UNPL 1376

Bell Laboratories **Cover Sheet for Technical Memorandum**

Title- **Semantics of the C programming language, part 1: statements**

Date- February 2, 1979

TM- 79-1271-04

Other Keywords-

Author	Location	Extension	Charging Case- 39199
Ravi Sethi	MH 2C-519	4006	Filing Case- 39199-11

ABSTRACT

This is one of a sequence of papers defining the semantics of the C programming language. After providing a brief introduction to the semantic method, the semantics of statements are given in section 3. Section 3 was prepared by adding denotational semantics for the various statements to the discussion in section 9 of the C Reference Manual, as it appears in: B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978. The work reported here was carried out with the cooperation of F. T. Grampp and A. R. Koenig.

Pages Text	16	Other	10	Total	26
No. Figures	4	No. Tables	0	No. Refs.	9

Carole Scheiderman
MH 2F-128

BELL TELEPHONE LABORATORIES, INC.

TM-79-1271-4

DISTRIBUTION
(REFER GEI 13.9-3)

COMPLETE MEMORANDUM TO

CORRESPONDENCE FILES

OFFICIAL FILE COPY
PLUS ONE COPY FOR
EACH ADDITIONAL FILING
CASE REFERENCED

DATE FILE COPY
(FORM E-1328)

10 REFERENCE COPIES

<AHO,ALFRED V
<BALENSON,CHRISTINE M
<BARON,ROBERT V
<BECKER,RICHARD A
BROWN,W STANLEY
<CHEN,STEPHEN
<CHERRY,LORINDA L
<FELDMAN,STUART I
<FRASER,A G
<GOLDSTEIN,A JAY
<GRAHAM,R L
+HANNAY,N B
<JOHNSON,STEPHEN C
<KEESE,W M
<KERNIGHAN,BRIAN W
<LUDERER,GOTTFRIED W R
<MARANZANO,J F
<MARKY,GERALDINE A
+MC DONALD,H S
<MC GILL,R
MCILROY,M DOUGLAS
<MENNINGER,R E
<MORGAN,SAMUEL P
+PRIM,R C
<RALEIGH,T M
<RIDDLE,GUY G
<SCHLEGEL,C T
<SETHI,RAVI
<STORER,JAMES A
<SZYMANSKI,THOMAS G
TERRY,M E
<WEINBERGER,PETER J
<YAMIN,ELAINE E
<YANNAKAKIS,MIHALIS
34 NAMES

COVER SHEET ONLY TO

CORRESPONDENCE FILES

4 COPIES PLUS ONE
COPY FOR EACH FILING
CASE

AAGESEN,JOHN
ACKERMAN,A FRANK
ACKERMAN,J T
AHRENS,RAINER B

COVER SHEET ONLY TO

ALCALAY,D
AMRON,IRVING
ANDERSON,FREDERICK L
ANDERSON,KATHRYN J
ANDERSON,MILTON M
APPELBAUM,MATTHEW A
ARMSTRONG,D B
ARNOLD,GEORGE W
ARNOLD,PHYLLIS A
ARNOLD,S L
ARNOLD,THOMAS F
ASTHANA,ABHAYA
ATAL,BISHNU S
BAILY,DAVID Z
BARCLAY,DAVID K
BAROFSKY,ALLEN
BASEIL,RICHARD J
BAUER,ANDREW E
BAUER,BARBARA T
BAUER,HELEN A
<BAUGH,C R
BEDNAR,JOSEPH A,JR
BENCO,DAVID S
BENISCH,JEAN
BENNETT,RAYMOND W
BERGH,A A
BERGLAND,G D
BERNSTEIN,DANIELLE R
BERNSTEIN,L
BEYER,JEAN-DAVID
BIANCHI,M H
BICKFORD,NEIL B
BILLINGTON,MARJORIE J
BILOWOS,R M
BIREN,IRMA B
BISHOP,VERONICA L
BLAZIER,S D
BLEECKER,SAMUEL E
BLINN,J C
BLOSSER,PATRICK A
BLUE,JAMES L
BLUMER,THOMAS P
BLUM,MARION
BOCKUS,ROBERT J
BODEN,F J
BOEHM,KIM B
BOIVIE,RICHARD H
BONANNI,L E
BORDELON,EUGENE F
BORISON,ELLEN A
BOURNE,STEPHEN R
BOWERMAN,REBECCA ELAINE
BOWYER,L RAY
<BOYCE,W M
BOYER,PHYLLIS J
BOYLE,W S
BRADLEY,R H
BRAM,ALAN
<BRANDT,RICHARD B
BRAUNE,DAVID P
BRIGGS,GLORIA A
BROSS,JEFFREY D

COVER SHEET ONLY TO

BROVMAN,INNA
BROWN,ELLINGTON L
BROWN,LAURENCE MC FEE
BROWN,W R
BULLEY,R M
BURG,F M
BURNETTE,W A
<BURNETT,DAVID S
BUROFF,STEVEN J
BURROWS,THOMAS A
BUTLETT,DARRELL L
BYRNE,EDWARD R
CAMPBELL,JERRY H
<CANADAY,RUDD H
CANNON,LAYNE W
CARTER,DONALD H
CASPERS,BARBARA E
CAVINESS,JOHN D
CERMAK,I A
CHAI,D T
CHAMBERS,B C
<CHAMBERS,J M
CHAYUT,IRA G
CHENG,Y
CHEN,E
CHESSON,GREGORY L
CHRISTENSEN,S W
CHRIST,C W,JR
CHUNG,MICHAEL
CLARK,DAVID L
CLAYTON,D P
CLINE,LAUREL M I
COBEN,ROBERT M
COCHRAN,ANITA J
<COLE,LOUIS M
COLE,MARILYN O
COLLICOTT,R B
CONDON,J H
CONKLIN,DANIEL L
CONNERS,RONALD R
<COOPER,ARTHUR E
COOPER,MICHAEL H
COSTELLO,PETER E
COTTRELL,JENNIE L
CRAGUN,DONALD W
CRISTOFOR,EUGENE
<CRUME,L L
CRUPI,JOSEPH A
CUTLER,C CHAPIN
DAVIS,R DREW
DE FAZIO,M J
DE GRAAF,D A
DE TREVILLE,JOHN D
DEAN,JEFFREY S
DENNY,MICHAEL S
DIB,GILBERT
DICKMAN,BERNARD N
<DIMMICK,JAMES O
DINEEN,THOMAS J
DITZEL,DAVID R
DOLOTTA,T A
DONNELLY,MARGARET M

