET New block [1 .. Block factor for T] TO
            Malloc (Size of T * Block factor for T);
        // Construct a free list in New block:
        SET Free list for T TO address of New block [1];
        FOR Record count IN [1 .. Block factor for T - 1]:
            SET New block [Record count] .link TO
                address of New block [Record count + 1];
        SET New block [Block factor for T] .link TO No T;

    // Extract a new record from the free list:
    SET New record TO Free list for T;
    SET Free list for T TO New record .link;

    // Zero the New record here, if required;
    RETURN New record;

PROCEDURE Free T (Old record):
    // Prepend Old record to free list:
    SET Old record .link TO Free list for T;
    SET Free list for T TO address of Old record;
```

Figure 5.4 Outline code for blockwise allocation of records of type *T*.

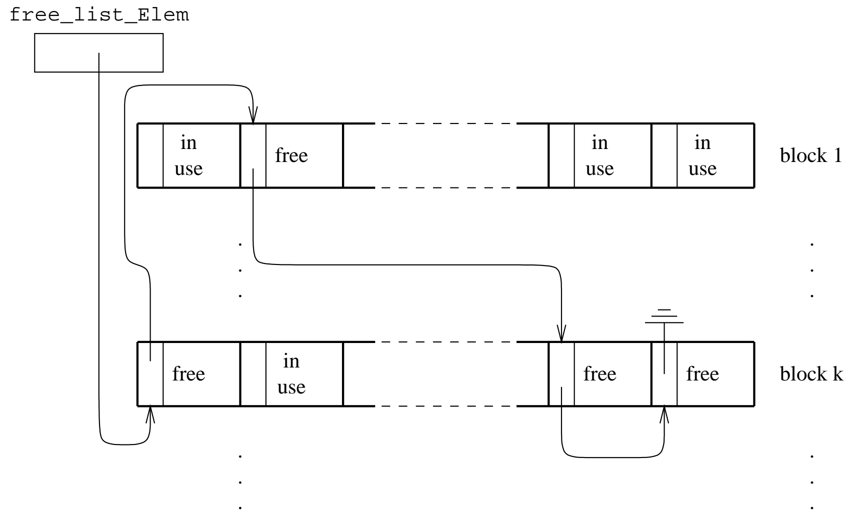


Figure 5.5 List of free records in allocated blocks.

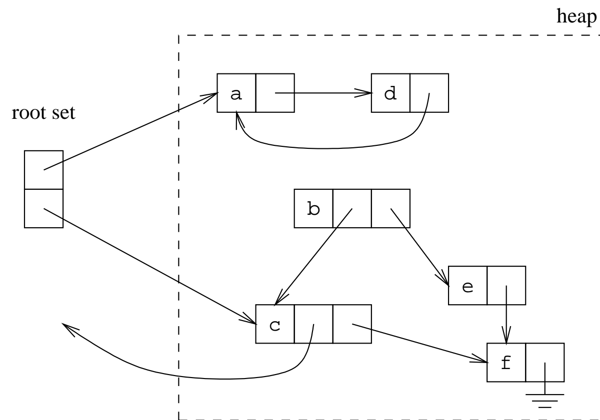


Figure 5.6 A root set and a heap with reachable and unreachable chunks.

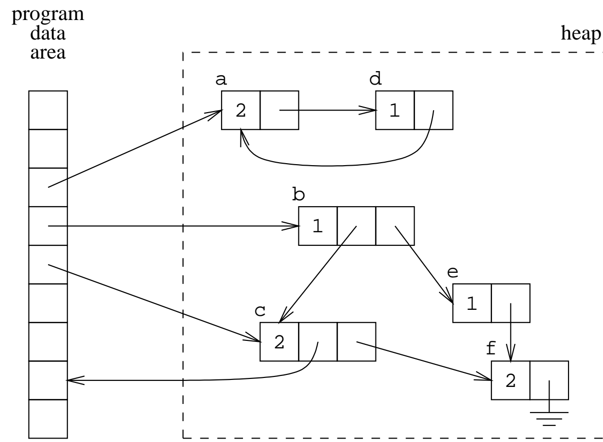


Figure 5.7 Chunks with reference count in a heap.

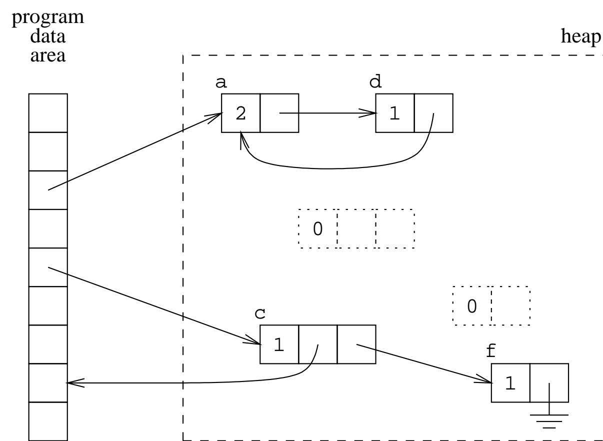


Figure 5.8 Result of removing the reference to chunk `b` in Figure 5.7.

```
IF Points into the heap (q):
    Increment q .reference count;
IF Points into the heap (p):
    Decrement p .reference count;
    IF p .reference count = 0:
        Free recursively depending on reference counts (p);
SET p TO q;
```

Figure 5.9 Code to be generated for the pointer assignment `p:=q`.

```
PROCEDURE Free recursively depending on reference counts(Pointer);
    WHILE Pointer /= No chunk:
        IF NOT Points into the heap (Pointer): RETURN;
        IF NOT Pointer .reference count = 0: RETURN;

        FOR EACH Index IN 1 .. Pointer .number of pointers - 1:
            Free recursively depending on reference counts
                (Pointer .pointer [Index]);

        SET Aux pointer TO Pointer;
        IF Pointer .number of pointers = 0:
            SET Pointer TO No chunk;
        ELSE Pointer .number of pointers > 0:
            SET Pointer TO
                Pointer .pointer [Pointer .number of pointers];
        Free chunk(Aux pointer);    // the actual freeing operation
```

Figure 5.10 Recursively freeing chunks while avoiding tail recursion.

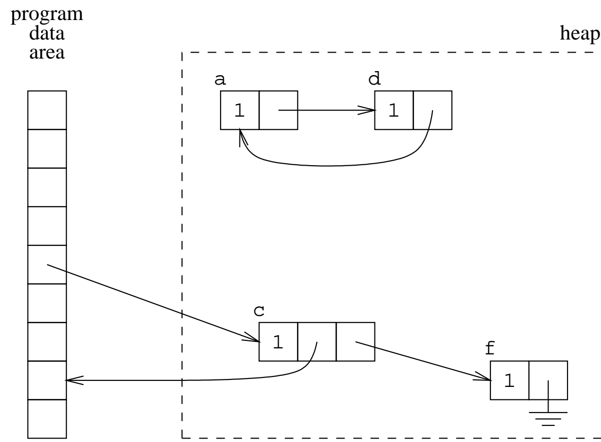


Figure 5.11 Reference counting fails to identify circular garbage.

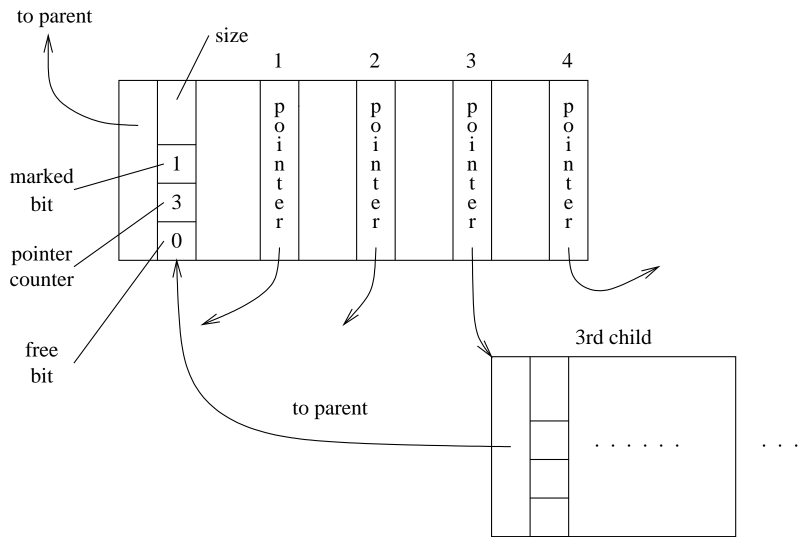


Figure 5.12 Marking the third child of a chunk.

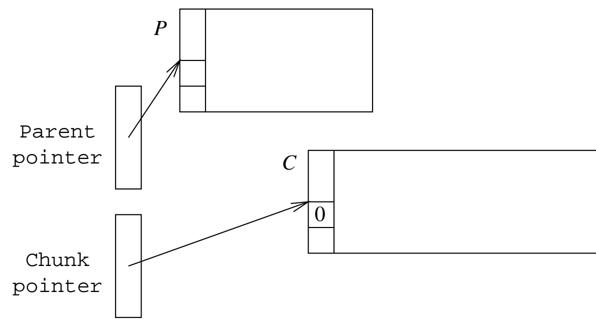


Figure 5.13 The Schorr and Waite algorithm, arriving at C.

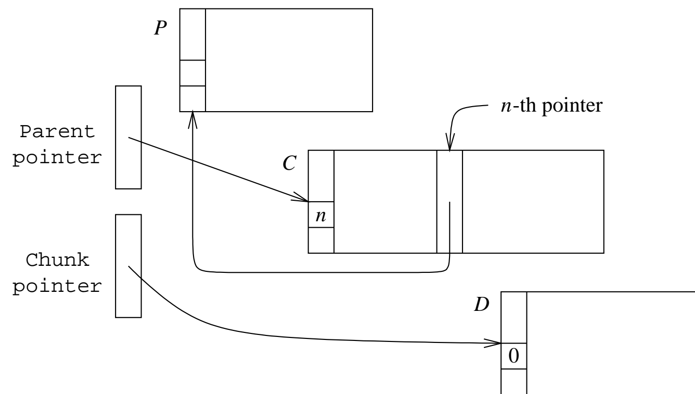


Figure 5.14 Moving to D.

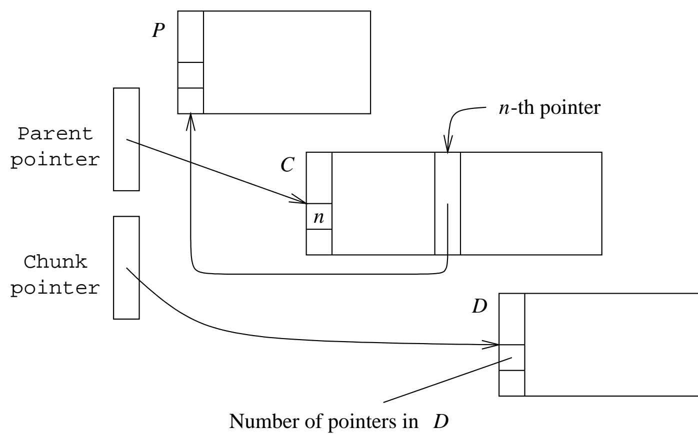


Figure 5.15 About to return from D.

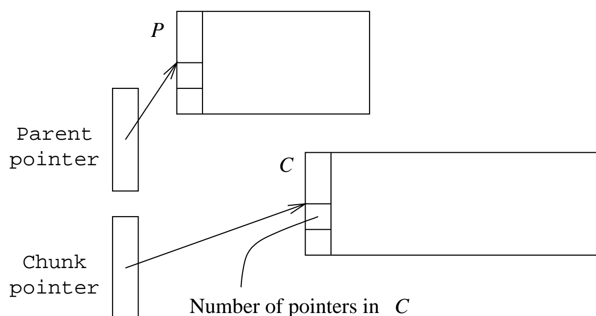


Figure 5.16 About to return from C.

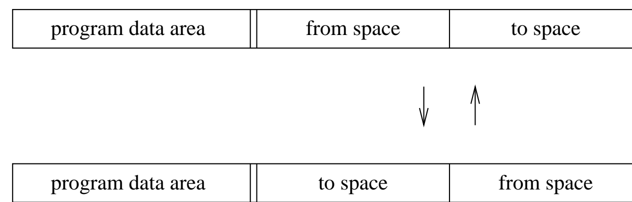


Figure 5.17 Memory layout for two-space copying.

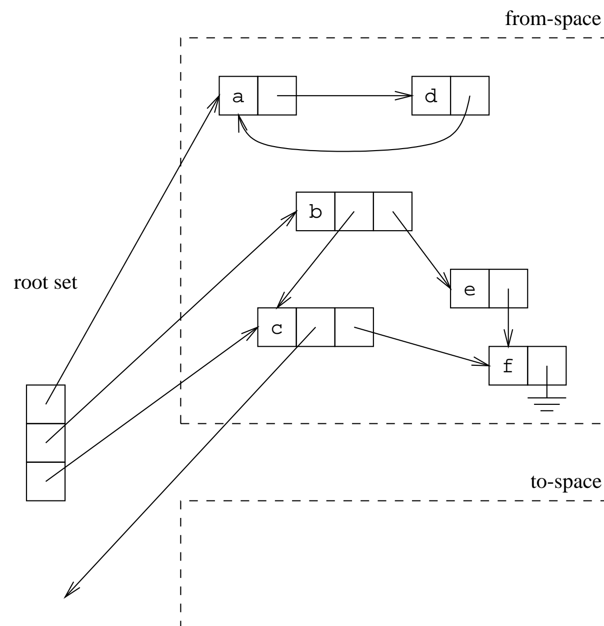


Figure 5.18 Initial situation in two-space copying.

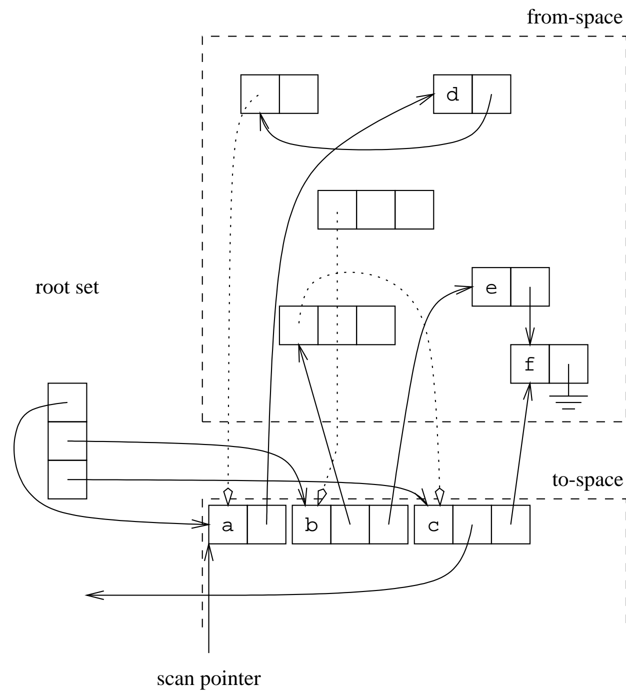


Figure 5.19 Snapshot after copying the first level.

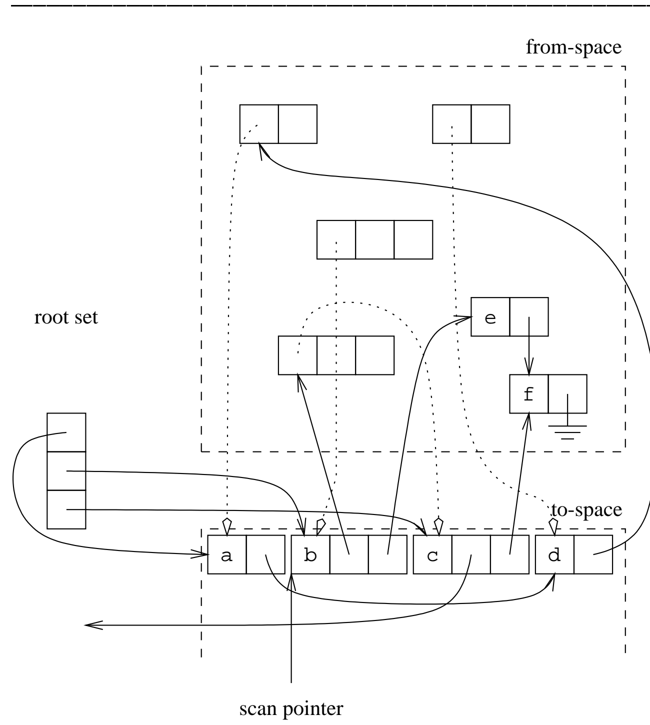


Figure 5.20 Snapshot after having scanned one chunk.

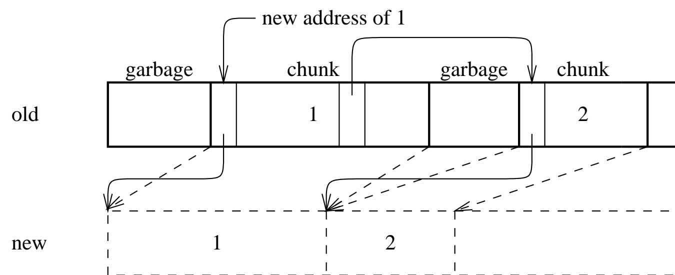


Figure 5.21 Address calculation during compaction.

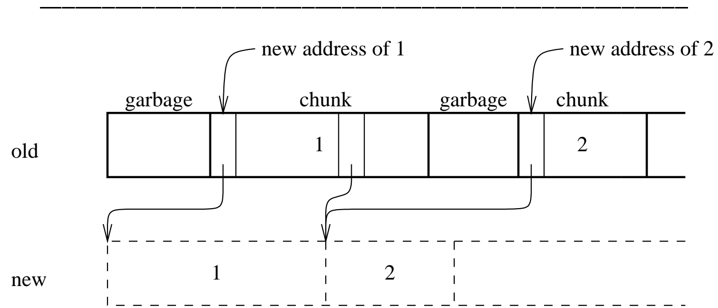


Figure 5.22 Pointer update during compaction.

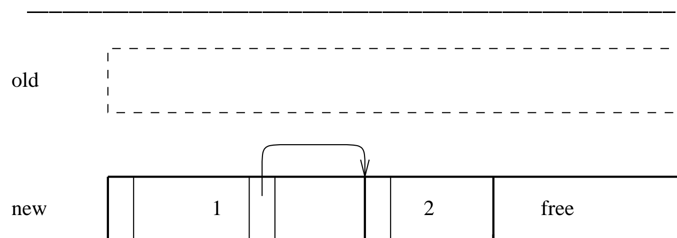


Figure 5.23 Moving the chunks during compaction.

6

Imperative and object-oriented programs

		Imperative and object-oriented	Functional	Logic	Parallel and distributed
Lexical and syntactic analysis,	general:	2			
	specific:	–			
Context handling,	general:	3			
	identification & type checking:	6.1			
	specific:	6	7	8	9
Code generation,	general:	4			
	specific:	6	7	8	9

Figure 6.1 Roadmap to paradigm-specific issues.

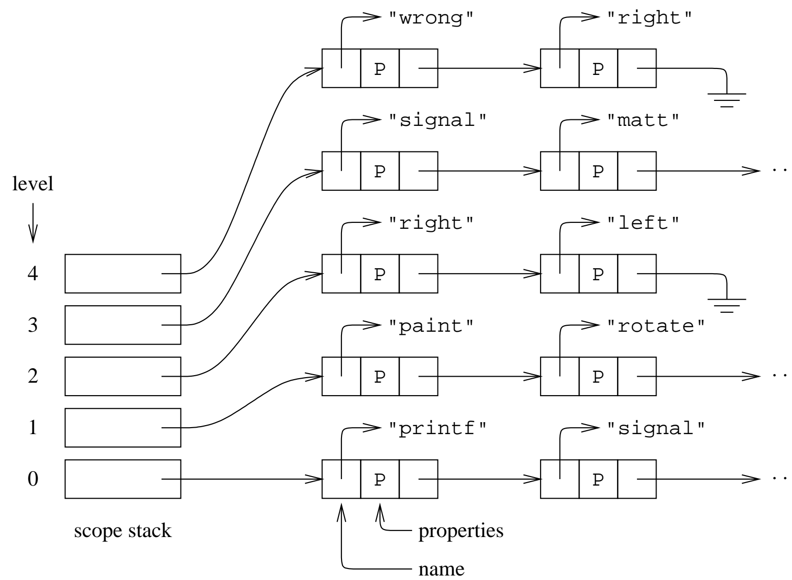


Figure 6.2 A naive scope-structured symbol table.

