A.1 THE DEFINITION OF C--

This section defines the structure and meaning of a C-- program. It has five subsections, each dealing with one aspect of the language: basic symbols, programs, declarations, statements, and expressions. Basic symbols are the indivisible atoms of the language, and their meanings are defined by relating them to our shared experience with programming languages. All other constructs are composite—each is formed by combining parts. Their meanings can thus be defined in terms of the meanings of their components and fundamental concepts such as "sequence of execution."

A.1.1 Basic Symbols

The basic symbols of C-- are its identifiers, denotations, and delimiters. Figure A.1 shows the structure of identifiers and denotations in C--. The set *letter* includes all upper- and lower-case alphabetic characters, plus the underscore "_". Upper- and lower-case versions of the same letter are distinct. The delimiters of the language are the following special characters and keywords:

```
+          -          *          /
%          !          ?          :
=          ,          <          >
(          )          {          }
||         &&         ==         ;
break      continue   else       float
if         int        return     while
```

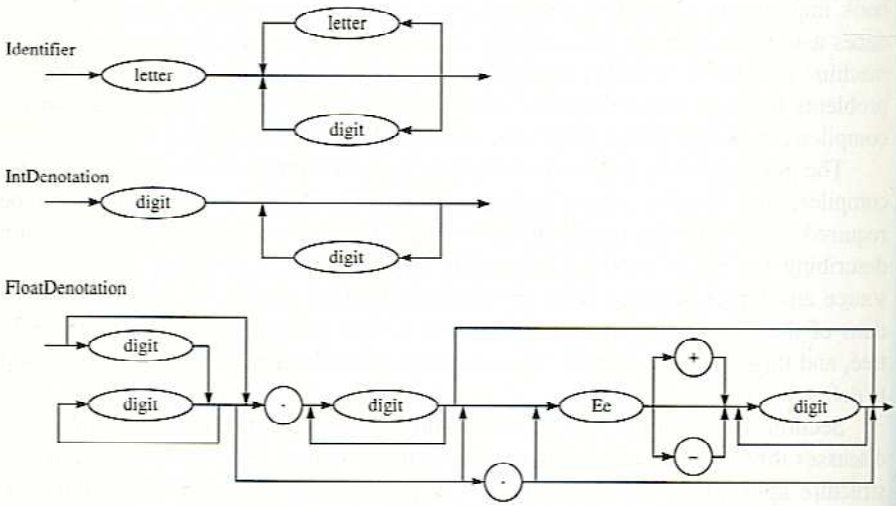

Figure A.1
C-- identifiers and denotations

An identifier is a freely chosen representation for an object. It is given meaning by a construct of the program. The appearances at which an identifier is given a meaning are called *defining occurrences* of that identifier. All other appearances of the identifier are called *applied occurrences*.

Denotations have the usual meanings.

Keywords can be used only as indicated by the syntax diagrams.

Comments are arbitrary sequences of characters beginning with "/*" and ending with "*/". Comments cannot be nested. Comments, spaces, and newlines may not appear within basic symbols. Two adjacent basic symbols must be separated by one or more comments, spaces, or newlines, unless one of the basic symbols is a special character. Otherwise comments, spaces, and newlines are meaningless.

A.1.2 Program Structure

Figure A.2 shows the structure of a C-- program. See Section A.1.3 for types, parameters, and declarations, and Section A.1.4 for statements.

A.1.2.1 Programs A program specifies a computation by describing a sequence of actions. A computation specified in C-- may be realized by any sequence of actions having the same effect as the one described here for the given computation. The meaning of constructs that do not satisfy the rules given here is undefined. Whether, and in what manner, a particular implementation of C-- gives meaning to undefined constructs is outside the scope of this definition.

A program is executed by reading parameter values from the standard input unit and executing the component *Compound*. A *Return* (Section A.1.4.5) must be executed to terminate the program.

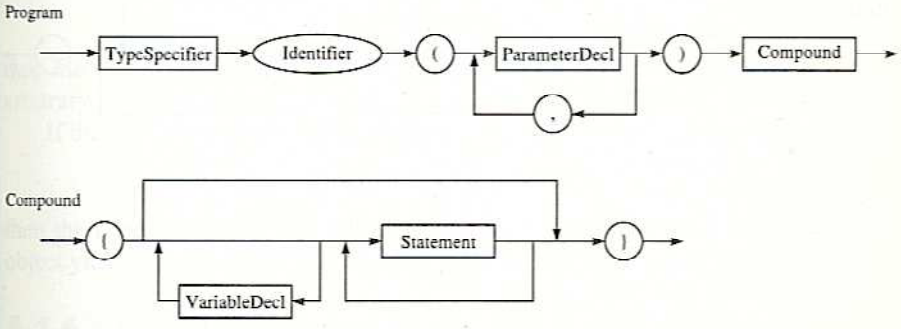

Figure A.2
The structure of a C-- program

A.1.2.2 Compounds The components of a *Compound* are executed in the sequence in which they were written.