COVER SHEET ONLY TO

DOWDEN,DOUGLAS C
DRAKE,LILLIAN
D'ANDREA,LOUISE A
DUCHARME,ROBERT LAWRENCE
DUFFY,F P
DUGGER,DONALD D
DWYER,T J
DYER,MARY E
EIGEN,D J
EISEN,STEVEN R
<EITELBACH,DAVID L
ELDREDGE,BARBARA D
ELLIOTT,RUBY J
ELLIS,DAVID J
ELY,T C
EPLEY,ROBERT V
ESCOLAR,CARLOS
ESSERMAN,ALAN R
FABISCH,M P
FABRICIUS,WAYNE N
FAIRCHILD,DAVID L
FEDERICO,FRANK W
FEDER,J
FERIDUN,K K
FINUCANE,J J
FISCHER,HERBERT B
FISCHER,MICHAEL T
FISHER,EDWARD E
FISHMAN,DANIEL E
FLANDRENA,R
FLEISCHER,RAYMOND C
FONG,K T
FORTNEY,V J
FOUNTOUKIDIS,A
FOWLER,GLENN S
FOWLER,H EUGENE
FOWLKES,EDWARD B
FOX,PHYLLIS A
FOY,J C
FRANK,AMALIE J
FRANK,RUDOLPH J
FREEMAN,K G
FREEMAN,MARTIN
FROST,H BONNELL
FRUCHTMAN,BARRY
FRYDMAN,URSZULA D
GABBE,JOHN D
GADENZ,RENATO N
GALLANT,R J
GANA,JORGE L
GARST,BLAINE,JR
<GATES,G W
GAY,FRANCIS A
GEARY,M J
GEESS,T J,JR
GEORGEN,MICHAEL R
GEPNER,JAMES R
GEYLING,F T
GIBB,KENNETH R
<GILLETTE,DEAN
GIMPEL,J F
<GITHENS,J A

COVER SHEET ONLY TO

GITHENS,JAY L
<GLASSER,ALAN L
GLUCK,F G
<GNANADESIKAN,R
GOGUEN,N H
GOLABEK,RUTH T
GOLDBERG,HAROLD JEFFREY
GORDON,MOSHE B
<GORMAN,J E
GORTON,D R
GRAYSON,C F,JR
GREENBAUM,HOWARD J
GREENLAW,A L
GROSS,ARTHUR G
GRUENWALD,JOHN
GRZELAKOWSKI,MAUREEN E
GUIDI,PIER V
GUNTHER,F L
GUSTAVSON,J R
GUTTMAN,NEWMAN
HAFER,E H
HAGGERTY,JOSEPH P
HAIGHT,R C
HAISCH,H F,JR
HALEY,CHARLES B
HALE,A L
HALLIN,THOMAS G
HALL,ANDREW D,JR
HALL,MILTON S,JR
HALL,WILLIAM G
HAMILTON,LINDA L
<HAMILTON,PATRICIA A
<HARKNESS,CAROL J
HARTMAN,DON W
HARUTA,K
HAUSE,A DICKSON
<HAWKINS,DONALD T
<HAYDEN,DONALD F,JR
HERGENHAN,C B
<HESSELGRAVE,MARY E
HOCHBERG,GLENN A
HOEHN,MARIE J
HOLTMAN,JAMES P
HORWEDEL,J H
HOUGHTON,THOMAS F
HOWARD,PHYLLIS A
HOYT,WILLIAM F
HO,DON T
HO,JENNY
HO,TIEN-LIN
HSU,TAU
HUBER,RICHARD V
HUNNICUTT,C F
IPPOLITI,O D
IRVINE,M M
ISMAN,MARSHALL A
<IVIE,EVAN L
JACKOWSKI,D J
JACKSON,F M,2ND
JACOBS,H S
JAMES,J W
<JENSEN,PAUL D
...
565 TOTAL

+ NAMED BY AUTHOR   > CITED AS REFERENCE   < REQUESTED BY READER   (NAMES WITHOUT PREFIX WERE SELECTED USING THE AUTHOR'S SUBJECT OR ORGANIZATIONAL SPECIFICATION AS GIVEN BELOW)

MERCURY SPECIFICATION..

COMPLETE MEMO TO:
127-SUP

COVER SHEET TO:
12-DIR   13-DIR   127

COPLGP = COMPUTING/PROGRAMMING LANGUAGES/GENERAL PURPOSE

SETHI,RAVI
MH 2C519

TM-79-1271-4
TOTAL PAGES   26

TO GET A COMPLETE COPY:

1. BE SURE YOUR CORRECT ADDRESS IS GIVEN ON THE OTHER SIDE.
2. FOLD THIS SHEET IN HALF WITH THIS SIDE OUT AND STAPLE.
3. CIRCLE THE ADDRESS AT RIGHT. USE NO ENVELOPE.

PLEASE SEND A COMPLETE COPY TO THE ADDRESS SHOWN ON THE OTHER SIDE
NO ENVELOPE WILL BE NEEDED IF YOU SIMPLY STAPLE THIS COVER SHEET TO THE COMPLETE COPY.
IF COPIES ARE NO LONGER AVAILABLE PLEASE FORWARD THIS REQUEST TO THE CORRESPONDENCE FILES.