```
void rotate(double angle) {
    ...
}
void paint(int left, int right) {
    Shade matt, signal;

    ...
    {   Counter right, wrong;
        ...
    }
}
```

Figure 6.3 C code leading to the symbol table in Figure 6.2.

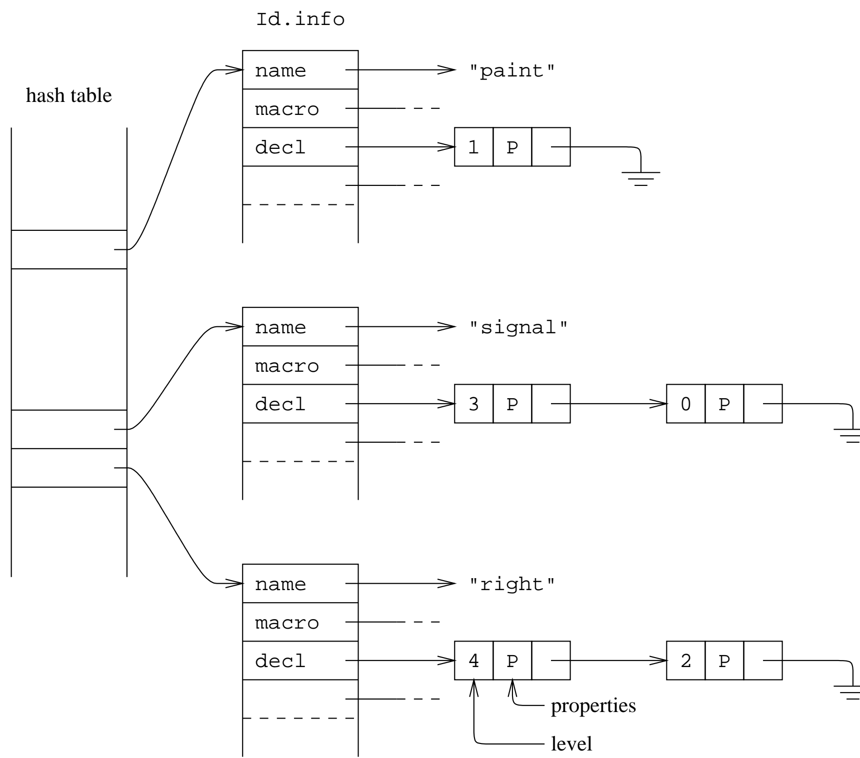


Figure 6.4 A hash-table based symbol table.

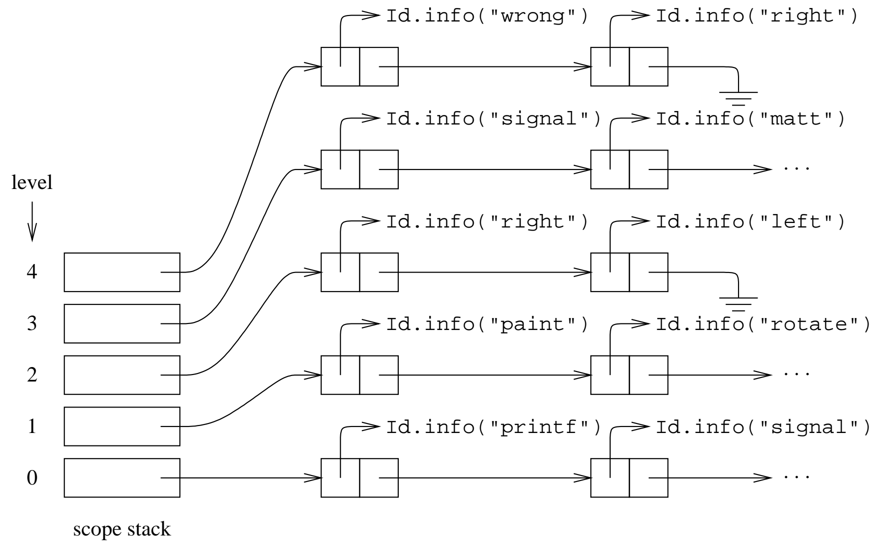


Figure 6.5 Scope table for the hash-table based symbol table.

```
PROCEDURE Remove topmost scope ():
    SET Link pointer TO Scope stack [Top level];
    WHILE Link pointer /= No link:
        // Get the next Identification info record
        SET Idf pointer TO Link pointer .idf_info;
        SET Link pointer TO Link pointer .next;
        // Get its first Declaration info record
        SET Declaration pointer TO Idf pointer .decl;
        // Now Declaration pointer .level = Top level
        // Detach the first Declaration info record
        SET Idf pointer .decl TO Declaration pointer .next;
        Free the record pointed at by Declaration pointer;
    Free Scope stack [Top level];
    SET Top level TO Top level - 1;
```

Figure 6.6 Outline code for removing declarations at scope exit.

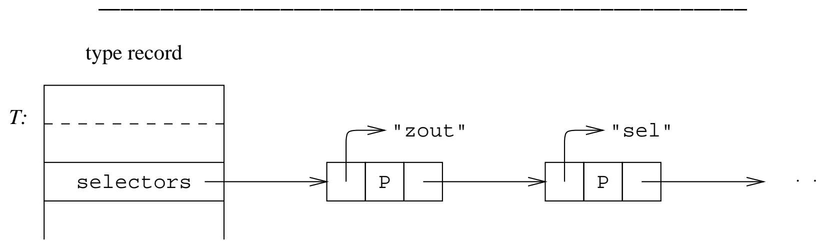


Figure 6.7 Finding the type of a selector.

Data definitions:

1. Let T be a type table that has entries containing either a type description or a reference (TYPE) to a type table entry.
2. Let *Cyclic* be a set of type table entries.

Initializations:

Initialize the *Cyclic* set to empty.

Inference rules:

If there exists a TYPE type table entry t_1 in T and t_1 is not a member of *Cyclic*, let t_2 be the type table entry referred to by t_1.

1. If t_2 is t_1 then add t_1 to *Cyclic*.
2. If t_2 is a member of *Cyclic* then add t_1 to *Cyclic*.
3. If t_2 is again a TYPE type table entry, replace, in t_1, the reference to t_2 with the reference referred to by t_2.

Figure 6.8 Closure algorithm for detecting cycles in type definitions.

Data definitions:

Let each node in the expression have a variable type set S attached to it. Also, each node is associated with a non-variable context C.

Initializations:

The type set S of each operator node contains the result types of all identifications of the operator in it; the type set S of each leaf node contains the types of all identifications of the node. The context C of a node derives from the language manual.

Inference rules:

For each node N with context C, if its type set S contains a type T_1 which the context C allows to be coerced to a type T_2, T_2 must also be present in S.

Figure 6.9 The closure algorithm for identification in the presence of overloading and coercions, phase 1.

Data definitions:

Let each node in the expression have a type set S attached to it.

Initializations:

Let the type set S of each node be filled by the algorithm of Figure 6.9.

Inference rules:

1. For each operator node N with type set S, if S contains a type T such that there is no operator identified in N that results in T and has operands T_1 and T_2 such that T_1 is in the type set of the left operand of N and T_2 is in the type set of the right operand of N, T is removed from S.
2. For each operand node N with type set S, if S contains a type T that is not compatible with at least one type of the operator that works on N, T is removed from S.

Figure 6.10 The closure algorithm for identification in the presence of overloading and coercions, phase 2.

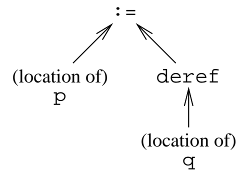


Figure 6.11 AST for `p := q` with explicit deref.

		expected	
		lvalue	rvalue
found	lvalue	–	deref
	rvalue	error	–

Figure 6.12 Basic checking rules for lvalues and rvalues.

expression construct	*resulting kind*
constant	rvalue
identifier (variable)	lvalue
identifier (otherwise)	rvalue
&lvalue	rvalue
*rvalue	lvalue
V[rvalue]	*V*
V.selector	*V*
rvalue+rvalue	rvalue
lvalue:=rvalue	rvalue

Figure 6.13 lvalue/rvalue requirements and results of some expression constructs.

```
void do_buffer(void) {
    char *buffer;

    obtain_buffer(&buffer);
    use_buffer(buffer);
}

void obtain_buffer(char **buffer_pointer) {
    char actual_buffer[256];

    *buffer_pointer = &actual_buffer;
        /* this is a scope-violating assignment: */
        /* scope of *buffer_pointer > scope of &actual_buffer */
}
```

Figure 6.14 Example of a scope violation in C.

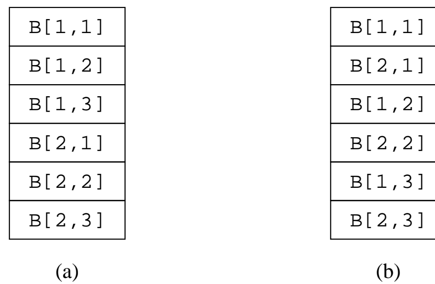


Figure 6.15 Array `B` in row-major (a) and column-major (b) order.

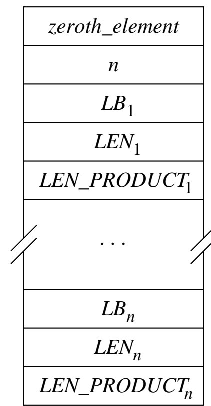


Figure 6.16 An array descriptor.

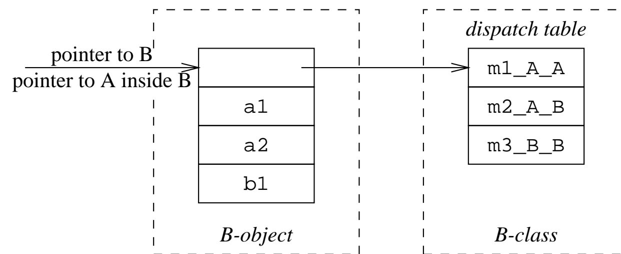


Figure 6.17 The representation of an object of class `B`.

```
class C {
    field c1;
    field c2;
    method m1();
    method m2();
};

class D {
    field d1;
    method m3();
    method m4();
};

class E extends C, D {
    field e1;
    method m2();
    method m4();
    method m5();
};
```

Figure 6.18 An example of multiple inheritance.

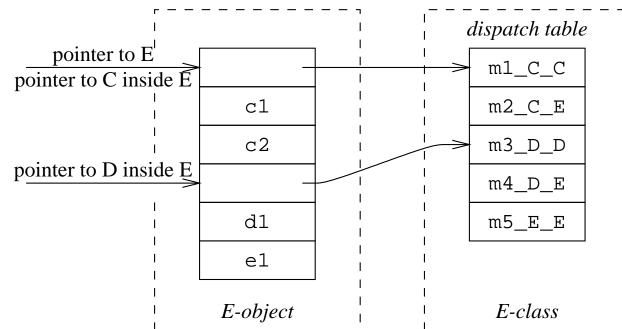


Figure 6.19 A representation of an object of class E.

```
class A {
    field a1;
    field a2;
    method m1();
    method m3();
};

class C extends A {
    field c1;
    field c2;
    method m1();
    method m2();
};

class D extends A {
    field d1;
    method m3();
    method m4();
};

class E extends C, D {
    field e1;
    method m2();
    method m4();
    method m5();
};
```

Figure 6.20 An example of dependent multiple inheritance.

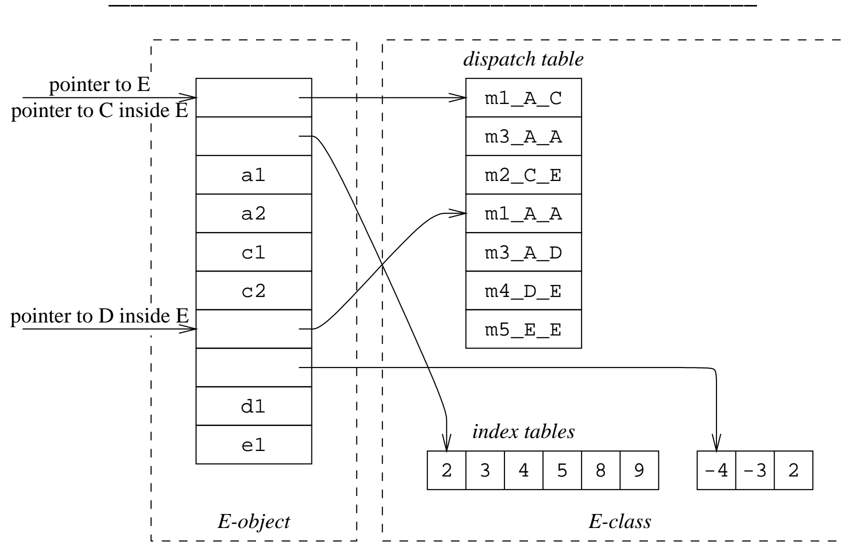


Figure 6.21 An object of class E, with dependent inheritance.

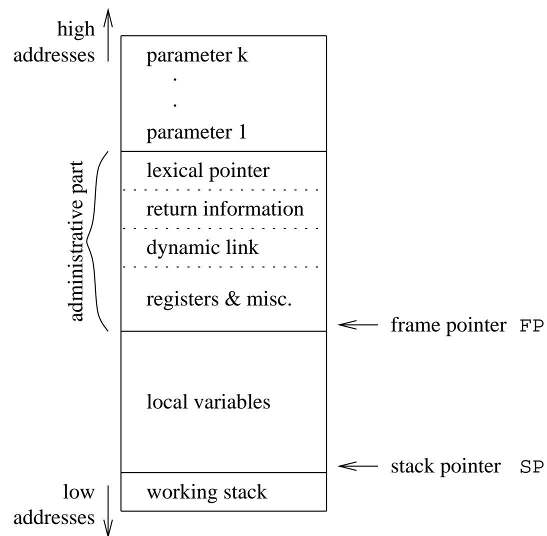


Figure 6.22 Possible structure of an activation record.