A.1.2.3 Visibility Rules Let the term *range* be used to describe either a *Program* or a *Compound* (Figure A.2). The text of a range, excluding the text of ranges nested within it, may contain no more than one defining occurrence of a given identifier. Every applied occurrence of an identifier must *identify* some defining occurrence of that identifier. Unless otherwise stated, the defining occurrence D identified by an applied occurrence A of the identifier I is determined as follows:

1. Let R be the text of A, and let P be the entire C-- program.
2. Let R' be the smallest range properly containing R, and let T be the text of R', excluding the text of all ranges nested within it.
3. If T does not contain a defining occurrence of I, and R' is not P, then let R be R' and go to step (2).
4. If T contains a defining occurrence of I, then let T' be the fragment of T that begins with the first defining occurrence of I and ends at the end of T.
5. If T' does not contain R, and R' is not P, then let R be R' and go to step (2).
6. If T' contains R, then the defining occurrence at the beginning of T' is D.

(For an example of this process, see Section A.2.)

Identifier is a defining occurrence in the syntax diagrams for *Program* (Figure A.2), *ParameterDecl* (Figure A.3), and *VariableDecl* (Figure A.3). All other instances of *Identifier* are applied occurrences.

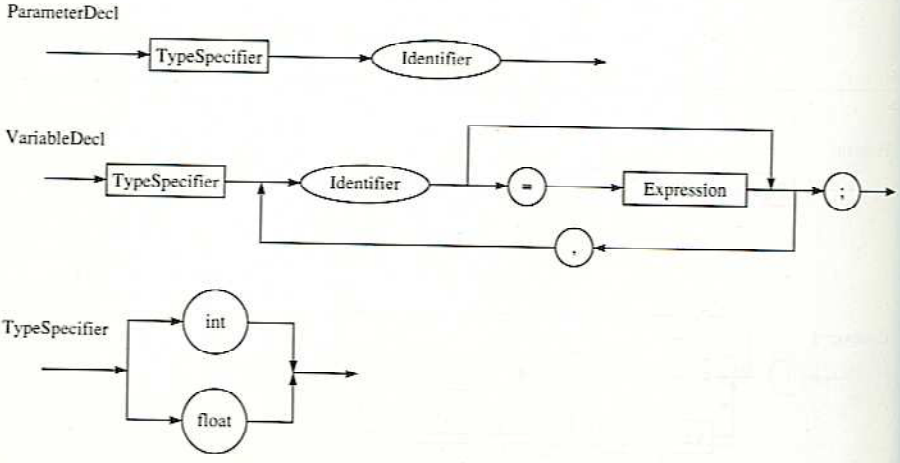

Figure A.3
Declarations

A.1.3 Declarations

Figure A.3 shows the structure of C-- declarations. See Section A.1.5 for Expressions.

A.1.3.1 Values, Types, Objects, and Variables *Values* are abstract entities upon which operations may be performed, *types* classify values according to the operations that may be performed upon them, and *objects* are the concrete instances of values that are operated upon. Two objects are *equal* if they are instances of the same value. A *variable* of type t is a concrete instance of an abstract entity that can *refer to* (or *contain*) an object that is an instance of a value of type t.

Every object and variable has a specified *extent*, during which it can be operated upon. The extents of denotations (see Section A.1.1) are unbounded; the extents of variables are determined by their declarations.

Values of types *int* and *float* have the usual meanings. The range of *int* values, and the range and precision of *float* values, are determined by the particular implementation of C–.

A.1.3.2 Parameter Declarations A variable is created, and the identifier represents this variable. The extent of the created variable is the entire execution history of the program. It refers initially to an object read from the standard input unit before program execution begins. The objects read are referred to by the variables named in the parameter list in order from left to right.

If the parameter declaration has the form

t Identifier

then the created variable can refer to objects of type t.

A.1.3.3 Variable Declarations A variable is created, and the identifier represents this variable. The extent of the created variable begins when the declaration is executed and ends when execution of the smallest *Compound* (Figure A.2) containing the declaration is complete.