Bell Laboratories

Subject: Semantics of the C programming language, part 1: statements
Case- 39199 -- File- 39199-11

date: February 2, 1979

from: Ravi Sethi

TM: 79-1271-04

MEMORANDUM FOR FILE

1. Introduction

1.1. *Background.* The purpose of this paper is to define the semantics of statements in the C programming language. In addition to serving as a reference manual, the semantic specification will be precise enough that a program like SIS by Mosses [mss76] can automatically construct an interpreter from the specification.

C is a general purpose programming language that is available in a number of environments. Most of the software running under the UNIX* operating system is written in C. Notes on the development of C and an assessment of the language may be found in [rit78b]. Our starting point for language details was the book on C by B. W. Kernighan and D. M. Ritchie [ker78] and the compilers by S. C. Johnson [joh78] and D. M. Ritchie [rit78a].

It is suggested that the reader proceed sequentially through this paper since the semantics of the various kinds of statements are inter-related.

The syntax of statements is shown in Figure 1.

1.2. *Notation.* In addition to giving some idea of the meaning of expressions, this subsection introduces notation that will be useful in the sequel. Much of the notation will be introduced by example. For a more detailed presentation see [set79a] where the semantic method and the notations we will use are discussed. References to the literature may also be found in [set79a]. The semantics of statements fall into the class of denotational specifications of programming languages [scs71,sto77,ten76].

The value of an identifier will be given in two stages, as shown in Figure 2. Given an environment e and state s, the value of identifier id is $s(e(id))$. There are a number of reasons for a two stage mapping: one of them is that a two stage mapping makes it easy to give meaning for identifiers that are type names or statement labels.

Expressions in C may have side effects since assignments may be embedded within expressions. Given an environment e and a state s as in Figure 2, an expression yields a value v and a modified state s'. The meaning of an expression will therefore be a function that takes e and s as parameters and returns a pair (v,s').

The nonterminal *exp* occurs in several places in Figure 1. The meaning of a program fragment generated by *exp* will be denoted by $[\![exp]\!]$. We therefore have

$$(v,s') = [\![exp]\!](e,s)$$

*UNIX is a trademark of Bell Laboratories.

```
stm:
    return ;                              §3.1
    return exp ;                          §3.1
    ;                                     §3.2
    exp ;                                 §3.3
    if ( exp ) stm                        §3.4
    if ( exp ) stm else stm               §3.4
    break ;                               §3.5
    continue ;                            §3.6
    do stm while ( exp ) ;                §3.7
    while ( exp ) stm                     §3.8
    for ( exp ; exp ; exp ) stm           §3.9
    switch ( exp ) stm                    §3.10
    case constant_exp : stm               §3.10
    default : stm                         §3.10
    { stm_s }                             §3.11
    goto identifier ;                     §3.12
    identifier : stm                      §3.12

stm_s:
    /* empty */                           §3.11
    stm_s stm                             §3.11
```

Figure 1. The above syntax for statements is a restriction of that in the C Reference Manual [ker78]. In particular, declarations are not permitted within compound statements i.e. "blocks" are not included. Since a `goto` may jump anywhere within a function, it is convenient to preprocess declarations away. Section numbers on the right refer to the place where the meaning of a construct is discussed.

$$\begin{tikzcd}\text{identifier} \arrow[r, bend left, "e"] & \text{location} \arrow[r, bend left, "s"] & \text{value}\end{tikzcd}$$

Figure 2. An *environment e* maps an identifier to a location, and then a *state s* maps a location to a *value*. Another use of environments occurs in §3.5, 3.6, and 3.12 where the meaning of `break`, `continue`, and `goto` is given.

Let V, E, and S be the domains of values, environments, and states, respectively. The special symbol $\perp$ (read "bottom") will denote the "undefined" value in V.

We can declare that $[\![exp]\!](e,s) = (v,s')$ has its first component in the domain V and its second component in domain S by writing

$$[\![exp]\!](e,s):\ (\mathrm{V},\mathrm{S})$$

Furthermore, we can declare that $[\![exp]\!]$ taken by itself is a function that maps an element of (E,S) to an element of (V,S) by writing

$$[\![exp]\!]:\ (\mathrm{E},\mathrm{S}) \rightarrow (\mathrm{V},\mathrm{S})$$

Suppose that starting with state s we assign a value v to a location l and obtain the modified state s'. The relation between s, v, l, and s' can be expressed using either of the following two equivalent forms:

$$s' = s[v/l]$$

$$s' = \lambda m.\ (\ m{=}l \rightarrow v,\ s(m)\ )$$

The latter form reads, "Given parameter m, if m equals l then $s'(m)$ is v, otherwise $s'(m)$ is $s(m)$.

Conditional expressions will be written using the syntax $a \rightarrow b, c$ rather than the C syntax *exp* ? *exp* : *exp* in order to keep the metalanguage for defining semantics quite distinct from the syntax of C.

2. The Semantic Method

The approach to defining the semantics of statements will be illustrated by examining the following program fragment. The approach is essentially that of §6 in [set79a], but will take into account side effects of procedures.

```
int next = 0;
square() {
    int sq;
    sq = next * next;
    next++;
    return (sq);
    sq = 0;    /* can never be reached */
    }
```

Procedures in C are called "functions" since they return values and may be called within expressions. When it is necessary to distinguish between a program fragment and a mathematical function, we will refer to the program fragment as a "C-function". `square` in the program fragment above is therefore a C-function.

Statements in C must occur within the body of a C-function. A `goto` may jump anywhere within a function, but it is not possible to jump out of the function. Consequently, a natural unit for specifying semantics is the sequence of statements that constitute the body of a function definition.

Successive calls to the parameterless function `square` will return the sequence `0, 1, 4, 9, ...`. In addition to returning the value of `sq`, the function `square` increments the value of `next` which is external to `square`. Like expressions, C-functions return values and change the state, so we will view C-functions as returning elements of the domain (V,S).