```
void use_iterator(void) {
    int i;

    while ((i = get_next_int()) > 0) {
        printf("%d\n", i);
    }
}

int get_next_int(void) {
    yield 8;
    yield 3;
    yield 0;
}
```

Figure 6.23 An example application of an iterator in C notation.

```
char buffer[100];

void Producer(void) {
    while (produce_buffer()) {
        resume(Consumer);
    }
    empty_buffer();      /* signal end of stream */
    resume(Consumer);
}

void Consumer(void) {
    resume(Producer);
    while (!empty_buffer_received()) {
        consume_buffer();
        resume(Producer);
    }
}
```

Figure 6.24 Simplified producer/consumer communication using coroutines.

```
int i;

void level_0(void) {
    int j;

    void level_1(void) {
        int k;

        void level_2(void) {
            int l;

            ...         /* code has access to i, j, k, l */
            k = l;
            j = l;
        }

        ...             /* code has access to i, j, k */
        j = k;
    }

    ...                 /* code has access to i, j */
}
```

Figure 6.25 Nested routines in C notation.

```
void level_0(void) {
    void level_1(void) {
        void level_2(void) {
            ...
            goto L_1;
            ...
        }
        ...
    L_1:...
        ...
    }
    ...
}
```

Figure 6.26 Example of a non-local goto.

```
#include <setjmp.h>

void find_div_7(int n, jmp_buf *jmpbuf_ptr) {
    if (n % 7 == 0) longjmp(*jmpbuf_ptr, n);
    find_div_7(n + 1, jmpbuf_ptr);
}

int main(void) {
    jmp_buf jmpbuf;          /* type defined in setjmp.h */
    int return_value;

    if ((return_value = setjmp(jmpbuf)) == 0) {
        /* setting up the label for longjmp() lands here */
        find_div_7(1, &jmpbuf);
    }
    else {
        /* returning from a call of longjmp() lands here */
        printf("Answer = %d\n", return_value);
    }
    return 0;
}
```

Figure 6.27 Demonstration of the setjmp/longjmp mechanism.

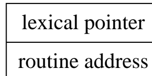


Figure 6.28 A routine descriptor for a language that requires lexical pointers.

```
void level_1(void) {
    int k;

    void level_2(void) {
        int l;

        ...
    }
    routine_descriptor level_2_as_a_value = {
        FP_of_this_activation_record(),  /* FP of level_1() */
        level_2                  /* code address of level_2()*/
    };

    A(level_2_as_a_value);      /* level_2() as a parameter */
}
```

Figure 6.29 Possible code for the construction of a routine descriptor.

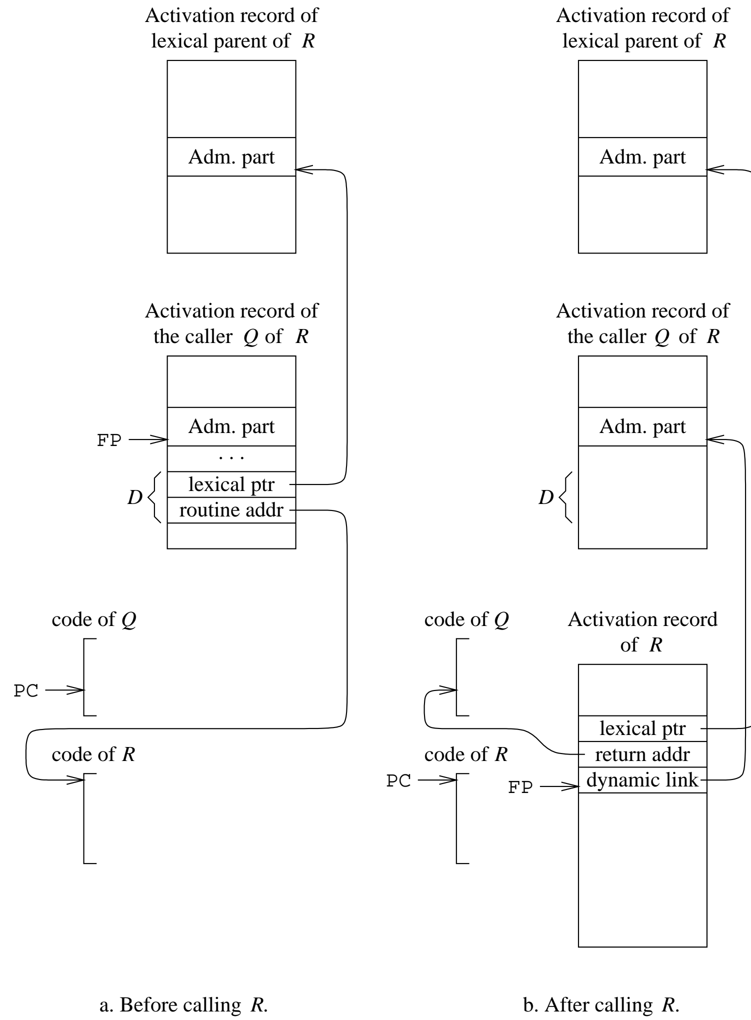


Figure 6.30 Calling a routine defined by the two-pointer routine descriptor *D*.

```
*(
    *(
        FP
        +
        offset(lexical_pointer)
    )
    +
    offset(k)
) =
*(FP + offset(l))
```

Figure 6.31 Intermediate code for the non-local assignment `k = l`.

```
*(
    *(
        *(  FP
            +
            offset(lexical_pointer)
        )
        +
        offset(lexical_pointer)
    )
    +
    offset(j)
) =
*(FP + offset(l))
```

Figure 6.32 Intermediate code for the non-local assignment `j = l`.

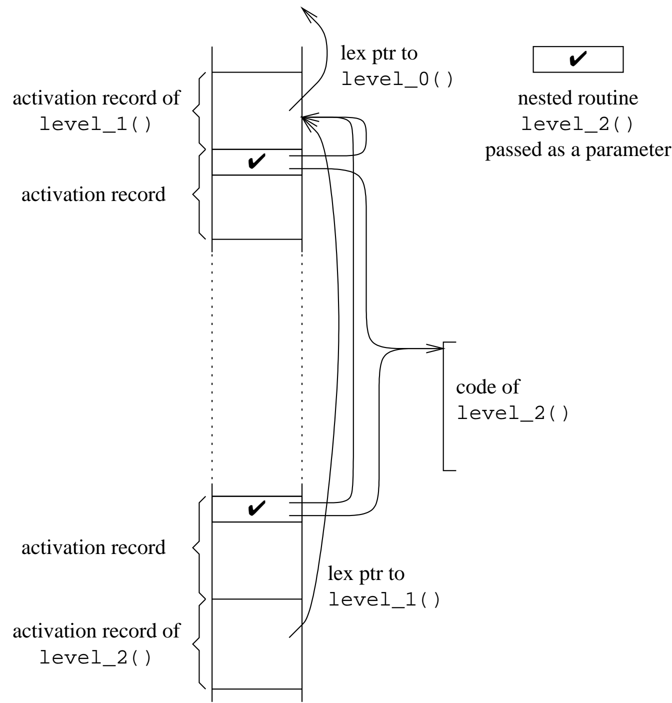


Figure 6.33 Passing a nested routine as a parameter and calling it.

```
void level_1(void) {
    non_local_label_descriptor L_1_as_a_value = {
        FP_of_this_activation_record(),  /* FP of level_1() */
        L_1                              /* code address of L_1 */
    };

    void level_2(void) {

        ...
        non_local_goto(L_1_as_a_value);     /* goto L_1; */
        ...
    }

    ...
L_1:...
    ...
}
```

Figure 6.34 Possible code for the construction of a label descriptor.

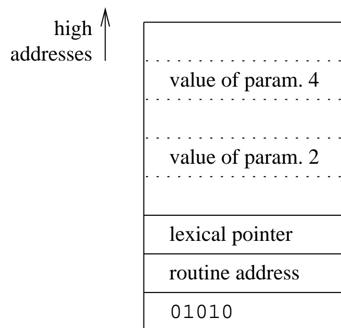


Figure 6.35 A closure for a partially parameterized routine.

```
int i;

void level_2(int *j, int *k) {
    int l;

    ...                         /* code has access to i, *j, *k, l */
    *k = l;
    *j = l;
}

void level_1(int *j) {
    int k;

    ...                         /* code has access to i, *j, k */
    level_2(j, &k);             /* was: level_2(); */
    A(level_2);                 /* level_2() as a parameter: */
                                /* this is a problem */
}

void level_0(void) {
    int j;

    ...                         /* code has access to i, j */
    level_1(&j);                /* was: level_1(); */
}
```

Figure 6.36 Lambda-lifted routines in C notation.

```
int i;

void level_2(int *j, int *k) {
    int l;

    ...                            /* code has access to i, *j, *k, l */
    *k = l;
    *j = l;
}

void level_1(int *j) {
    int *k = (int *)malloc(sizeof (int));

    ...                            /* code has access to i, *j, *k */
    level_2(j, k);                 /* was: level_2(); */
    A(closure(level_2, j, k));     /* was: A(level_2); */
}

void level_0(void) {
    int *j = (int *)malloc(sizeof (int));

    ...                            /* code has access to i, *j */
    level_1(j);                    /* was: level_1(); */
}
```

Figure 6.37 Lambda-lifted routines with additional heap allocation in C notation.

```
PROCEDURE Generate code for Boolean control expression (
    Node, True label, False label
):
    SELECT Node .type:
        CASE Comparison type:                     // <, >, ==, etc. in C
            Generate code for comparison expression (Node .expr);
            // The comparison result is now in the condition register
            IF True label /= No label:
                Emit ("IF condition register THEN GOTO" True label);
                IF False label /= No label:
                    Emit ("GOTO" False label);
            ELSE True label = No label:
                IF False label /= No label:
                    Emit ("IF NOT condition register THEN GOTO"
                        False label);
        CASE Lazy and type:                       // the && in C
            Generate code for Boolean control expression
                (Node .left, No label, False label);
            Generate code for Boolean control expression
                (Node .right, True label, False label);
        CASE ...
        CASE Negation type:                       // the ! in C
            Generate code for Boolean control expression
                (Node, False label, True label);
```

Figure 6.38 Code generation for Boolean expressions.

```
    tmp_case_value := case expression;
    IF tmp_case_value = I_1 THEN GOTO label_1;
    ...
    IF tmp_case_value = I_n THEN GOTO label_n;
    GOTO label_else;     // or insert the code at label_else
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 6.39 A simple translation scheme for case statements.

```
    i := lower bound;
    tmp_ub := upper bound;
    tmp_step_size := step_size;
    IF tmp_step_size = 0 THEN
        ... probably illegal; cause run-time error ...
    IF tmp_step_size < 0 THEN GOTO neg_step;
    IF i > tmp_ub THEN GOTO end_label;
    // the next statement uses tmp_ub - i
    //     to evaluate tmp_loop_count to its correct, unsigned value
    tmp_loop_count := (tmp_ub - i) DIV tmp_step_size + 1;
    GOTO loop_label;
neg_step:
    IF i < tmp_ub THEN GOTO end_label;
    // the next statement uses i - tmp_ub
    //     to evaluate tmp_loop_count to its correct, unsigned value
    tmp_loop_count := (i - tmp_ub) DIV (-tmp_step_size) + 1;
loop_label:
    code for statement sequence
    tmp_loop_count := tmp_loop_count - 1;
    IF tmp_loop_count = 0 THEN GOTO end_label;
    i := i + tmp_step_size;
    GOTO loop_label;
end_label:
```

Figure 6.40 Code generation for a general for-statement.

```
for (expr1; expr2; expr3) {
    body;
}
```

Figure 6.41 A for-loop in C.

```
expr1;
while (expr2) {
    body;
    expr3;
}
```

Figure 6.42 A while loop that is almost equivalent to the for-loop of Figure 6.41.

```
FOR i := 1 TO n-3 STEP 4 DO
    // The first loop takes care of the indices 1 .. (n div 4) * 4
    sum := sum + a[i];
    sum := sum + a[i+1];
    sum := sum + a[i+2];
    sum := sum + a[i+3];
END FOR;

FOR i := (n div 4) * 4 + 1 TO n DO
    // This loop takes care of the remaining indices
    sum := sum + a[i];
END FOR;
```

Figure 6.43 Two for-loops resulting from unrolling a for-loop.

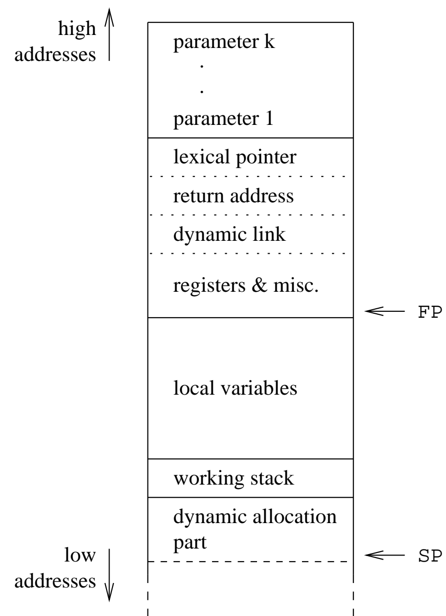


Figure 6.44 An activation record with a dynamic allocation part.

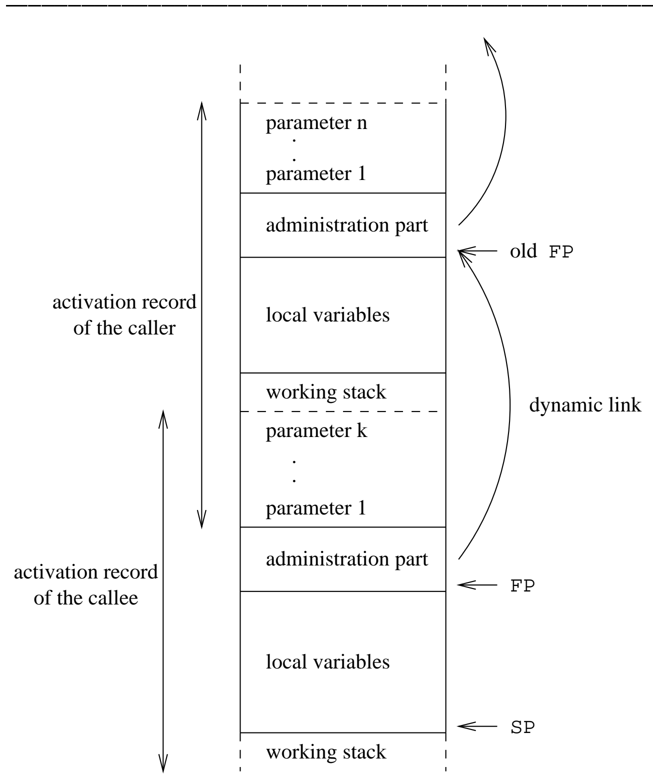


Figure 6.45 Two activation records on a stack.

```
#include <setjmp.h>

jmp_buf jmpbuf;          /* type defined in setjmp.h */

void handler(int signo) {
    printf("ERROR, signo = %d\n", signo);
    longjmp(jmpbuf, 1);
}

int main(void) {
    signal(6, handler);          /* install the handler ... */
    signal(12, handler);         /* ... for some signals */
    if (setjmp(jmpbuf) == 0) {
        /* setting up the label for longjmp() lands here */
        /* normal code: */
        ...
    } else {
        /* returning from a call of longjmp() lands here */
        /* exception code: */
        ...
    }
}
```

Figure 6.46 An example program using `setjmp`/`longjmp` in a signal handler.