If the variable declaration has the form

t Identifier

then the created variable can refer to objects of type t. Initially, it refers to an arbitrary object.

If the variable declaration has the form

t Identifier=*Expression*

then the created variable can refer to objects of type t. Initially, it refers to the object yielded by the given *Expression*.

A.1.4 Statements

Figure A.4 defines the structure of C-- Statements. See Section A.1.2 for Compounds and Section A.1.5 for Expressions.

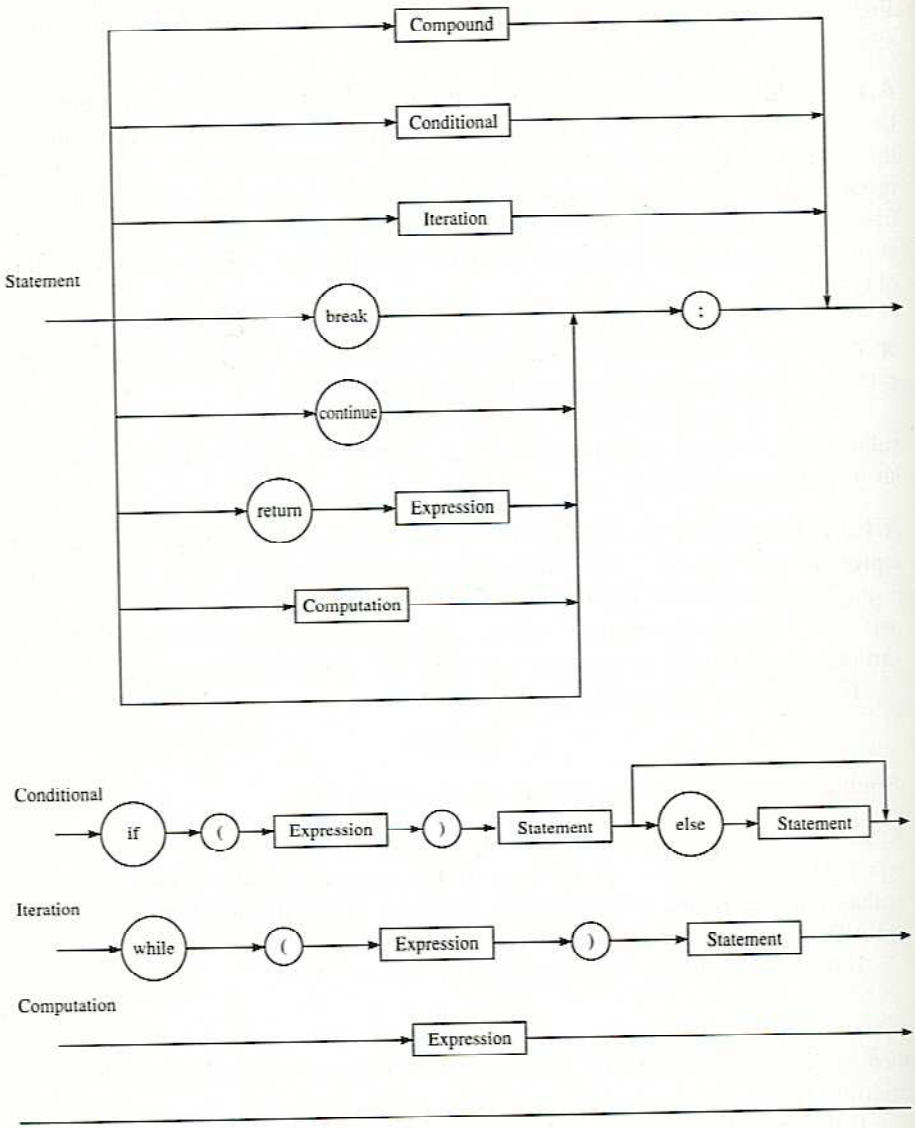

Figure A.4
Statements

A.1.4.1 Conditionals A *Conditional* is executed by first evaluating the component *Expression*, which must yield an object of type *int*. If the result of this evaluation is not 0, the first component *Statement* is executed. Otherwise, if the keyword *'else'* is present, then the second component *Statement* is executed.

If the keyword *'else'* is present, then it forms a *Conditional* with the closest preceding *'if'* in the same *Compound*. Therefore the statement

$$\texttt{if}\ (e_1)\ \texttt{if}\ (e_2)\ s_1\ \texttt{else}\ s_2$$

is identical in meaning to the statement

$$\texttt{if}\ (e_1)\ \{\ \texttt{if}\ (e_2)\ s_1\ \texttt{else}\ s_2\ \}$$

A.1.4.2 Iterations The iteration

$$\texttt{while}\ (\ e\ )\ s$$

is identical in meaning to the conditional

$$\texttt{if}\ (\ e\ )\ \{\ s\ \texttt{while}\ (\ e\ )\ s\ \}$$

A.1.4.3 Break A break may appear only within the component *Statement* of an iteration. Execution of a break causes termination of the smallest containing iteration.