A portion of the parse tree for the above program fragment is given in Figure 3. The statements of `square` are in the subtree at node x in Figure 3. From this subtree we will construct a function f to determine the pair (v,s') returned by `square`. When f is applied to the state s with which the statements in the subtree are reached, we will get

$$f(s) = (v,s')$$

A function like f which maps a state to the "answer" of a C-function will be called a (*statement*) *continuation*. With f as a typical example, it is clear that the domain C of continuations must be

$$\mathrm{C} = \mathrm{S} \rightarrow (\mathrm{V},\mathrm{S})$$

A C-function returns to its caller either on executing a `return` statement, or on reaching the end of the statements in the function. In the latter case, the value returned is *grb*, which is a special "garbage" value.

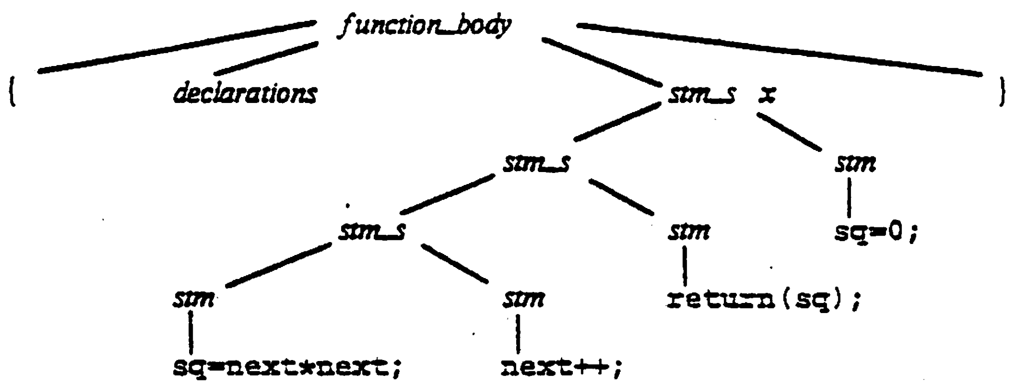

Figure 3. A sketch of a parse tree for a program fragment.

The continuation f for the subtree at node x in Figure 3 will be determined during a post-order traversal of the subtree, as shown in Figure 4. We start the traversal with a starting continuation c_0 which corresponds to a C-function containing just the null statement. Clearly, for all states s,

$$c_0(s) = (grb,s)$$

After the assignment `sq=0;` is traversed, we get continuation c_1. On state s, we want $c_1(s)$ to be determined as follows: the state s is updated to s' by entering 0 for the location of `sq`, and then $c_0(s') = (grb,s')$ is returned. In other words, the effect of the assignment `sq=0;` has to be accumulated by changing the state, and then the continuation for the rest of the statements has to be used.

The interesting case is `return(sq);` which is the next statement to be traversed. When a `return` is executed, control returns to the caller of `square`, regardless of the statements that follow the `return`. Continuation c_2 will therefore be such that for all states s,

$$c_2(s) = (v,s) \quad \textit{where} \quad v = s(e(\texttt{sq}))$$

Note that c_2 does not depend in any way on c_1. Thus the assignment `sq=0;`, which can never be reached, cannot affect the continuation f for the whole tree in Figure 4.

Let x be any of the statements in Figure 4. Suppose we know the continuation c_x for the statements to the right of x, the environment e_x, and the state s_x that x is reached with. With e_x, c_x, and s_x, we can predict the "answer" of the C-function. (Note that the assignment `sq=0;` can never be reached, so the fact that we may possibly predict an incorrect "answer" at this statement does not matter.)

The meaning, as outlined above, of the program fragment generated by *stm* will be denoted by **sc**$[\![stm]\!]$, and can be declared by writing

$$\mathbf{sc}[\![stm]\!](e,c,s)\text{: } (\mathrm{V},\mathrm{S})$$

In **sc**, the s is from statement, and the c is from continuation. We will refer to **sc** as a *valuation* and treat it as a function from *stm* to its meaning **sc**$[\![stm]\!]$. The reason for writing **sc**$[\![stm]\!]$ instead of just $[\![stm]\!]$ is to avoid confusion with other valuations in §4-5 where the meaning of `goto` and `switch` statements will be discussed in more detail.

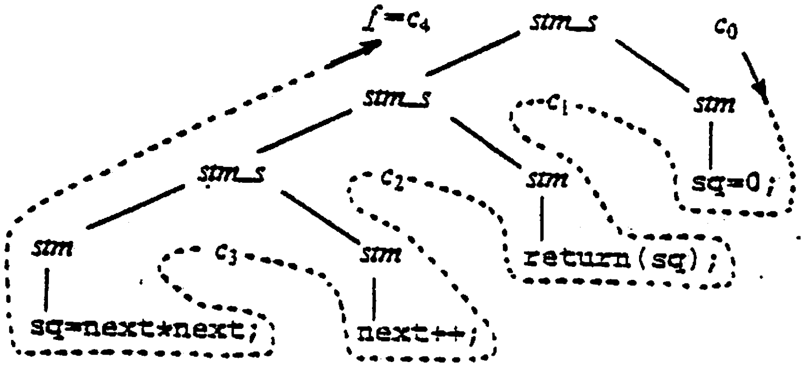

Figure 4. The meaning of a statement will be given using continuations. The continuation for the entire tree will be determined during a post-order traversal of the tree.

3. Semantics of Statements

The semantics of each kind of statement will be given by first discussing the statement and then presenting precise definitions. This section was prepared by extensively editing §9 of the C Reference Manual in [ker78].

As mentioned at the end of §2, the meaning of statements will be such that

$$\mathbf{sc}[\![stm]\!](e,c,s)\colon\ (\mathrm{V},\mathrm{S})$$

Appendix A contains a concise listing of the meaning of statements under valuation **sc**.