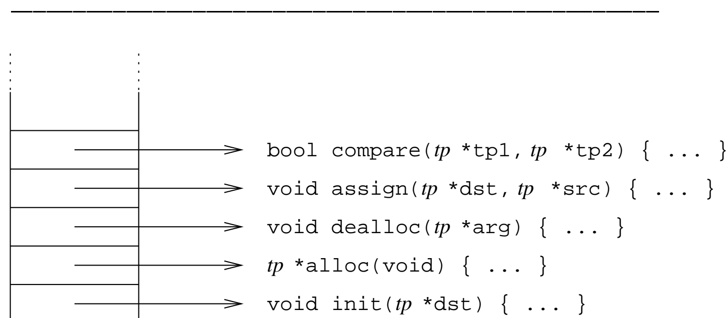


Figure 6.47 A dope vector for generic type *tp*.

```
void A(int i) {
    showSP("A.SP.entry");
    if (i > 0) {B(i-1);}
    showSP("A.SP.exit");
}
void B(int i) {
    showSP("B.SP.entry");
    if (i > 0) {A(i-1);}
    showSP("B.SP.exit");
}
```

Figure 6.48 Mutually recursive test routines.

```
void A(int i, void C()) {
    showSP("A.SP.entry");
    if (i > 0) {C(i-1, A);}
    showSP("A.SP.exit");
}
void B(int i, void C()) {
    showSP("B.SP.entry");
    if (i > 0) {C(i-1, B);}
    showSP("B.SP.exit");
}
```

Figure 6.49 Routines for testing routines as parameters.

```
void A(int i, label L) {
    showSP("A.SP.entry");
    if (i > 0) {B(i-1, exit); return;}
    exit: showSP("A.SP.exit"); goto L;
}
void B(int i, label L) {
    showSP("B.SP.entry");
    if (i > 0) {A(i-1, exit); return;}
    exit: showSP("B.SP.exit"); goto L;
}
```

Figure 6.50 Routines for testing labels as parameters.

7

Functional programs

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

Figure 7.1 Functional specification of factorial in Haskell.

```
int fac(int n) {
    int product = 1;

    while (n > 0) {
        product *= n;
        n--;
    }
    return product;
}
```

Figure 7.2 Imperative definition of factorial in C.

Compiler phase	*Language aspect*
Lexical analyzer	Offside rule
Parser	List notation List comprehension Pattern matching
Context handling	Polymorphic type checking
Run-time system	Referential transparency Higher-order functions Lazy evaluation

Figure 7.3 Overview of handling functional language aspects.

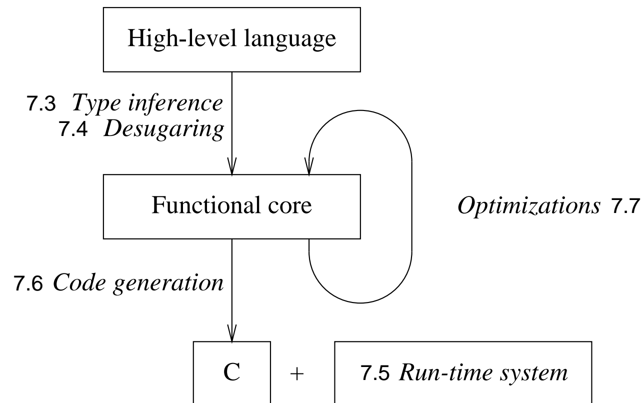


Figure 7.4 Structure of a functional compiler.

```
fun p1,1 ... p1,n = expr1
fun p2,1 ... p2,n = expr2
.
.
.
fun pm,1 ... pm,n = exprm
```

Figure 7.5 The general layout of a function using pattern matching.

```
fun a1 ... an  ⇒  if (cond1) then let defs1 in expr1
                  else if (cond2) then let defs2 in expr2
                  .
                  .
                  .
                  else if (condm) then let defsm in exprm
                  else error "fun: no matching pattern"
```

Figure 7.6 Translation of the pattern-matching function of Figure 7.5.

```
take a1 a2 = if (a1 == 0) then
               let xs = a2 in [ ]
             else if (a2 == [ ]) then
               let n = a1 in [ ]
             else if (_type_constr a2 == Cons) then
               let
                 n  = a1
                 x  = _type_field 1 a2
                 xs = _type_field 2 a2
               in
                 x : take (n-1) xs
             else
               error "take: no matching pattern"
```

Figure 7.7 Translation of the function `take`.

```
take a1 a2 = if (a1 == 0) then
               let xs = a2 in [ ]
             else if (a2 == [ ]) then
               let n = a1 in [ ]
             else
               let
                 n  = a1
                 x  = _type_field 1 a2
                 xs = _type_field 2 a2
               in
                 x : take (n-1) xs
```

Figure 7.8 Optimized translation of the function `take`.

```
last a1 = if (_type_constr a1 == Cons) then
            let
              x  = _type_field 1 a1
              xs = _type_field 2 a1
            in
              if (xs == [ ]) then
                x
              else
                last xs
          else
            error "last: no matching pattern"
```

Figure 7.9 Translation of the function `last`.

T{ [*expr* |] }	⇒	[*expr*]	(1)
T{ [*expr* | *F* , *Q*] }	⇒	`if` (*F*) `then` T{ [*expr* | *Q*] } `else` [ ]	(2)
T{ [*expr* | *e* <- *L* , *Q*] }	⇒	`mappend` $\mathtt{f}_Q$ *L* `where` $\mathtt{f}_Q$ *e* = T{ [*expr* | *Q*] }	(3)

Figure 7.10 Translation scheme for list comprehensions.

```
sv_mul scal vec = let
                    s_mul x = scal * x
                  in
                    map s_mul vec
```

Figure 7.11 A function returning a function.

```
sv_mul_dot_s_mul scal x = scal * x

sv_mul scal vec = map (sv_mul_dot_s_mul scal) vec
```

Figure 7.12 The function of Figure 7.11, with `s_mul` lambda-lifted.

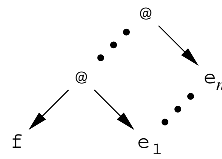


Figure 7.13 Identifying an essential redex.

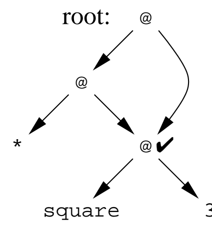


Figure 7.14 A graph with a built-in operator with strict arguments.

```
typedef enum {FUNC, NUM, NIL, CONS, APPL} node_type;

typedef struct node *Pnode;

typedef Pnode (*unary)(Pnode *arg);

struct function_descriptor {
    int arity;
    const char *name;
    unary code;
};

struct node {
    node_type tag;
    union {
        struct function_descriptor func;
        int num;
        struct {Pnode hd; Pnode tl;} cons;
        struct {Pnode fun; Pnode arg;} appl;
    } nd;
};

/* Constructor functions */
extern Pnode Func(int arity, const char *name, unary code);
extern Pnode Num(int num);
extern Pnode Nil(void);
extern Pnode Cons(Pnode hd, Pnode tl);
extern Pnode Appl(Pnode fun, Pnode arg);
```

Figure 7.15 Declaration of node types and constructor functions.

```
    Pnode Appl(const Pnode fun, Pnode arg) {
        Pnode node = (Pnode) heap_alloc(sizeof(*node));

        node->tag = APPL;
        node->nd.appl.fun = fun;
        node->nd.appl.arg = arg;
        return node;
    }
```

Figure 7.16 The node constructor function for application (@) nodes.

```
#include "node.h"
#include "eval.h"

#define STACK_DEPTH     10000

static Pnode arg[STACK_DEPTH];
static int top = STACK_DEPTH;                    /* grows down */

Pnode eval(Pnode root) {
    Pnode node = root;
    int frame, arity;

    frame = top;
    for (;;) {
        switch (node->tag) {
        case APPL:                                    /* unwind */
            arg[--top] = node->nd.appl.arg;  /* stack argument */
            node = node->nd.appl.fun;      /* application node */
            break;

        case FUNC:
            arity = node->nd.func.arity;
            if (frame-top < arity) {       /* curried function */
                top = frame;
                return root;
            }
            node = node->nd.func.code(&arg[top]);    /* reduce */
            top += arity;                 /* unstack arguments */
            *root = *node;              /* update root pointer */
            break;

        default:
            return node;
        }
    }
}
```

Figure 7.17 Reduction engine.

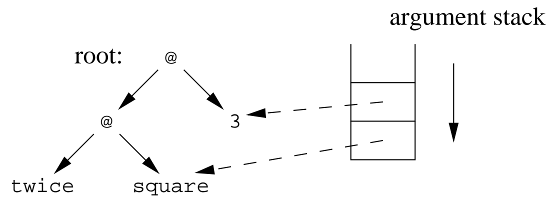


Figure 7.18 Stacking the arguments while searching for a redex.

```
Pnode mul(Pnode _arg0, Pnode _arg1) {
    Pnode a = eval(_arg0);
    Pnode b = eval(_arg1);

    return Num(a->nd.num * b->nd.num);
}

Pnode _mul(Pnode *arg) {
    return mul(arg[0], arg[1]);
}

struct node __mul = {FUNC, {2, "mul", _mul}};
```

Figure 7.19 Code and function descriptor of the built-in operator `mul`.

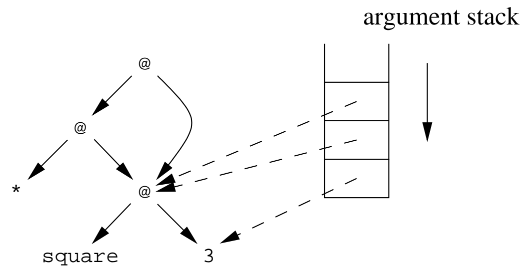


Figure 7.20 A more complicated argument stack.

```
Pnode conditional(Pnode _arg0, Pnode _arg1, Pnode _arg2) {
    Pnode cond = _arg0;
    Pnode _then = _arg1;
    Pnode _else = _arg2;

    return eval(cond)->nd.num ? _then : _else;
}

Pnode _conditional(Pnode *arg) {
    return conditional(arg[0], arg[1], arg[2]);
}

struct node __conditional =
    {FUNC, {3, "conditional", _conditional}};
```

Figure 7.21 Code and function descriptor of the built-in conditional operator.

```
f a = let
        b = fac a
        c = b * b
      in
        c + c
```

Figure 7.22 Example of a let-expression.

```
Pnode f(Pnode _arg0) {
    Pnode a = _arg0;
    Pnode b = Appl(&__fac, a);
    Pnode c = Appl(Appl(&__mul, b), b);

    return Appl(Appl(&__add, c), c);
}
```

Figure 7.23 Translation of the let-expression of Figure 7.22.

```
Pnode average(Pnode _arg0, Pnode _arg1) {
    Pnode a = _arg0;
    Pnode b = _arg1;

    return
        Appl(
            Appl(
                &__div,
                Appl(
                    Appl(&__add, a),
                    b
                )
            ),
            Num(2)
        );
}
```

Figure 7.24 Translation of the function `average`.

```
Pnode average(Pnode _arg0, Pnode _arg1) {
    Pnode a = _arg0;
    Pnode b = _arg1;

    return div(Appl(Appl(&__add, a), b), Num(2));
}
```

Figure 7.25 Suppressing the top-level spine in the translation of the function `average`.

```
Pnode average(Pnode _arg0, Pnode _arg1) {
    Pnode a = _arg0;
    Pnode b = _arg1;

    return div(add(a, b), Num(2));
}
```

Figure 7.26 Suppressing the argument spine to `div` in Figure 7.25.

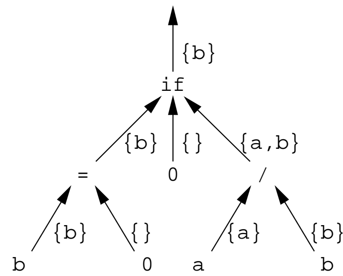


Figure 7.27 Flow of strictness information in the AST of `safe_div`.

Language construct	Set to be propagated	
L operator R	$L \cup R$	
`if` C `then` T `else` E	$C \cup (T \cap E)$	
fun_m @ A_1 @ ... @ A_n	$\bigcup_{\substack{\text{strict(fun},\, i) \\ i=1}}^{\min(m,n)} A_i$	
F @ A_1 @ ... @ A_n	F	
`let v =` V `in` E	$(E \backslash \{\texttt{v}\}) \cup V$	, if $\texttt{v} \in E$
	E	, otherwise

Figure 7.28 Rules for strictness information propagation.

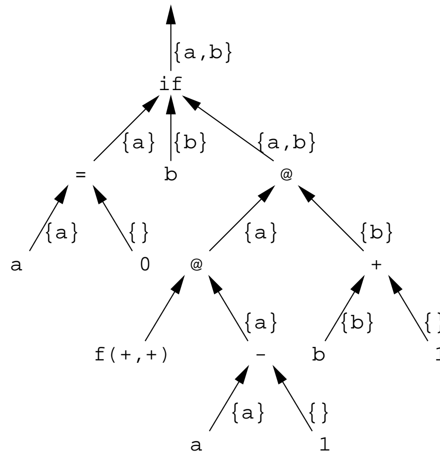


Figure 7.29 Optimistic strictness analysis of the function `f`.

Data definitions:

1. Let *Strict* be an associative mapping from (identifier, int) pairs to Booleans, where each pair describes a function name and argument index.

Initializations:

1. For each function *f* with arity *m* set *Strict*[*f*, *i*] to true for arguments *i* ranging from 1 to *m*.

Inference rules:

1. If propagation of strictness information through the AST of function *f* shows that *f* is not strict in argument *i*, then *Strict*[*f*, *i*] must be false.

Figure 7.30 Closure algorithm for strictness predicate.

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

    return equal(n, Num(0))->nd.num ? Num(1)
        : mul(n, fac(Appl(Appl(&__sub, n), Num(1))));
}
```

Figure 7.31 Lazy translation of the argument of `fac`.

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

    return equal(n, Num(0))->nd.num ? Num(1)
        : mul(n, fac(sub(n, Num(1))));
}
```

Figure 7.32 The first step in exploiting the strictness of the argument of `fac`.

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

    return equal_evaluated(n, Num(0))->nd.num ? Num(1)
        : mul_evaluated(n, fac_evaluated(sub_evaluated(n, Num(1))));
}

Pnode fac(Pnode _arg0) {
    return fac_evaluated(eval(_arg0));
}
```

Figure 7.33 Exploiting the strictness of `fac` and the built-in operators.

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

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

    return Num(fac_unboxed(n->nd.num));
}

Pnode fac(Pnode _arg0) {
    return fac_evaluated(eval(_arg0));
}
```

Figure 7.34 Strict, unboxed translation of the function `fac`.

```
Pnode drop_unboxed(int n, Pnode lst) {
    if (n == 0) {
        return lst;
    } else {
        return drop_unboxed(n-1, tail(lst));
    }
}
```

Figure 7.35 Strict, unboxed translation of the function `drop`.

```
Pnode drop_unboxed(int n, Pnode lst) {
L_drop_unboxed:
    if (n == 0) {
        return lst;
    } else {
        int   _tmp0 = n-1;
        Pnode _tmp1 = tail(lst);

        n = _tmp0; lst = _tmp1;
        goto L_drop_unboxed;
    }
}
```

Figure 7.36 Translation of `drop` with tail recursion eliminated.

```
fac n = if (n == 0) then 1 else prod n (n-1)
        where
          prod acc n = if (n == 0) then acc
                       else prod (acc*n) (n-1)
```

Figure 7.37 Tail-recursive `fac` with accumulating argument.

```
int fac_dot_prod_unboxed(int acc, int n) {
L_fac_dot_prod_unboxed:
    if (n == 0) {
        return acc;
    } else {
        int _tmp0 = n * acc;
        int _tmp1 = n-1;

        acc = _tmp0; n = _tmp1;
        goto L_fac_dot_prod_unboxed;
    }
}

int fac_unboxed(int n) {
    if (n == 0) {
        return 1;
    } else {
        return fac_dot_prod_unboxed(n, n-1);
    }
}
```

Figure 7.38 Accumulating translation of `fac`, with tail recursion elimination.

$$F\ \mathtt{x}_1 \ldots \mathtt{x}_n\ =\ \mathtt{if}\ B\ \mathtt{then}\ V\ \mathtt{else}\ E \oplus F\ E_1 \ldots E_n$$
$$\Rightarrow$$
$$F\ \mathtt{x}_1 \ldots \mathtt{x}_n\ =\ \mathtt{if}\ B\ \mathtt{then}\ V\ \mathtt{else}\ F'\ E\ E_1 \ldots E_n$$
$$F'\ \mathtt{x}_{acc}\ \mathtt{x}_1 \ldots \mathtt{x}_n\ =\ \mathtt{if}\ B\ \mathtt{then}\ \mathtt{x}_{acc} \oplus V\ \mathtt{else}\ F'\ (\mathtt{x}_{acc} \oplus E)\ E_1 \ldots E_n$$

Figure 7.39 Accumulating argument translation scheme.