A.1.4.4 Continue A continue may appear only within the component *Statement* of an iteration. Execution of a continue causes termination of the component *Statement* of the smallest containing iteration. The component *Expression* of the iteration is evaluated next. Therefore, the continue has the effect of cutting short the current execution of the controlled statement and testing whether another should be commenced. Contrast this behavior with that of the break, which escapes from the iteration entirely.

A.1.4.5 Return A return is executed by first evaluating the component *Expression*. Execution of the program is then terminated, with the value of the component *Expression* as the result.

A.1.4.6 Computation A computation is executed by first evaluating the component *Expression*. The result of that evaluation is then discarded.

A.1.5 Expressions

Expression structure in C-- is determined by operator precedence and association in the usual way (see Section 4.3):

Expression = *Identifier* '=' *Expression* | *Condition* .
Condition = *Disjunction* | *Disjunction* '?' *Expression* ':' *Condition* .
Disjunction = *Conjunction* | *Disjunction* '||' *Conjunction* .
Conjunction = *Comparison* | *Conjunction* '&&' *Comparison* .
Comparison = *Relation* | *Relation* '==' *Relation* .
Relation = *Sum* | *Sum* ('<' | '>') *Sum* .
Sum = *Term* | *Sum* ('+' | '−') *Term* .
Term = *Factor* | *Term* ('*' | '/' | '%') *Factor* .
Factor = *Primary* | ('!' | '−') *Factor* .
Primary = *IntDenotation* | *FloatDenotation* | *Identifier* | '(' *Expression* ')' .

Every subexpression (*Condition*, *Disjunction*, *Conjunction*, etc.) may be evaluated to yield an object of a certain type. The operands of an expression are evaluated collaterally unless the expression is a *Condition*, a *Disjunction*, or a *Conjunction* (see Section A.1.5.2). Each operator indication denotes a set of possible operations, with the particular one meant in a given context being determined by the operand types according to Table A.1. When the type of value delivered by an operand does not satisfy the requirements of an operation, a *coercion* (Section A.1.5.1) can be applied to yield a value that does satisfy the requirements. Any ambiguities in the process of selecting computations and coercions is resolved in favor of the choice with the smallest total number of coercions.

It must be possible to determine an operation for every operator indication appearing in a program.

A.1.5.1 Coercions The context in which an expression appears may permit a stated set of types for the result of that expression, prescribe a single type, or require that the result be discarded. When the a priori type of the result does not satisfy the requirements of the context, coercion is employed. (See Section 10.1.2 for further discussion.)

C-- allows only one coercion operation: Conversion of an integer value to floating point.

A.1.5.2 Operations An assignment causes the variable represented by the left operand to refer to a new instance of the value yielded by the right operand. The result of the assignment is the value yielded by the right operand.

If the left operand of an assignment represents an integer variable and the right operand yields a floating-point value, then the floating-point value is truncated to an integer by removing any nonzero fractional part. Thus, 3.6 would become 3 and -3.6 would become -3. If the resulting integer is not representable, then the behavior of the assignment is undefined.

A *Condition* containing the '?' and ':' operators is evaluated by first evaluating the component *Disjunction*. If the result is not zero, then the value of the condition is the value of the component *Expression*, and the component *Condition* is not evaluated. Otherwise, the value of the condition is the value of the component *Condition*, and the component *Expression* is not evaluated.

The component *Expression* and component *Condition* of a *Condition* must be *balanced* to ensure that the type of the result yielded is the same, no matter which alternative was chosen. Balancing involves coercing the result of each to a common type. When the type is uniquely prescribed by the context, then this type is chosen as the common result type for all alternatives. If the context of the expression is such that several result types are possible, the one leading to the smallest total number of coercions is chosen.