Since we will be considering one kind of statement at a time, a minor extension of our notation will be useful. Rather than writing something like

if stm is `return ;` *then* $\mathbf{sc}[\![stm]\!](e,c,s) = (grb,s)$

we will write

$\mathbf{sc}[\![$`return ;`$]\!](e,c,s) = (grb,s)$

3.1. `return`.

A function returns to its caller by means of the `return` statement, which has one of the forms

```
return ;
return exp ;
```

In the first case, the special "garbage" value *grb* and the current state are returned. In the second case, the value of the expression and the state after expression evaluation are returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of the function in which it appears. This type conversion will become explicit when the semantics of function definitions are given.

More precisely,

$\mathbf{sc}[\![$`return ;`$]\!](e,c,s) = (grb,s)$

Note that the continuation *c* is ignored, thereby indicating that control returns to the caller of the function. Finally,

$\mathbf{sc}[\![$`return` *exp* `;`$]\!](e,c,s) = [\![exp]\!](e,s)$

Recall from §1.2 that $[\![exp]\!](e,s)$ will be a pair (v,s') where v is a value and s' is a modified state.

3.2. *Null.* The null statement has the form

```
;
```

$$\mathbf{sc}[\![;]\!](e,c,s) = c(s)$$

The null statement does not change the state and control passes to the following statement. The continuation c for the statements following the null statement is therefore applied to the state s with which the null statement is reached.

3.3. *Expression.* An expression followed by a semicolon is a statement. Such statements are usually assignments or function calls.

$$\mathbf{sc}[\![exp\ ;]\!](e,c,s) = c(s') \quad \textit{where} \quad (v,s') = [\![exp]\!](e,s)$$

Evaluation of the expression yields a state s', and the continuation for the statements following the expression statement is then applied to s' to predict the "answer" of the function. The value v of the expression is thrown away.

3.4. *Conditional.* The two forms of the conditional statement are

```
if ( exp ) stm
if ( exp ) stm else stm
```

In both cases the expression is evaluated and if it is non-zero, the first substatement is executed. In the latter case the second substatement is executed if the expression is 0. As usual the "else" ambiguity is resolved during syntax analysis by connecting an `else` as follows: given a choice between `if`'s to connect the `else` to, the nearest `if` is chosen.

We will give the semantic equation only for conditionals with `else` clauses since the other form is equivalent to a conditional with a null statement, §3.2, following the `else`.

```
if ( exp ) stm else ;
```

The semantic equation for the `if` statement is

$$\begin{aligned}&\mathbf{sc}[\![\texttt{if}\ (\ exp\ )\ stm_1\ \texttt{else}\ stm_2]\!](e,c,s) = \\ &\quad \textit{let}\ (v,s') = [\![exp]\!](e,s) \\ &\quad \textit{in}\ \ v \neq 0 \rightarrow \mathbf{sc}[\![stm_1]\!](e,c,s')\ ,\ \mathbf{sc}[\![stm_2]\!](e,c,s')\end{aligned}$$

3.5. `break`. The `break` statement has the form

```
break ;
```

This statement causes termination of the smallest enclosing `do`, `while`, `for`, or `switch` statement, §3.7-3.10; control passes to the statement following the terminated statement. In each of the statements

```
while (...) {       do {                for (...) {        switch (...) {
  ...                 ...                 ...                ...
}                   } while (...);      }                  }
breakout: ;         breakout: ;         breakout: ;        breakout: ;
```

a `break` is equivalent to `goto breakout`. (Following the `breakout:` is a null statement.

§3.2.)

The continuation c for the statements following the `break` must be ignored since control passes to the statement x following the `do`, `while`, `for`, or `switch` enclosing the `break`. We therefore need the continuation c' for x and the statements that follow x.

Sitting at a parse tree node corresponding to a `break` statement, we want to be able to use the continuation c' without having to leave the node to find c'. The continuation c' will therefore have to be "passed" in some way. The environment e and a special identifier `brk`, which is invisible to the user, will be used for this purpose. The smallest enclosing `do`, `while`, `for`, or `switch` will already have set up the environment e so that $c' = e(\texttt{brk})$. Thus all that needs to be done here is to look up the continuation c' in the environment and apply it to the state s.

$$\textbf{sc}[\![\texttt{break ;}]\!](e,c,s) = c'(s) \quad \textit{where} \quad c' = e(\texttt{brk})$$

The purpose of an environment is to map an identifier to the semantic object denoted by the identifier. Some identifiers denote locations: others, like statement labels, and the special identifier `brk` denote continuations.

3.6. `continue`. The `continue` statement has the form

```
continue ;
```

This statement causes control to pass to the loop-continuation portion of the smallest enclosing `do`, `while`, or `for` statement, §3.7-3.9; that is to the end of the loop. In each of the statements

```
while (...) {        do {                 for (...) {
  ...                  ...                  ...
contin: ;            contin: ;            contin: ;
}                    } while (...);       }
```

a `continue` is equivalent to `goto contin`. (Following the `contin:` is a null statement, §3.2.)

As with the `break` statement, §3.5, the continuation c' to be used will have already been entered in the environment at the special identifier `con` by the `do`, `while`, or `for` statement.

$$\textbf{sc}[\![\texttt{continue ;}]\!](e,c,s) = c'(s) \quad \textit{where} \quad c' = e(\texttt{con})$$

3.7. `do`. The `do` statement has the form

do *stm* while (*exp*) ;

The substatement *stm* is executed repeatedly until the value of the expression becomes zero. The test takes place after each execution of the statement.

The intent of the following two statements is similar, but an exact analogy is frustrated by `break` and `continue` statements, §3.5-3.6, and the fact that *stm* may contain labeled statements, §3.12.

do *stm* while (*exp*) ;
stm while (*exp*) *stm*

We will give the meaning of the `do` statement in terms of the `while` statement, §3.8. Appropriate environments will be set up so that `break` and `continue` statements within *stm* are handled properly.