8

Logic programs

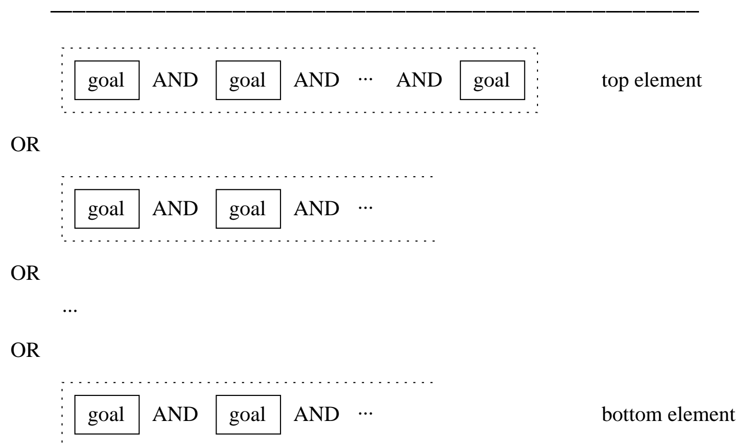


Figure 8.1 The goal list stack.

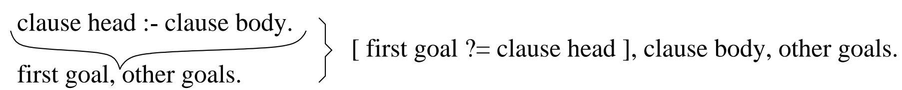


Figure 8.2 The action of 'Attach clauses' using one clause.

```
struct variable {
    char *name;
    struct term *term;
};

struct structure {
    char *functor;
    int arity;
    struct term **components;
};

typedef enum {Is_Constant, Is_Variable, Is_Structure} Term_Type;
typedef struct term {
    Term_Type type;
    union {
        char *constant;                  /* Is_Constant */
        struct variable variable;        /* Is_Variable */
        struct structure structure;      /* Is_Structure */
    } term;
} Term;
```

Figure 8.3 Data types for the construction of terms.

```
int unify_terms(Term *goal_arg, Term *head_arg) {
    /* Handle any bound variables: */
    Term *goal = deref(goal_arg);
    Term *head = deref(head_arg);

    if (goal->type == Is_Variable || head->type == Is_Variable) {
        return unify_unbound_variable(goal, head);
    }
    else {
        return unify_non_variables(goal, head);
    }
}
```

Figure 8.4 C code for the unification of two terms.

```
Term *deref(Term *t) {
    while (t->type == Is_Variable && t->term.variable.term != 0) {
        t = t->term.variable.term;
    }
    return t;
}
```

Figure 8.5 The function `deref()`.

```
int unify_unbound_variable(Term *goal, Term *head) {
    /* Handle identical variables: */
    if (goal == head) {
        /* Unification of identical variables is trivial */
    }
    else {
        /* Bind the unbound variable to the other term: */
        if (head->type == Is_Variable) {
            trail_binding(head);
            head->term.variable.term = goal;
        }
        else {
            trail_binding(goal);
            goal->term.variable.term = head;
        }
    }
    return 1;       /* variable unification always succeeds */
}
```

Figure 8.6 C code for unification involving at least one unbound variable.

```
int unify_non_variables(Term *goal, Term *head) {
    /* Handle terms of different type: */
    if (goal->type != head->type) return 0;

    switch (goal->type) {
    case Is_Constant:            /* both are constants */
        return unify_constants(goal, head);
    case Is_Structure:           /* both are structures */
        return unify_structures(
            &goal->term.structure, &head->term.structure
        );
    }
}
```

Figure 8.7 C code for the unification of two non-variables.

```
int unify_constants(Term *goal, Term *head) {
    return strcmp(goal->term.constant, head->term.constant) == 0;
}
```

Figure 8.8 C code for the unification of two constants.

```
int unify_structures(
    struct structure *s_goal, struct structure *s_head
) {
    int counter;

    if (s_goal->arity != s_head->arity
    ||  strcmp(s_goal->functor, s_head->functor) != 0
    ) return 0;

    for (counter = 0; counter < s_head->arity; counter++) {
        if (!unify_terms(
            s_goal->components[counter], s_head->components[counter]
        ))  return 0;
    }

    return 1;
}
```

Figure 8.9 C code for the unification of two structures.

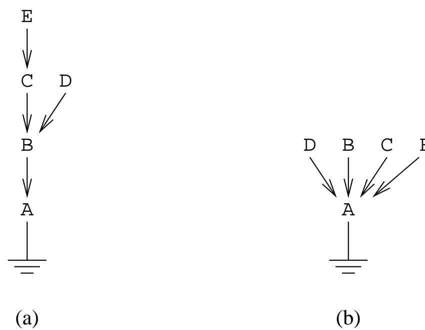


Figure 8.10 Examples of variable–variable bindings.

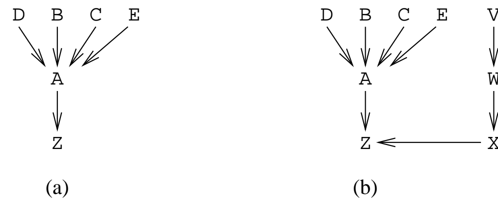


Figure 8.11 Examples of more complicated variable–variable bindings.

```
void number_pair_list(void (*Action)(int v1, int v2)) {
    (*Action)(1, 3);
    (*Action)(4, 6);
    (*Action)(7, 3);
}
```

Figure 8.12 An example of a list procedure.

```
int fs_sum;

void fs_test_sum(int v1, int v2) {
    if (v1 + v2 == fs_sum) {
        printf("Solution found: %d, %d\n", v1, v2);
    }
}

void find_sum(int sum) {
    fs_sum = sum;
    number_pair_list(fs_test_sum);
}
```

Figure 8.13 An application of the list procedure.

```
void find_sum(int sum) {
    void test_sum(int v1, int v2) {
        if (v1 + v2 == sum) {
            printf("Solution found: %d, %d\n", v1, v2);
        }
    }

    number_pair_list(test_sum);
}
```

Figure 8.14 An application of the list procedure, using nested routines.

```
void parent_2(
    Term *goal_arg1,
    Term *goal_arg2,
    Action goal_list_tail
) {
    ... /* the local routines parent_2_clause_[1-5]() */

    /* OR of all clauses for parent/2 */
    parent_2_clause_1();    /* parent(arne, james). */
    parent_2_clause_2();    /* parent(arne, sachiko). */
    parent_2_clause_3();    /* parent(koos, rivka). */
    parent_2_clause_4();    /* parent(sachiko, rivka). */
    parent_2_clause_5();    /* parent(truitje, koos). */
}
```

Figure 8.15 Translation of the relation `parent/2`.

```
/* translation of 'parent(arne, sachiko).' */
void parent_2_clause_2(void) {

    /* translation of 'arne, sachiko).' */
    void parent_2_clause_2_arg_1(void) {

        /* translation of 'sachiko).' */
        void parent_2_clause_2_arg_2(void) {

            /* translation of '.' */
            void parent_2_clause_2_body(void) {
                goal_list_tail();
            }

            /* translation of head term 'sachiko)' */
            unify_terms(goal_arg2, put_constant("sachiko"),
                        parent_2_clause_2_body
            );
        }

        /* translation of head term 'arne, ' */
        unify_terms(goal_arg1, put_constant("arne"),
                    parent_2_clause_2_arg_2
        );
    }

    /* translation of 'parent(' */
    parent_2_clause_2_arg_1();
}
```

Figure 8.16 Generated clause routine for `parent(arne, sachiko)`.

```
void unify_unbound_variable(
    Term *goal_arg, Term *head_arg, Action goal_list_tail
) {
    if (goal_arg == head_arg) {
        /* Unification of identical variables succeeds trivially */
        goal_list_tail();
    }
    else {
        /* Bind the unbound variable to the other term: */
        if (head_arg->type == Is_Variable) {
            head_arg->term.variable.term = goal_arg;
            goal_list_tail();
            head_arg->term.variable.term = 0;
        }
        else {
            goal_arg->term.variable.term = head_arg;
            goal_list_tail();
            goal_arg->term.variable.term = 0;
        }
    }
}
```

Figure 8.17 C code for full backtracking variable unification.

```
void query(void) {
    Term *X =  put_variable("X");

    void display_X(void) {
        print_term(X);
        printf("\n");
    }

    parent_2(put_constant("arne"), X, display_X);
}
```

Figure 8.18 Possible generated code for the query `?- parent(arne, X).`

```
void grandparent_2(
    Term *goal_arg1, Term *goal_arg2, Action goal_list_tail
) {
    /* translation of 'grandparent(X,Z):-parent(X,Y),parent(Y,Z).' */
    void grandparent_2_clause_1(void) {

        /* translation of 'X,Z):-parent(X,Y),parent(Y,Z).' */
        void grandparent_2_clause_1_arg_1(void) {
            Term *X = put_variable("X");

            /* translation of 'Z):-parent(X,Y),parent(Y,Z).' */
            void grandparent_2_clause_1_arg_2(void) {
                Term *Z = put_variable("Z");

                /* translation of body 'parent(X,Y),parent(Y,Z).' */
                void grandparent_2_clause_1_body(void) {
                    Term *Y = put_variable("Y");

                    /* translation of 'parent(Y,Z).' */
                    void grandparent_2_clause_1_goal_2(void) {
                        parent_2(Y, goal_arg2, goal_list_tail);
                    }

                    /* translation of 'parent(X,Y),' */
                    parent_2(goal_arg1, Y, grandparent_2_clause_1_goal_2);
                }

                /* translation of head term 'Z):-' */
                unify_terms(goal_arg2, Z, grandparent_2_clause_1_body);
            }

            /* translation of head term 'X,' */
            unify_terms(goal_arg1, X, grandparent_2_clause_1_arg_2);
        }

        /* translation of 'grandparent(' */
        grandparent_2_clause_1_arg_1();
    }

    /* OR of all clauses for grandparent/2 */
    grandparent_2_clause_1();
}
```

Figure 8.19 Clause routine generated for `grandparent/2`.

```
void parent_2(
    Term *goal_arg1,
    Term *goal_arg2,
    Action goal_list_tail
) {
    ... /* the local routines parent_2_clause_[1-5]() */

    /* switch_on_term(L_Variable, L_Constant, L_Structure) */
    switch (deref(goal_arg1)->type) {
    case Is_Variable:   goto L_Variable;
    case Is_Constant:   goto L_Constant;
    case Is_Structure:  goto L_Structure;
    }

    ... /* code labeled by L_Variable, L_Constant, L_Structure */

L_fail:;
}
```

Figure 8.20 Optimized translation of the relation `parent/2`.

```
    L_Variable:
        /* OR of all clauses for parent/2 */
        parent_2_clause_1();    /* parent(arne, james). */
        parent_2_clause_2();    /* parent(arne, sachiko). */
        parent_2_clause_3();    /* parent(koos, rivka). */
        parent_2_clause_4();    /* parent(sachiko, rivka). */
        parent_2_clause_5();    /* parent(truitje, koos). */
        goto L_fail;
```

Figure 8.21 Code generated for the label `L_Variable`.

```
L_Constant:
    /* switch_on_constant(8, constant_table) */
    switch (
        entry(deref(goal_arg1)->term.constant, 8, constant_table)
    ) {
    case ENTRY_arne:    goto L_Constant_arne;
    case ENTRY_koos:    goto L_Constant_koos;
    case ENTRY_sachiko: goto L_Constant_sachiko;
    case ENTRY_truitje: goto L_Constant_truitje;
    }
    goto L_fail;

L_Constant_arne:
    parent_2_clause_1();    /* parent(arne, james). */
    parent_2_clause_2();    /* parent(arne, sachiko). */
    goto L_fail;

L_Constant_koos:
    parent_2_clause_3();    /* parent(koos, rivka). */
    goto L_fail;

L_Constant_sachiko:
    parent_2_clause_4();    /* parent(sachiko, rivka). */
    goto L_fail;

L_Constant_truitje:
    parent_2_clause_5();    /* parent(truitje, koos). */
    goto L_fail;
```

Figure 8.22 Code generated for the labels `L_Constant_...`.

```
L_Structure:
    goto L_fail;
```

Figure 8.23 Code generated for the label `L_Structure`.

```
void first_gp_2(
  Term *goal_arg1, Term *goal_arg2, Action goal_list_tail
) {
/* label declarations required by GNU C for non-local jump */
__label__ L_cut;

  /* translation of 'first_gp(X,Z):-parent(X,Y),!,parent(Y,Z).' */
  void first_gp_2_clause_1(void) {
    /* translation of 'X,Z):-parent(X,Y),!,parent(Y,Z).' */
    void first_gp_2_clause_1_arg_1(void) {
      Term *X = put_variable("X");
      /* translation of 'Z):-parent(X,Y),!,parent(Y,Z).' */
      void first_gp_2_clause_1_arg_2(void) {
        Term *Z = put_variable("Z");
        /* translation of body 'parent(X,Y),!,parent(Y,Z).' */
        void first_gp_2_clause_1_body(void) {
          Term *Y = put_variable("Y");
          /* translation of '!,parent(Y,Z).' */
          void first_gp_2_clause_1_goal_2(void) {
            /* translation of 'parent(Y,Z).' */
            void first_gp_2_clause_1_goal_3(void) {
              parent_2(Y, goal_arg2, goal_list_tail);
            }
            /* translation of '!,' */
            first_gp_2_clause_1_goal_3();

            goto L_cut;
          }
          /* translation of 'parent(X,Y),' */
          parent_2(goal_arg1, Y, first_gp_2_clause_1_goal_2);
        }
        /* translation of head term 'Z):-' */
        unify_terms(goal_arg2, Z, first_gp_2_clause_1_body);
      }
      /* translation of head term 'X,' */
      unify_terms(goal_arg1, X, first_gp_2_clause_1_arg_2);
    }
    /* translation of 'first_gp(' */
    first_gp_2_clause_1_arg_1();
  }
  /* OR of all clauses for first_gp/2 */
  first_gp_2_clause_1();

L_fail:;
L_cut:;
}
```

Figure 8.24 Clause routine generated for `first_grandparent/2`.

```
void parent_2(
    Term *goal_arg1,
    Term *goal_arg2,
    Action goal_list_tail
) {
    ... /* the local routines parent_2_clause_[1-5]() */

    /* guarded OR of all clauses for parent/2 */
    if (!retracted_parent_2_clause_1)
        parent_2_clause_1();    /* parent(arne, james). */
    if (!retracted_parent_2_clause_2)
        parent_2_clause_2();    /* parent(arne, sachiko). */
    if (!retracted_parent_2_clause_3)
        parent_2_clause_3();    /* parent(koos, rivka). */
    if (!retracted_parent_2_clause_4)
        parent_2_clause_4();    /* parent(sachiko, rivka). */
    if (!retracted_parent_2_clause_5)
        parent_2_clause_5();    /* parent(truitje, koos). */
L_fail:
    if (number_of_clauses_added_at_end_for_parent_2 > 0) {
        stack_argument(goal_arg2);
        stack_argument(goal_arg1);
        stack_argument(put_constant("parent/2"));
        interpret(goal_list_tail);
    }
L_cut:;
}
```

Figure 8.25 Translation of `parent/2` with code for controlling asserted and retracted clauses.

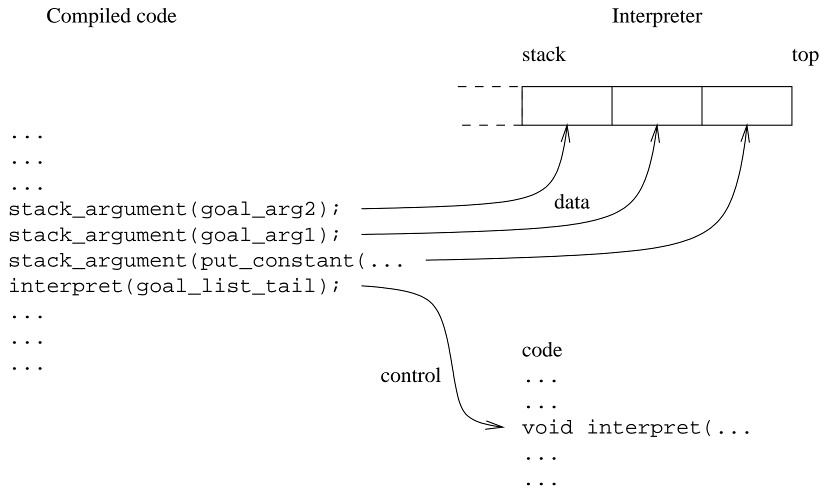


Figure 8.26 Switch to interpretation of added clauses.

```
void do_relation(const char *relation, Action goal_list_tail) {
    switch (entry(relation, N, relation_table)) {
    case ENTRY_...:
    case ENTRY_parent_2:
        goal_arg1 = unstack_argument();
        goal_arg2 = unstack_argument();
        parent_2(goal_arg1, goal_arg2, goal_list_tail);
        break;
    case ENTRY_grandparent_2:
        ...
        break;
    case ENTRY_...:
    ...
    default:
        interpret_asserted_relation(relation, goal_list_tail);
        break;
    }
}
```

Figure 8.27 Dispatch routine for goals in asserted clauses.