The expression $e_1 \parallel e_2$ has the same meaning as the expression

$$e_1 == 0 \,?\, e_2 : 1.$$

The expression e_1 && e_2 has the same meaning as the expression

$$e_1 == 0 \,?\, 0 : e_2.$$

TABLE A.1 OPERATOR IDENTIFICATION

Indication	Operand Type Left	Operand Type Right	Result Type	Operation
=	*int*	*int*	*int*	integer assignment
		float		truncating assignment
	float	*float*	*float*	floating assignment
||	*int*	*int*	*int*	disjunction
&&				conjunction
==				integer equality
	float	*float*		floating equality
<	*int*	*int*		integer less than
	float	*float*		floating less than
>	*int*	*int*		integer greater than
	float	*float*		floating greater than
+	*int*	*int*	*int*	integer addition
	float	*float*	*float*	floating addition
−	*int*	*int*	*int*	integer subtraction
	float	*float*	*float*	floating subtraction
*	*int*	*int*	*int*	integer multiplication
	float	*float*	*float*	floating multiplication
/	*int*	*int*	*int*	integer division
	float	*float*	*float*	floating division
%	*int*	*int*	*int*	remainder
!				complement
−				integer negation
		float	*float*	floating negation

The expression $!e$ has the same meaning as the expression $e == 0\ ?\ 1 : 0$.

Equality yields the value 1 if its operands have the same value. Otherwise, it yields the value 0.

Relational operators yield the value 1 if the relation they describe is satisfied. Otherwise, they yield the value 0.

The arithmetic operators addition, subtraction, multiplication, floating division, and negation have the usual meaning as long as the values of all operands

and results lie in the range permitted by the mapping from C-- objects to target machine objects (Section 8.1.1). Division and remainder are defined only when the value of the right operand is nonzero.

The result of an integer division operation with dividend m and divisor $n \neq 0$ is determined as follows:

1. Let q and $0 \leqslant r < |n|$ be two integers such that $m = q \times n + r$.
2. If $r = 0$ then the result of m/n is q.
3. Otherwise, if $m > 0$ and $n > 0$ then the result of m/n is q.
4. Otherwise, the result of m/n is either q or $q+1$ at the discretion of the implementor.

The result of the remainder operation $m\%n$ is given by:

$$m - (m/n)*n$$

A.2 SAMPLE C-- PROGRAMS

Figure A.5 shows a C-- program to compute the greatest common divisor of two integers. The initial values of the parameters are read from the standard input device as a part of the initialization process. The result returned by the program is written to the standard output device as a part of the finalization process.

The factorial program of Figure A.6 uses an initialized variable declaration to define the limit imposed by machine arithmetic. If the initial value of v is invalid, the program returns -1 as its answer.

Figure A.7 illustrates the visibility rules of C--. According to Section A.1.2.3, the identification of the identifier v in $v+7$ is carried out as follows:

1. Let R be the identifier v in $v+7$, and let P be the complete text of Figure A.7.
2. Let R' be the compound statement beginning with {, and ending with }; this is the smallest range containing R. Since there are no ranges contained within R', $T = R'$.

```
int
GCD(int x, int y)
{
   while (!(x == y))
      if (x > y) x = x-y;
      else y = y-x;
   return x;
}
```

Figure A.5
GCD in C--

```
int
Factorial(int v)
{
    int limit = 7;

    if (v < 0 || v > limit) return -1;
    {
        int c = 0, fact = 1;

        /* Loop invariant: fact == c! */
        while (c < v) {
            c = c+1; fact = fact*c;
        }
        return fact;
    }
}
```

Figure A.6
Factorial in C--

3. T contains a defining occurrence of v (the variable declarator $v=3$), so we do not return to step (2).
4. Since $v=3$ is the first defining occurrence of v in T, let T' be the fragment of T beginning with $v=3$ and ending with }.
5. T' does not contain R, R' is not P, so we let R be R' and return to step (2).

2′. Since R is now the entire compound statement, the smallest range containing R is P, the whole text of Figure A.7. Therefore $R'=P$. Let T be the lines up to (but not including) {, (because T must exclude the text of the compound statement—a range nested within R').

3′. T contains a defining occurrence of v (the declaration *int v*), so we do not return to step (2).

4′. Since *int v* is the first defining occurrence of v in T, let T' be the fragment of T beginning with *int v* and ending just after }.

5′. T' contains R, so we do not return to step (2).

```
int
Scope(int v)
{
    int limit = v + 7, v = 3;

    return limit-v;
}
```

Figure A.7
Visibility in C--

6. T' contains R, so D is the defining occurrence at the beginning of T'—the v in the parameter declaration.

A similar analysis can be used to show that the v in the return statement identifies the defining occurrence in the variable declaration $v=3$. Thus, the effect of this program is to print an integer larger by 4 than its input parameter.

It is important to note that the visibility rules of C-- make use of an identifier before its definition impossible.