$\mathbf{sc}[\![$`do` *stm* `while ( ` *exp* ` ) ;`$]\!](e,c,s) =$

let $f = \lambda s'.\mathbf{sc}[\![$`while ( ` *exp* ` ) ` *stm*$]\!](e,c,s')$;

$e' = e[c/\mathtt{brk}]$;

$e'' = e'[f/\mathtt{con}]$;

in $\mathbf{sc}[\![stm]\!](e'',f,s)$

Given the state s', $\mathbf{sc}[\![$`while (`*exp*`)` *stm*$]\!](e,c,s')$ will be the "answer" of the C-function. Therefore, f is the continuation for `while (`*exp*`)` *stm* and its following statements. The above semantic equation for `do` corresponds to executing *stm* using the environment e'' and following *stm* by `while (`*exp*`)` *stm*.[1]

3.8. `while`. The `while` statement has the form

`while ( ` *exp* ` ) ` *stm*

The substatement *stm* is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

Since there is no bound on the number of times the body of a `while` loop might be executed, we will determine the continuation f, corresponding to the `while` and the statements that follow it, using a sequence of approximations $f_0, f_1, \cdots$ of f.

An intuitive view — no to be taken too literally — of the functions $f_0, f_1, \cdots$ is as follows. Let `loop()` be some parameterless C-function that loops forever. Then

```
f0  corresponds to  loop();
f1  corresponds to  if ( exp ) { stm loop(); }
f2  corresponds to  if ( exp ) { stm
                         if ( exp ) { stm loop(); }
                    }
```

and so on.

As discussed in §3.5-3.6, the loop body must use an environment in which appropriate continuations have been entered for the special identifiers `brk` and `con`. Since a `continue` causes re-execution of the `while`, at `con` we would like to enter f, the continuation for `while` and its succeeding statements. However, we have not yet determined f, since we need the meaning of the loop body before we can determine the meaning of the loop. During the determination of f_{i+1} we will use the best available approximation f_i of f for `con`.

Execution of a `while` loop either loops forever, or terminates after some n steps. The approximation f_{n+2} or for that matter f_{n+102} will correctly predict the "answer" of the C-function, so it suffices to make f the limit of the sequence $f_0, f_1, \cdots$, written $\sqcup\{f_i \mid i = 0,1,\cdots\}$ (see [set79a] for a discussion of $\sqcup$).

[1] The meaning of `do` *stm* `while ( ` *exp* ` ) ;` can be expressed strictly in terms of the meanings of *exp* and *stm* by replacing $\mathbf{sc}[\![$`while ( ` *exp* ` ) ` *stm*$]\!](e,c,s')$ by the appropriate expression from §3.8.

```
sc⟦while ( exp ) stm⟧(e,c,s) =
    let  $f_0 = \bot$;
        $f_{i+1} = \lambda s'.$ let $e' = e[c/\mathtt{brk}]$;
                            $e'' = e'[f_i/\mathtt{con}]$;
                            $(v,s'') = ⟦exp⟧(e,s')$;
                     in $v \neq 0 \rightarrow$ sc⟦stm⟧$(e'',f_i,s'')$,  $c(s'')$;
        $f = \sqcup \{f_i \mid i = 0, 1, \cdots\}$;
    in $f(s)$
```

An equivalent definition is given below. (The *fix* operator is discussed in [set79a].)

```
sc⟦while ( exp ) stm⟧(e,c,s) =
    let  $f = fix.\ \lambda g.$
                 $\lambda s'.$   let  $e' = e[c/\mathtt{brk}]$;
                             $e'' = e'[g/\mathtt{con}]$;
                             $(v,s'') = ⟦exp⟧(e,s')$;
                       in $v \neq 0 \rightarrow$ sc⟦stm⟧$(e'',g,s'')$,  $c(s'')$;
    in  $f(s)$
```

3.9. `for`. The `for` statement has the form

```
for ( exp1 ; exp2 ; exp3 ) stm
```