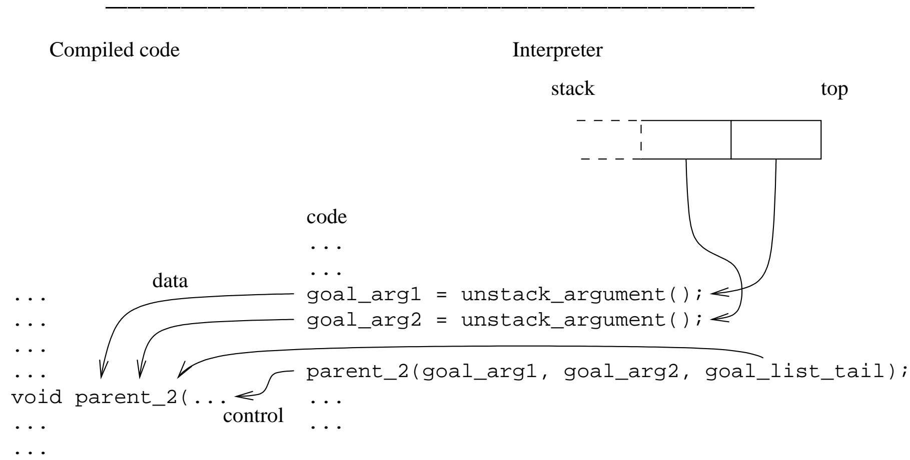


Figure 8.28 Switch to execution of compiled clauses.

```
/* translation of 'parent(arne, sachiko).' */
void parent_2_clause_2(void) {
    Trail_Mark trail_mark;

    set_trail_mark(&trail_mark);
    /* translation of '(arne, sachiko)' */
    if (unify_terms(goal_arg1, put_constant("arne"))
    &&  unify_terms(goal_arg2, put_constant("sachiko"))
    ) {
        /* translation of '.' */
        void parent_2_clause_2_body(void) {
            goal_list_tail();
        }

        /* translation of '.' */
        parent_2_clause_2_body();
    }
    restore_bindings_until_trail_mark(&trail_mark);
}
```

Figure 8.29 Clause routine with flat unification for `parent(arne, sachiko)`.

```
/* optimized translation of 'parent(arne, sachiko).' */
void parent_2_clause_2(void) {
    Trail_Mark trail_mark;

    set_trail_mark(&trail_mark);
    /* optimized translation of '(arne, sachiko)' */
    UNIFY_CONSTANT(goal_arg1, "arne");              /* macro */
    UNIFY_CONSTANT(goal_arg2, "sachiko");           /* macro */

    /* optimized translation of '.' */
    goal_list_tail();
L_fail:
    restore_bindings_until_trail_mark(&trail_mark);
}
```

Figure 8.30 Clause routine with compiled unification for `parent(arne, sachiko)`.

```
g = deref(goal_arg);
if (g->type == Is_Variable) {
    trail_binding(g);
    g->term.variable.term = put_constant(head_text);
}
else {
    if (g->type != Is_Constant) goto L_fail;
    if (strcmp(g->term.constant, head_text) != 0) goto L_fail;
}
```

Figure 8.31 The unification instruction `UNIFY_CONSTANT`.

```
v = deref(head_var);         /* head_var may already be bound */
g = deref(goal_arg);
if (v->type == Is_Variable) {
    if (g != v) {
        trail_binding(v);
        v->term.variable.term = g;
    }
}
else {
    /* no further compilation possible */
    /* call interpreter */
    if (!unify_terms(g, v)) goto L_fail;
}
```

Figure 8.32 The unification instruction `UNIFY_VARIABLE`.

```
int unify_terms(Term *goal_arg, Term *head_arg) {
    /* Handle any bound variables: */
    Term *goal = deref(goal_arg);
    Term *head = deref(head_arg);

    if (goal->type == Is_Variable || head->type == Is_Variable) {
        /* Handle identical variables: */
        if (goal == head) return 1;

        /* Bind the unbound variable to the other term: */
        if (head->type == Is_Variable) {
            trail_binding(head);
            head->term.variable.term = goal;
        }
        else {
            trail_binding(goal);
            goal->term.variable.term = head;
        }
        return 1;           /* always succeeds */
    }
    else {
        /* Handle terms of different type: */
        if (goal->type != head->type) return 0;

        switch (goal->type) {
        case Is_Constant:           /* both are constants */
            return strcmp(goal->term.constant, head->term.constant) == 0;
        case Is_Structure:          /* both are structures */
            return unify_structures(
                &goal->term.structure, &head->term.structure
            );
        }
    }
}
```

Figure 8.33 Deriving the instruction `UNIFY_CONSTANT`, stage 1.

```
int unify_constant(Term *goal_arg, char *head_text) {
    Term *goal;

    /* Handle bound goal_arg: */
    goal = deref(goal_arg);

    if (goal->type == Is_Variable
    ||  (put_constant(head_text))->type == Is_Variable
    ) {
        /* Handle identical variables: */
        if (goal == (put_constant(head_text))) return 1;

        /* Bind the unbound variable to the other term: */
        if ((put_constant(head_text))->type == Is_Variable) {
            trail_binding((put_constant(head_text)));
            (put_constant(head_text))->term.variable.term = goal;
        }
        else {
            trail_binding(goal);
            goal->term.variable.term = (put_constant(head_text));
        }
        return 1;           /* always succeeds */
    }
    else {
        /* Handle terms of different type: */
        if (goal->type != (put_constant(head_text))->type) return 0;

        switch (goal->type) {
        case Is_Constant:           /* both are constants */
            return
                strcmp(
                    goal->term.constant,
                    (put_constant(head_text))->term.constant
                ) == 0;
        case Is_Structure:          /* both are structures */
            return unify_structures(
                &goal->term.structure,
                &(put_constant(head_text))->term.structure
            );
        }
    }
}
```

Figure 8.34 Deriving the instruction `UNIFY_CONSTANT`, stage 2.

```
int unify_constant(Term *goal_arg, char *head_text) {
    Term *goal;

    /* Handle bound goal_arg: */
    goal = deref(goal_arg);

    if (goal->type == Is_Variable || Is_Constant == Is_Variable) {
        /* Handle identical variables: */
        if (goal == (put_constant(head_text))) return 1;

        /* Bind the unbound variable to the other term: */
        if (Is_Constant == Is_Variable) {
            trail_binding((put_constant(head_text)));
            ERRONEOUS = goal;
        }
        else {
            trail_binding(goal);
            goal->term.variable.term = (put_constant(head_text));
        }
        return 1;           /* always succeeds */
    }
    else {
        /* Handle terms of different type: */
        if (goal->type != Is_Constant) return 0;

        switch (goal->type) {
        case Is_Constant:           /* both are constants */
            return strcmp(goal->term.constant, head_text) == 0;
        case Is_Structure:          /* both are structures */
            return unify_structures(
                &goal->term.structure, ERRONEOUS
            );
        }
    }
}
```

Figure 8.35 Deriving the instruction UNIFY_CONSTANT, stage 3.

```
int unify_constant(Term *goal_arg, char *head_text) {
    Term *goal;

    /* Handle bound goal_arg: */
    goal = deref(goal_arg);

    if (goal->type == Is_Variable || 0) {
        /* Handle identical variables: */
        if (0) return 1;

        /* Bind the unbound variable to the other term: */
        if (0) {
            trail_binding((put_constant(head_text)));
            ERRONEOUS = goal;
        }
        else {
            trail_binding(goal);
            goal->term.variable.term = (put_constant(head_text));
        }
        return 1;           /* always succeeds */
    }
    else {
        /* Handle terms of different type: */
        if (goal->type != Is_Constant) return 0;

        switch (Is_Constant) {
        case Is_Constant:           /* both are constants */
            return strcmp(goal->term.constant, head_text) == 0;
        case Is_Structure:          /* both are structures */
            return unify_structures(
                &goal->term.structure, ERRONEOUS
            );
        }
    }
}
```

Figure 8.36 Deriving the instruction `UNIFY_CONSTANT`, stage 4.

```
int unify_constant(Term *goal_arg, char *head_text) {
    Term *goal;

    /* Handle bound goal_arg: */
    goal = deref(goal_arg);

    if (goal->type == Is_Variable) {
        trail_binding(goal);
        goal->term.variable.term = (put_constant(head_text));
        return 1;           /* always succeeds */
    }
    else {
        if (goal->type != Is_Constant) return 0;
        return strcmp(goal->term.constant, head_text) == 0;
    }
}
```

Figure 8.37 Deriving the instruction UNIFY_CONSTANT, stage 5.

```
int unify_constant(Term *goal_arg, char *head_text) {
    Term *goal;

    goal = deref(goal_arg);
    if (goal->type == Is_Variable) {
        trail_binding(goal);
        goal->term.variable.term = put_constant(head_text);
    }
    else {
        if (goal->type != Is_Constant) goto L_fail;
        if (strcmp(goal->term.constant, head_text) != 0) goto L_fail;
    }
    return 1;
L_fail:
    return 0;
}
```

Figure 8.38 Deriving the instruction UNIFY_CONSTANT, final result.

```
g = deref(goal_arg);
if (g->type == Is_Variable) {
    trail_binding(g);
    g->term.variable.term =
        put_structure(head_functor, head_arity);
    initialize_components(g->term.variable.term);
}
else {
    if (g->type != Is_Structure) goto L_fail;
    if (g->term.structure.arity != head_arity
    ||  strcmp(g->term.structure.functor, head_functor) != 0
    ) goto L_fail;
}
```

Figure 8.39 The unification instruction `UNIFY_STRUCTURE`.

```
/* match times(2, X) */
UNIFY_STRUCTURE(goal_arg, "times", 2);              /* macro */
{   /* match (2, X) */
    Term **goal_comp = deref(goal_arg)->term.structure.components;

    UNIFY_CONSTANT(goal_comp[0], "2");              /* macro */
    UNIFY_VARIABLE(goal_comp[1], X);                /* macro */
}
```

Figure 8.40 Compiled code for the head `times(2, X)`.

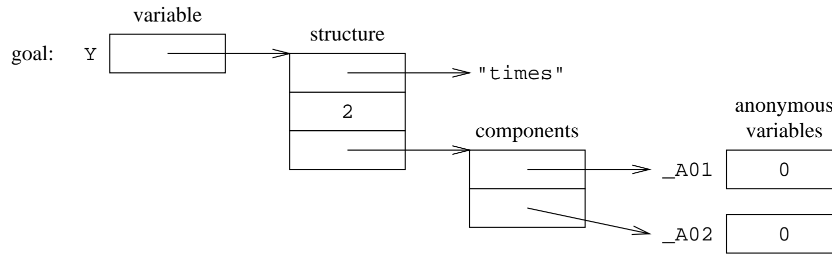


Figure 8.41 The effect of the instruction UNIFY_STRUCTURE.

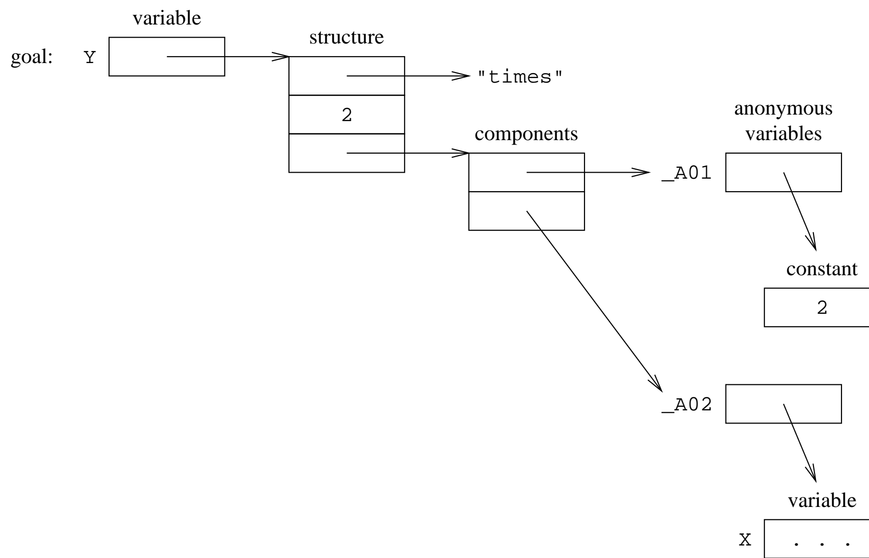


Figure 8.42 The effect of the instructions of Figure 8.40.

```
/* match times(2, X) */
mode_1 = mode;              /* save mode */
UNIFY_STRUCTURE(&goal_arg, "times", 2); /* macro */
    /* saves read/write mode and may set it to Write_Mode */
{   /* match (2, X) */
    register Term **goal_comp =
        deref(goal_arg)->term.structure.components;

    UNIFY_CONSTANT(&goal_comp[0], "2"); /* macro */
    UNIFY_VARIABLE(&goal_comp[1], X);   /* macro */
}
mode = mode_1;
```

Figure 8.43 Compiled code with read/write mode for the head `times(2, X)`.

```
if (mode == Read_Mode) {
    g = deref(*goal_ptr);
    if (g->type == Is_Variable) {
        trail_binding(g);
        g->term.variable.term = put_constant(head_text);
    }
    else {
        if (g->type != Is_Constant) goto L_fail;
        if (strcmp(g->term.constant, head_text) != 0) goto L_fail;
    }
}
else {  /* mode == Write_Mode */
    *goal_ptr = put_constant(head_text);
}
```

Figure 8.44 The instruction `UNIFY_CONSTANT` with read/write mode.

```
if (mode == Read_Mode) {
    g = deref(*goal_ptr);
    if (g->type == Is_Variable) {
        trail_binding(g);
        g->term.variable.term =
            put_structure(head_functor, head_arity);
        mode = Write_Mode;      /* signal uninitialized goals */
    }
    else {
        if (g->type != Is_Structure) goto L_fail;
        if (g->term.structure.arity != head_arity
        ||  strcmp(g->term.structure.functor, head_functor) != 0
        ) goto L_fail;
    }
}
else {  /* mode == Write_Mode */
    *goal_ptr = put_structure(head_functor, head_arity);
}
```

Figure 8.45 The instruction `UNIFY_STRUCTURE` with read/write mode.

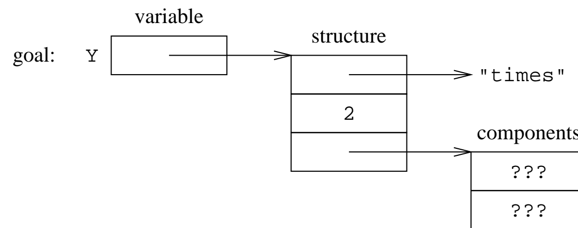


Figure 8.46 The effect of the instruction `UNIFY_STRUCT` with read/write mode.

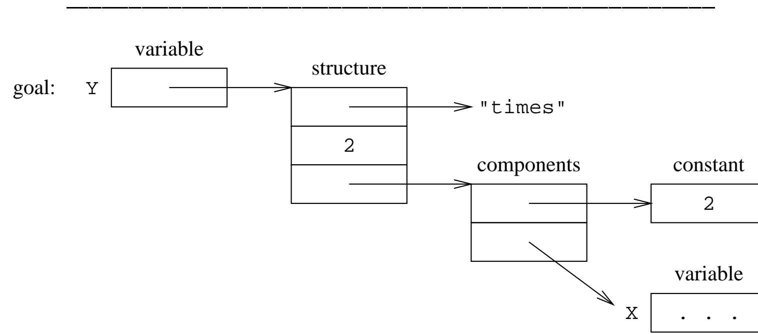
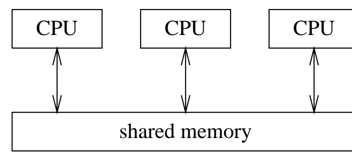


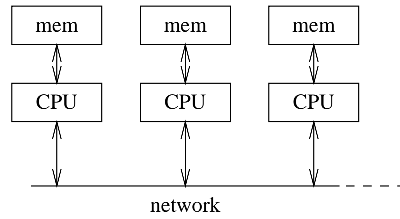
Figure 8.47 The effect of the instructions of Figure 8.40 with read/write mode.

9

Parallel and distributed programs



(a) multiprocessor



(b) multicomputer

Figure 9.1 Multiprocessors and multicomputers.

```
monitor BinMonitor;
    bin: integer;
    occupied: Boolean := false;
    full, empty: Condition;

    operation put(x: integer);
    begin
        while occupied do     # wait if the bin already is occupied
            wait(empty);
        od;
        bin := x;             # put the item in the bin
        occupied := true;
        signal(full);         # wake up a process blocked in get
    end;

    operation get(x: out integer);
    begin
        while not occupied do # wait if the bin is empty
            wait(full);
        od;
        x := bin;             # get the item from the bin
        occupied := false;
        signal(empty);        # wakeup a process blocked in put
    end;
end;
```

Figure 9.2 An example monitor.

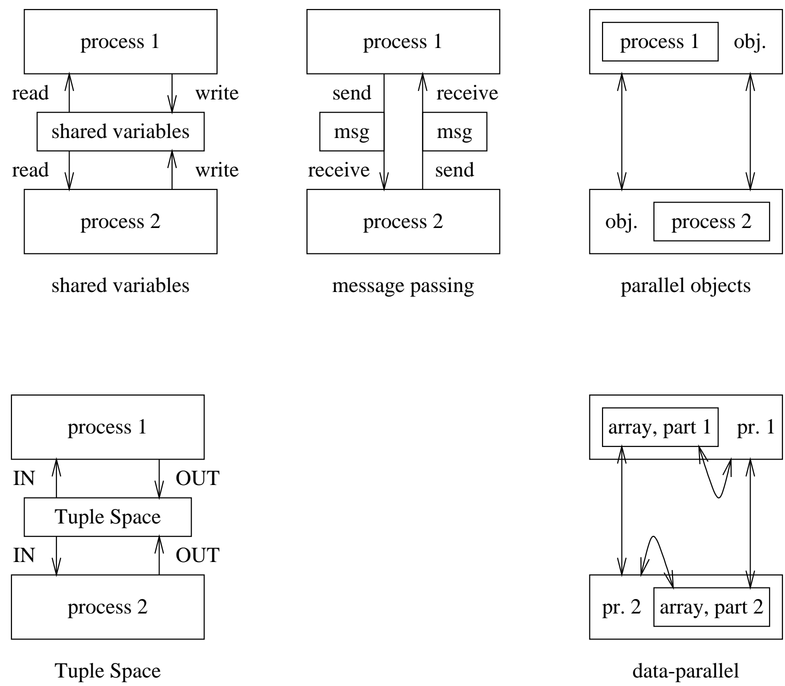


Figure 9.3 Summary of the five parallel programming models.

Status
Program counter
Stack pointer
Stack limits
Registers
Next-pointer

Figure 9.4 Structure of a thread control block.

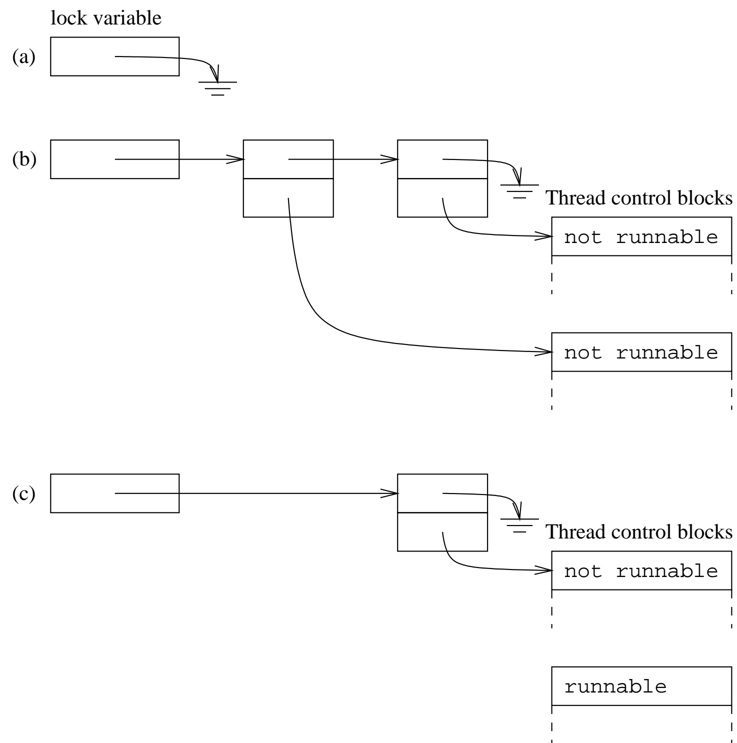


Figure 9.5 A lock variable containing a list of blocked threads.

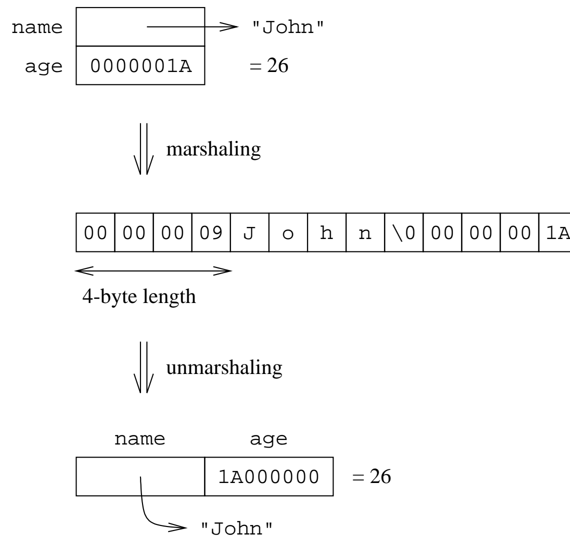


Figure 9.6 Marshaling.

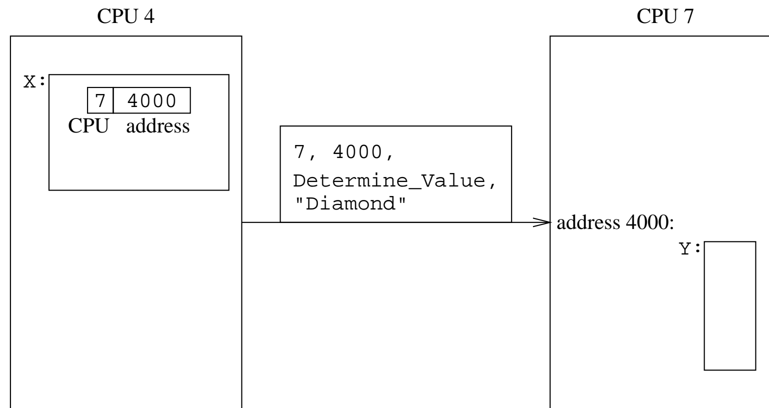


Figure 9.7 Using object identifiers for remote references.

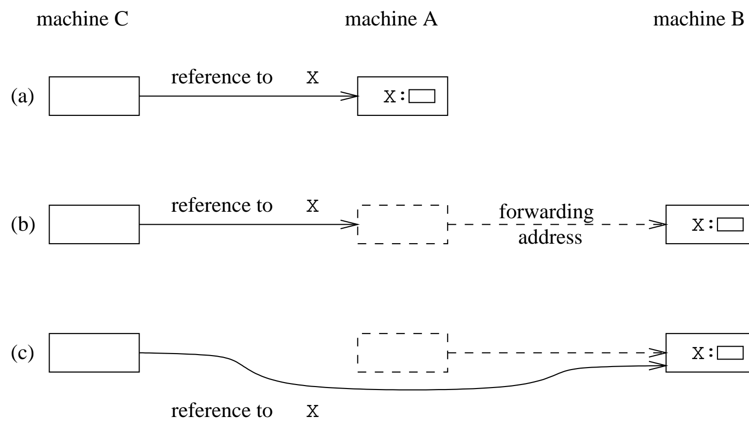


Figure 9.8 Pointer chasing.

```
int n, year;
float f;

     ...                /* code initializing n and f */

(1)  in("Olympic year", 1928);
(2)  in("Classic movie", "Way out West", 1937);
(3)  in("Classic movie", "Sons of the desert", 1933);
(4)  in("Classic movie", "Sons of the desert", ? &year);
(5)  in("Popular number", 65536);
(6)  in("Popular number", 3.14159);
(7)  in("Popular number", n);
(8)  in("Popular number", f);
```

Figure 9.9 A demo program in C/Linda.

Partition 1:
```
(1)  in("Olympic year", 1928);
```
Partition 2:
```
(2)  in("Classic movie", "Way out West", 1937);
```
Partition 3:
```
(3)  in("Classic movie", "Sons of the desert", 1933);
(4)  in("Classic movie", "Sons of the desert", ? &year);
```
Partition 4:
```
(5)  in("Popular number", 65536);
(7)  in("Popular number", n);
```
Partition 5:
```
(6)  in("Popular number", 3.14159);
(8)  in("Popular number", f);
```

Figure 9.10 The partitions for the demo C/Linda program in Figure 9.9.

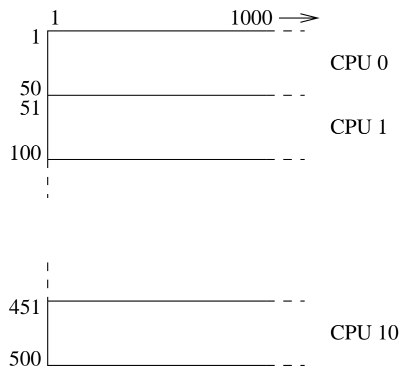


Figure 9.11 An array distributed over 10 CPUs.

```
INPUT:
    ID: number of this processor (ranging from 0 to P-1)
    lower_bound: number of first row assigned to this processor
    upper_bound: number of last row assigned to this processor

real A[lower_bound..upper_bound, 1000],
    B[lower_bound-1 .. upper_bound+1, 1000];

if ID > 0 then send B[lower_bound, *] to processor ID-1;
if ID < P-1 then send B[upper_bound, *] to processor ID+1;
if ID > 0 then receive B[lower_bound-1, *] from processor ID-1;
if ID < P-1 then receive B[upper_bound+1, *] from processor ID+1;
for i := lower_bound to upper_bound do
    for j := 1 to 1000 do
        A[i, j] := B[i-1, j] + B[i+1, j];
    od;
od;
```

Figure 9.12 Generated (pseudo-)code.

```
int Maximum(int value) {
    if (mycpu == 1) {
        /* the first CPU creates the tuple */
        out("max", value, 1);
    } else {
        /* other CPUs wait their turn */
        in("max", ? &max, mycpu);
        if (value > max) max = value;
        out("max", max, mycpu + 1);
    }
    /* wait until all CPUs are done */
    read("max", ? &max, P);
    return max;
}
```

Figure 9.13 C/Linda code for the operation `Maximum`.

Appendix A – A simple object-oriented compiler/interpreter

	declarations	simple expressions	assignments	for-loops
lexical analysis	lexical analysis of declarations			
parsing		parsing of simple expressions		
context handling	context handling of declarations			
optimization				optimization of for-loops
code generation			code generation for assignments	

Figure A.1 Table of tasks in a compiler with some tasks filled in.

```
expression → digit | parenthesized_expression
digit → DIGIT
parenthesized_expression → '(' expression operator expression ')'
operator → '+' | '*'
```

Figure A.2 The grammar for simple fully parenthesized expressions in AND/OR form.

```
class Demo_Compiler {
    public static void main(String[ ] args) {
        // Get the intermediate code
        Lex.Get_Token();            // start the lexical analyser
        Expression icode = Expression.Parse_Or();
        if (Lex.Token_Class != Lex.EOF) {
            System.err.println("Garbage at end of program ignored");
        }
        if (icode == null) {
            System.err.println("No intermediate code produced");
            return;
        }

        // Print it
        System.out.print("Expression: ");
        icode.Print();
        System.out.println();

        // Interpret it
        System.out.print("Interpreted: ");
        System.out.println(icode.Interpret());

        // Generate code for it
        System.out.println("Code:");
        icode.Generate_Code();
        System.out.println("PRINT");
    }
}
```

Figure A.3 Class `Demo_Compiler`, driver for the object-oriented demo compiler.

```
import java.io.IOException;

class Lex {
    public static int Token_Class;
    public static final int EOF = 256;
    public static final int DIGIT = 257;
    public static char Token_Char_Value;

    public static void Get_Token() {
        char ch;

        // Get non-whitespace character
        do {int ch_or_EOF;
            try {ch_or_EOF = System.in.read();}
            catch (IOException ex) {ch_or_EOF = -1;}
            if (ch_or_EOF < 0) {Token_Class = EOF; return;}
            ch = (char)ch_or_EOF;               // cast is safe
        } while (Character.isWhitespace(ch));

        // Classify it
        if (Character.isDigit(ch)) {
            Token_Class = DIGIT; Token_Char_Value = ch;
        }
        else {
            Token_Class = ch;
        }
    }
}
```

Figure A.4 Class `Lex`, lexical analyzer for the object-oriented demo compiler.

```
abstract class Expression {
    public static Expression Parse_Or() {
        // Try to parse a parenthesized_expression
        Parenthesized_Expression pe = new Parenthesized_Expression();
        if (pe.Parse_And()) return pe;

        // Try to parse a digit
        Digit d = new Digit();
        if (d.Parse_And()) return d;

        return null;
    }

    abstract public void Print();
    abstract public int Interpret();
    abstract public void Generate_Code();
}
```

Figure A.5 Class `Expression` for the object-oriented demo compiler.

```
class Digit extends Expression {
    private char digit = '\0';

    public boolean Parse_And() {
        if (Lex.Token_Class == Lex.DIGIT) {
            digit = Lex.Token_Char_Value;
            Lex.Get_Token();
            return true;
        }
        return false;
    }

    public void Print() {System.out.print(digit);}

    public int Interpret() {return digit - '0';}

    public void Generate_Code() {
        System.out.print("PUSH " + digit + "\n");
    }
}
```

Figure A.6 Class `Digit` for the object-oriented demo compiler.

```
class Parenthesized_Expression extends Expression {
    private Expression left;
    private Operator oper;
    private Expression right;

    public boolean Parse_And() {
        if (Lex.Token_Class == '(') {
            Lex.Get_Token();
            if ((left = Expression.Parse_Or()) == null) {
                System.err.println("Missing left expression");
                return false;
            }
            if ((oper = Operator.Parse_Or()) == null) {
                System.err.println("Missing operator");
                return false;
            }
            if ((right = Expression.Parse_Or()) == null) {
                System.err.println("Missing right expression");
                return false;
            }
            if (Lex.Token_Class != ')') {
                System.err.println("Missing ) added");
            }
            else {Lex.Get_Token();}
            return true;
        }
        return false;
    }

    public void Print() {
        System.out.print('(');
        left.Print(); oper.Print(); right.Print();
        System.out.print(')');
    }

    public int Interpret() {
        return oper.Interpret(left.Interpret(), right.Interpret());
    }

    public void Generate_Code() {
        left.Generate_Code();
        right.Generate_Code();
        oper.Generate_Code();
    }
}
```

Figure A.7 Class `Parenthesized_Expression` for the object-oriented demo compiler.

```
abstract class Operator {
    public static Operator Parse_Or() {
        // Try to parse a '+'
        Actual_Operator co_plus = new Actual_Operator('+');
        if (co_plus.Parse_And()) return co_plus;

        // Try to parse a '*'
        Actual_Operator co_times = new Actual_Operator('*');
        if (co_times.Parse_And()) return co_times;

        return null;
    }

    abstract public void Print();
    abstract public int Interpret(int e_left, int e_right);
    abstract public void Generate_Code();
}

class Actual_Operator extends Operator {
    private char oper;

    public Actual_Operator(char op) {oper = op;}

    public boolean Parse_And() {
        if (Lex.Token_Class == oper) {Lex.Get_Token(); return true;}
        return false;
    }

    public void Print() {System.out.print(oper);}

    public int Interpret(int e_left, int e_right) {
        switch (oper) {
        case '+': return e_left + e_right;
        case '*': return e_left * e_right;
        }
        return 0;
    }

    public void Generate_Code() {
        switch (oper) {
        case '+': System.out.print("ADD\n"); break;
        case '*': System.out.print("MULT\n"); break;
        }
    }
}
```

Figure A.8 Class `Operator` for the object-oriented demo compiler.

```
class A extends ... {
    private A1 a1;     // fields for the components
    private A2 a2;
    ...
    private An an;

    public boolean Parse_And() {
        // Try to parse the alternative A1 A2 ... An:
        if ((a1 = A1.Parse_Or()) != null) {
            if ((a2 = A2.Parse_Or()) == null)
                error("Missing A2");
            ...
            if ((an = An.Parse_Or()) == null)
                error("Missing An");
            return true;
        }
        return false;
    }
}
```

Figure A.9 The template for an AND non-terminal.

```
abstract class A {
    public static A Parse_Or() {

        // Try to parse an A1:
        A1 a1 = new A1();
        if (a1.Parse_And()) return a1;
        ...
        // Try to parse an An:
        An an = new An();
        if (an.Parse_And()) return an;

        return null;
    }
}
```

Figure A.10 The template for an OR non-terminal.

More answers to exercises

```
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.

```
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.

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

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

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

```
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.