The intent of the `for` statement is similar to that of the following program fragment, but an exact analogy is frustrated by `continue` statements, §3.6.

```
exp1 ;
while ( exp2 ) {
    stm
    exp3 ;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0; the third expression often specifies an incrementation which is performed after each iteration.

Any or all of the expressions in the `for` statement may be dropped. However, we assume that during syntax analysis missing expressions are replaced by `1`. Note that `1;` is equivalent to the null statement since it does not change the state. Replacing a missing exp_2 by `1` makes the test of the implied `while` clause unconditionally true.

The meaning of the `for` statement is similar to that of the above program fragment with the implied `while`; the exception is that on encountering a `continue` in *stm*, exp_3 is evaluated before the implied `while` is repeated.

$$
\begin{array}{l}
\mathbf{sc}[\![\texttt{for (} exp_1 \texttt{;} exp_2 \texttt{;} exp_3 \texttt{) } stm]\!](e,c,s) = \\
\quad \textit{let} \;\; f = \textit{fix}\; \lambda g. \\
\qquad\qquad \lambda s'. \quad \textit{let} \;\; g' = \lambda t. \mathbf{sc}[\![exp_3 \texttt{ ;}]\!](e,g,t); \\
\qquad\qquad\qquad\qquad e' = e[c/\texttt{brk}]; \\
\qquad\qquad\qquad\qquad e'' = e'[g'/\texttt{con}]; \\
\qquad\qquad\qquad\qquad (v,s'') = [\![exp_2]\!](e,s'); \\
\qquad\qquad\qquad \textit{in}\; v \neq 0 \rightarrow \mathbf{sc}[\![stm]\!](e'',g',s''),\;\; c(s''); \\
\quad \textit{in}\; \mathbf{sc}[\![exp_1 \texttt{ ;}]\!](e,f,s)
\end{array}
$$

3.10. switch

3.10.1. *Discussion from* [ker78]. The `switch` statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

`switch (` *exp* `)` *stm*

The result of *exp* must be `int`. (Such type checking will be done elsewhere.) The statement *stm*, which is the `switch` body, is typically compound. Any statement within *stm* may be labeled with one or more case prefixes as follows:

`case` *constant_exp* `:`

where the constant expression must be `int`. No two of the case constants in the same switch may have the same value.

There may also be at most one statement prefix of the form

`default :`

When the `switch` statement is executed, its expression is evaluated and compared with each case constant. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression, and if there is a `default` prefix, control passes to the prefixed statement. If no case matches and if there is no `default` then none of the statements in the switch is executed.

`case` and `default` prefixes in themselves do not alter the flow of control, which continues unimpeded across such prefixes. To exit from a switch, see `break`, §3.5.

3.10.2. *Continuation semantics of* `switch`. Recall that the semantics of flow of control are given by using the continuation for the point that control flows to. In

`switch (` *exp* `)` *stm*

the value of *exp* determines the `case` that control flows to. We will therefore construct a function k_f that will map the value of *exp* to the continuation c'' of the selected `case`.

A function like k_f is sometimes called an *expression continuation*, which is distinct from the "statement" continuation c''. As a convention, the term continuation by itself will always refer to a statement continuation.

The domain K of expression continuations will be

$$\mathrm{K} = \mathrm{V} \rightarrow \mathrm{C}$$

The function k_f will obviously depend on the `switch` body generated by *stm*. The process of constructing k_f is reminiscent of the process in Figure 4, where starting with an initial continuation c_0, continuations $c_1, c_2, \cdots$ were constructed, until the continuation corresponding to the entire statement sequence had been constructed.

Starting with a suitable initial function k_0, we will proceed from the end of the statements generated by *stm* to the beginning. It will be convenient to start with k_0 mapping every value to a special constant continuation DUMMY. In this traversal, suppose a statement x is reached with function k_i. If x happens to be the statement `case` *const* : *st* then k_i will be updated to the function k_{i+1}, which maps *const* to c_{st}, the continuation for *st* and its following statements.

Now suppose x is the statement `default` : *st*. Since we started with k_0 mapping every value to the DUMMY continuation, $k_i(v)$ will still be DUMMY exactly when no `case` prefix for v has been encountered. k_{i+1} will therefore be such that

$$k_{i+1}(v) = k_i(v) = \text{DUMMY} \rightarrow c_{st}\ ,\ k_i(v)$$

where c_{st} is the continuation for *st* and its following statements. Note that if a `case` prefix for v is subsequently encountered, then at v, c_{st} will be "overwritten" by the appropriate continuation.

The above discussion describes the meaning for statements that will be given by valuation **sk** in §5. The valuation will be such that after the `switch` body *stm* has been processed, we will have determined k', where k' is almost the expression continuation k_f we set out to determine. k' will differ from k_f only if the `switch` body does not contain a `default`. When a `default` is missing, there will be values v for which $k'(v)$ will be DUMMY. Since the `switch` body is skipped if `default` is missing, all we need to do is to modify k' so that instead of getting DUMMY at v we get c the continuation for the statements following the `switch`.[2]

```
sc⟦switch ( exp ) stm⟧(e,c,s) =
    λs. let  k = λa:V. (DUMMY:C);
             e' = e[c/brk];
             k' = sk⟦stm⟧(e',k,c);
             k_f = λa. k'(a) = DUMMY → c,  k'(a);
             (v,s') = ⟦exp⟧(e',s);
             c'' = k_f(v);
        in c''(s')
```

Note that while processing the `switch` body *stm* we must do two things: the function k_0 must be updated as `case` and/or the `default` prefixes are encountered; moreover, the statement continuation corresponding to flow through the *stm* must also be accumulated. Details of the updating of k_0 will be given in §5 when valuation **sk** is discussed. Accumulation of the statement continuation ignores all `case` and the `default` prefixes:

[2] The check that no two `case` constants in the same `switch` have the same value, and that there is at most one `default` prefix, can also be done during the traversal that constructs k_f. The details of how this is done are not significant, but one possible "implementation" is as follows. Let DEF be another special constant continuation. When `default` : *st* is encountered, construct k_{i+1} such that $k_{i+1}(v)$=DEF whenever $k_i(v)$=DUMMY. (The continuation c_{st} for *st* and its following statements will need to be remembered somewhere, say in k_{i+1} at a special value *vd* distinct from any other value.) A `case` prefix for value v will be encountered for the first time if $k_i(v)$ is either DUMMY or DEF. When all statements in the `switch` body have been traversed, then all instances of DEF will have to be replaced by the remembered continuation at the special value *vd*.

In the semantics of `switch` we have assumed that the above check has been made.

$\mathbf{sc}[\![$`case` *constant_exp* : *stm*$]\!](e,c,s) = \mathbf{sc}[\![$*stm*$]\!](e,c,s)$

$\mathbf{sc}[\![$`default` : *stm*$]\!](e,c,s) = \mathbf{sc}[\![$*stm*$]\!](e,c,s)$

3.11. *Compound statement.* So that several statements can be used where one is expected, the compound statement is provided:

{ *stm_s* }

Note that (contrary to [ker78]) declarations are not permitted within compound statements i.e. "blocks" are not allowed. Since we do not have a clear idea of what the semantics of a `goto` into a block ought to be, we do not want to allow blocks and also allow a `goto` to jump anywhere within a function. We have chosen to remain true to the language as it is implemented [joh78,rit78a] and allow a `goto` to jump anywhere within a C-function. It is then conveninent to assume that declarations have been preprocessed away by renaming identifiers to restrict scope; moving the declarations of the renamed identifiers to the head of the function definition; and converting initializations into explicit assignments. The braces { and } are ignored:

$\mathbf{sc}[\![\{$ *stm_s* $\}]\!](e,c,s) = \mathbf{sc}[\![$*stm_s*$]\!](e,c,s)$

The semantics of *stm_s* are given as follows.