```
%Start    Comment

Layout              ([ \t])

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

StartComment        ("/*")
EndComment          ("*/")
SafeCommentChar     ([^*\n])
UnsafeCommentChar   ("*")

%%

\"{StringChar}*\"   {printf("%s", yytext);}    /* string */
\'{CharChar}\'      {printf("%s", yytext);}    /* character */

{Layout}*{StartComment}         {BEGIN Comment;}
<Comment>{SafeCommentChar}+     {}  /* safe comment chunk */
<Comment>{UnsafeCommentChar}    {}  /* unsafe char, read one by one */
<Comment>"\n"                   {}  /* to break up long comments */
<Comment>{EndComment}           {BEGIN INITIAL;}
```

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

```
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.

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.

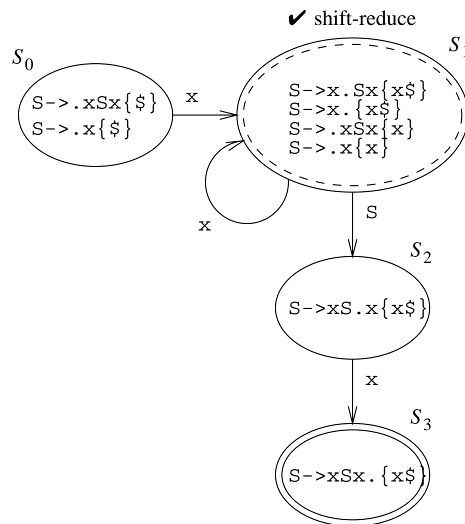


Figure Answers.9 The SLR(1) automaton for S → x S x | x.

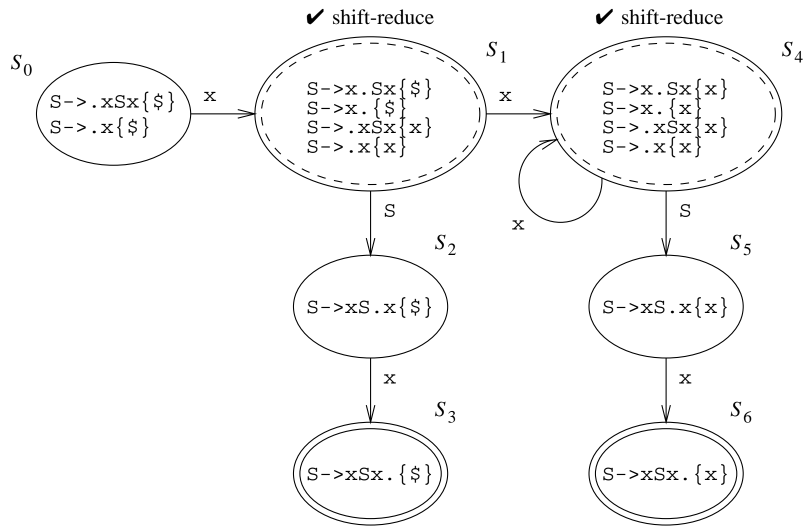


Figure Answers.10 The LR(1) automaton for `S → x S x | x`.

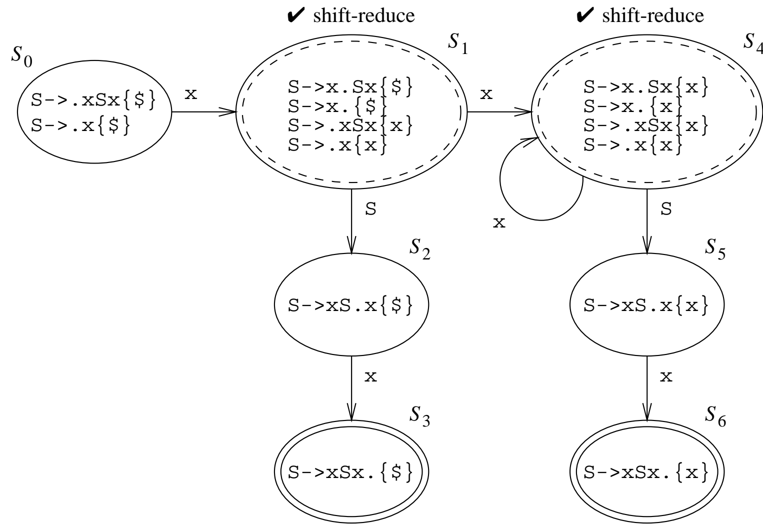


Figure Answers.11 The LALR(1) automaton for `S → x S x | x`.

```
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.

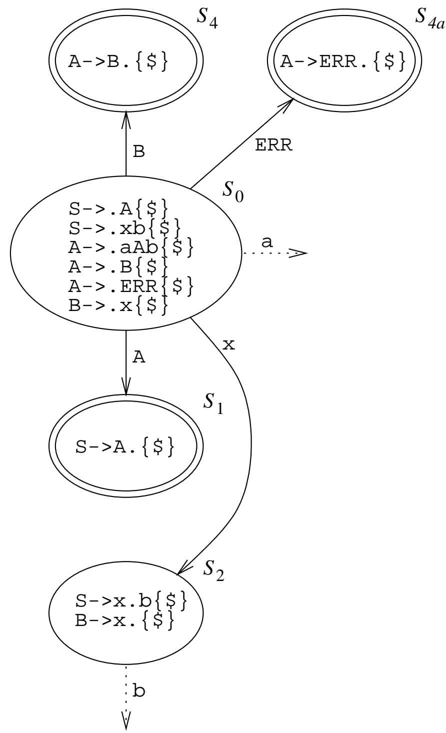


Figure Answers.13 State S_0 of the error-recovering parser.

```
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.

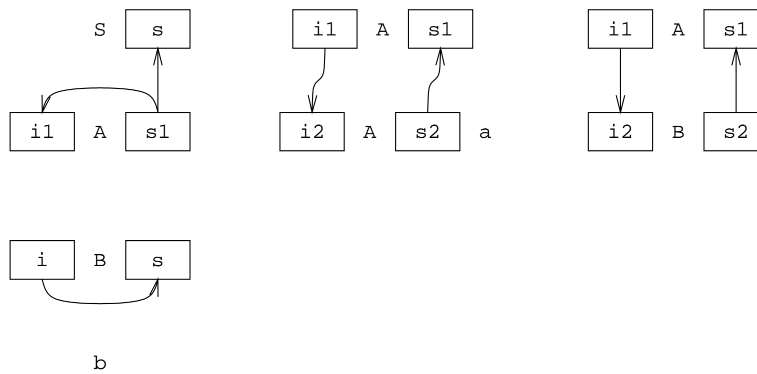


Figure Answers.15 Dependency graphs for S, A, and B.

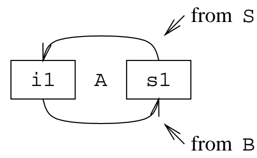


Figure Answers.16 IS-SI graph of A.

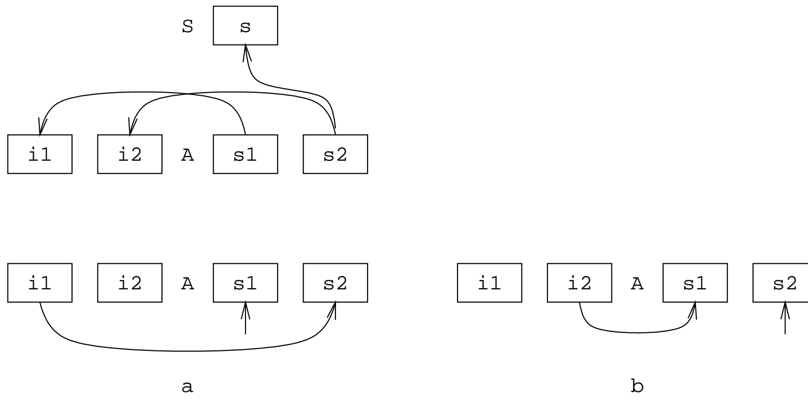
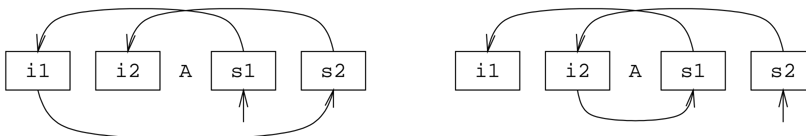


Figure Answers.17 Dependency graphs for S and A.

IS-SI graph set of A:



merged IS-SI graph of A:

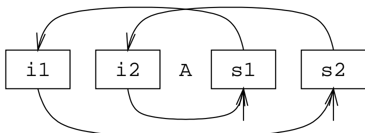


Figure Answers.18 IS-SI graph sets and IS-SI graph of A.

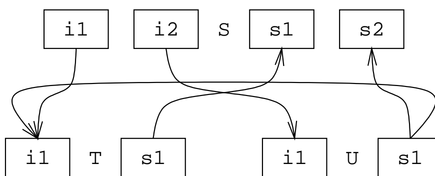


Figure Answers.19 Dependency graph of S.

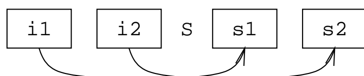


Figure Answers.20 IS-SI graph of S.

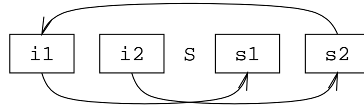


Figure Answers.21 IS-SI graph of S.

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().

```
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`.

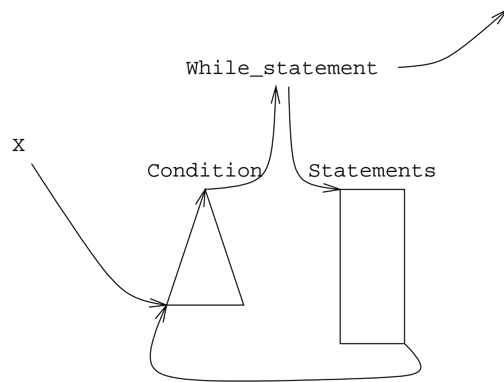


Figure Answers.24 Threaded AST of the while statement.

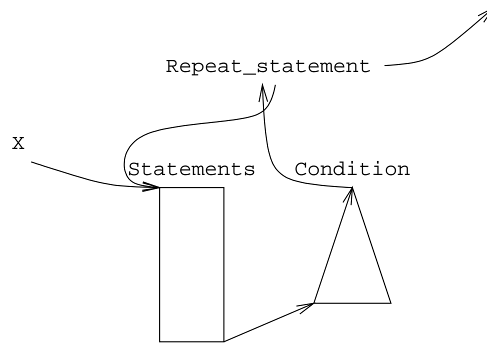


Figure Answers.25 Threaded AST of the repeat statement.

```
#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.

```
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.

```
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.

```
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.

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.

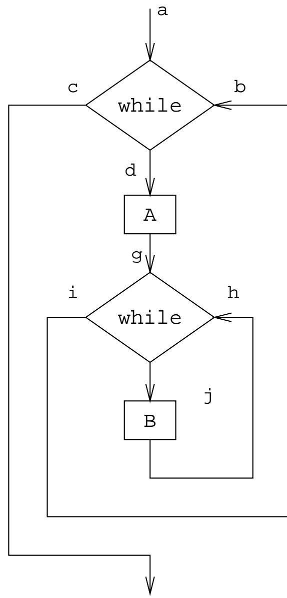


Figure Answers.31 Flow graph for static profiling of a nested while loop.

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.

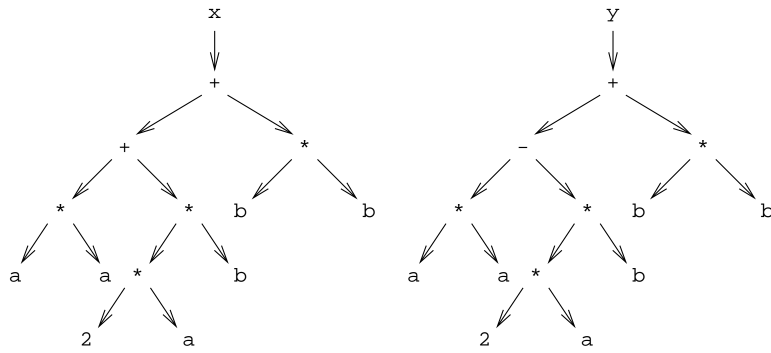


Figure Answers.33 The dependency graph before common subexpression elimination.

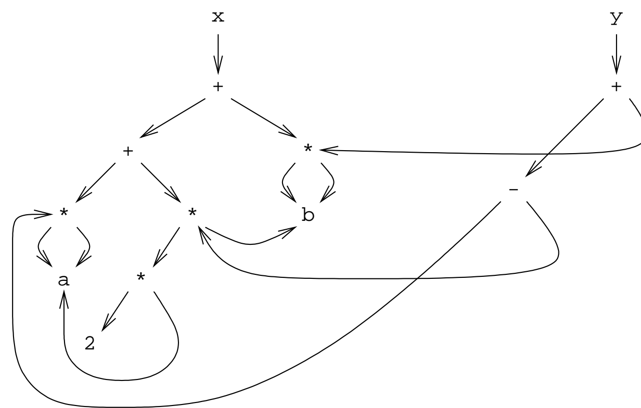


Figure Answers.34 The dependency graph after common subexpression elimination.

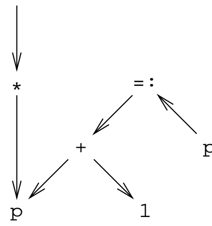


Figure Answers.35 The dependency graph of the expression `*p++`.

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.

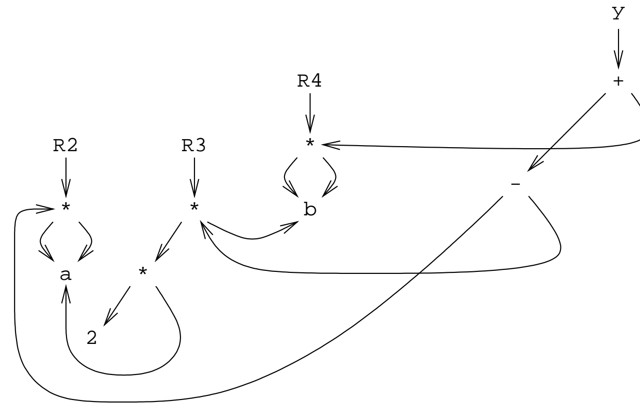


Figure Answers.37 The dependency graph of Figure Answers.34 with first ladder sequence removed.

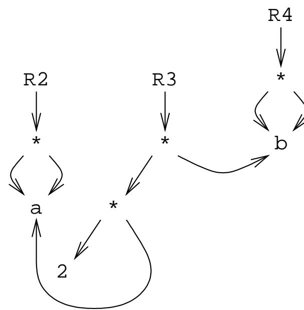


Figure Answers.38 The dependency graph of Figure Answers.37 with second ladder sequence removed.

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.

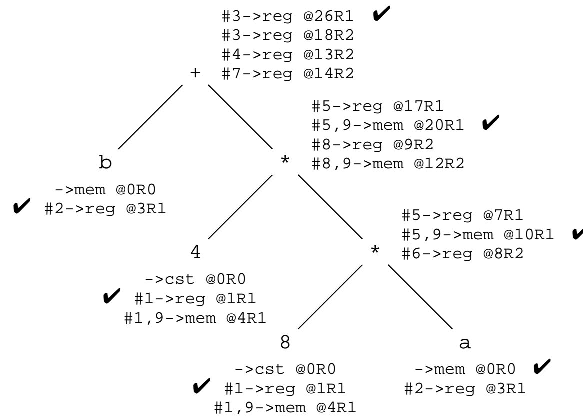


Figure Answers.41 Bottom-up pattern matching with costs and register usage.

```
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.

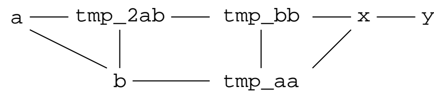


Figure Answers.43 The register interference graph for Exercise {#IntGr2}.

```
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.

```
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.

$$
\begin{aligned}
zeroth_offset(A) = -(&LB_1 \times LEN_PRODUCT_1 \\
&+LB_2 \times LEN_PRODUCT_2 \\
&\ldots \\
&+LB_n \times LEN_PRODUCT_n)
\end{aligned}
$$

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

$$base(A) + zeroth_offset(A)$$
$$+ i_1 \times LEN_PRODUCT_1 + ... + i_n \times LEN_PRODUCT_n$$

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

method table for `Rectangle`	method table for `Square`
`IsShape_Shape_Shape`	`IsShape_Shape_Shape`
`IsRectangle_Shape_Rectangle`	`IsRectangle_Shape_Rectangle`
`IsSquare_Shape_Shape`	`IsSquare_Shape_Square`
`SurfaceArea_Shape_Rectangle`	`SurfaceArea_Shape_Rectangle`

Figure Answers.48 Method tables for `Rectangle` and `Square`.

```
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 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.

```
    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.

```
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.

```
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`.

```
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`.

```
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`.