```
sc[[stm_s]](e,c,s):  (V,S)
    | /* empty */
        = $c(s)$
    | stm_s stm
        = $\mathbf{sc}[\![stm\_s]\!](e,c',s)$   where   $c' = \lambda t.\ \mathbf{sc}[\![stm]\!](e,c,t)$
```

3.12. `goto`. Control may be transferred unconditionally by means of the statement

`goto` *identifier* ;

The identifier must be a label (see below) located in the current function. As with `break` and `continue`, §3.5-3.6, we assume that the continuation corresponding to the statement labeled with *identifier* has already been entered in the environment. Thus all that needs to be done here is to look up this continuation in the environment and apply it to the state.

$\mathbf{sc}[\![$`goto` *identifier* ;$]\!](e,c,s) = c'(s)$ **where** $c' = e($*identifier*$)$

Labeled statement. Any statement may be preceded by label prefixes of the form

identifier :

which serve to declare the identifier as a label. The only use of a label is as a target of a `goto`. The scope of a label is the current function, excluding any sub-blocks in which the same identifier has been redeclared.

We will discuss in §4 how continuations corresponding to labeled statements will be entered into the environment.

The following semantics for labeled statements reflects the fact that flow of control is not impeded by the presence of statement labels:

$\mathbf{sc}[\![$*identifier* : *stm*$]\!](e,c,s) = \mathbf{sc}[\![$*stm*$]\!](e,c,s)$

4. Environments for `goto` Statements

In §3, the semantics of flow of control were given by using the continuation for the point that control flowed to. Suppose x is a statement with label *id*. Then the meaning of `goto` *id* `;` was given by using the continuation c_x for x and its following statements. In §3.12 it was assumed that the continuation c_x could be looked up in the environment: in particular, with environment e, it was expected that $c_x = e(id)$.

We will now discuss how continuations like c_x are entered into the environment. A labeled statement like *id* : *stm* will update the environment by associating with *id* the continuation for *stm* and its following statements. We will therefore use a new valuation, **se**, for statements:

$$\mathbf{se}[\![stm]\!](e,c)\text{: } \mathrm{E}$$

Given a labeled statement *id* : *stm*, valuation **se** will determine continuation c' for *stm* and its following statements, and will then modify the environment so that continuation c' is associated with label *id*. We already have a way (from §3) of constructing the continuation c': $c' = \lambda s.\ \mathbf{sc}[\![stm]\!](e,c,s)$, using valuation **sc**. [3]

A concise listing of the meaning of statements under valuation **se** may be found in Appendix B.

4.1. *Labeled statements.* The following meaning for labeled statements takes into account the fact that a labeled statement may be compound, and may itself contain further labeled statements.

$$\begin{aligned}
&\mathbf{se}[\![id : stm]\!](e,c) = \\
&\quad \textit{let}\ \ e' = \mathbf{se}[\![stm]\!](e,c); \\
&\qquad\quad c' = \lambda s.\ \mathbf{sc}[\![stm]\!](e,c,s); \\
&\quad \textit{in}\ \ e'[c'/id]
\end{aligned}$$

4.2. *Basic statements.* If *stm* is just a `return`, null, expression, `break`, `continue`, or `goto` statement, it has no label and cannot affect the environment:

$$\mathbf{se}[\![stm]\!](e,c) = e$$

4.3. *Embedded labels.* The statement `if` (*exp*) *stm* is not a labeled statement, but there may be labels within *stm*. A jump to a label within *stm* will bypass the test of *exp*, so *exp* will play no role in **se**[[`if` (*exp*) *stm*]]. In fact the effect on the environment of this conditional will be exactly the same as that of *stm* by itself:

$$\mathbf{se}[\![\texttt{if}\ (\ exp\ )\ stm]\!](e,c) = \mathbf{se}[\![stm]\!](e,c)$$

Similar remarks apply to the `switch` statement. `case` and `default` prefixes are also ignored.

In `if` (*exp*) stm_1 `else` stm_2, there may be labels within both stm_1 and stm_2, so the effect of both stm_1 and stm_2 needs to be accumulated. Since control flows to exactly the same point from stm_1 and stm_2 the same continuation c is used in $\mathbf{se}[\![stm_2]\!](e',c)$ and $\mathbf{se}[\![stm_1]\!](e,c)$:

$$\mathbf{se}[\![\texttt{if}\ (\ exp\ )\ stm_1\ \texttt{else}\ stm_2]\!](e,c) = \mathbf{se}[\![stm_2]\!](e',c) \quad \textit{where}\ \ e' = \mathbf{se}[\![stm_1]\!](e,c)$$

Since the labels in stm_1 and stm_2 will be distinct, the order in which the effect of stm_1 and stm_2 is accumulated is not significant, and we could equivalently have accumulated the effect of stm_2 before that of stm_1.

[3] The interdependence of valuations **se** and **sc** will be discussed in §4.5.

4.4. *Jumps into loops.* Recall from §3.7 that the intent of the following two program fragments is similar (even though the fragments are not equivalent):

```
do stm while ( exp ) ;
stm while ( exp ) stm
```

The meaning of the do statement below is inspired by the second program fragment.

In the second program fragment, let c' be the continuation for the `while` statement. Since *stm* is followed by the `while`, we will use c' when "processing" *stm*. In order to properly handle `break` and `continue` statements within *stm*, as in §3.7, appropriate continuations are entered into the environment (see e'' below) for the special identifiers `brk` and `con`.

```
se⟦do stm while ( exp ) ;⟧(e,c) =
    let c' = λs. sc⟦while ( exp ) stm⟧(e,c,s);
        e' = e[c/brk];
        e'' = e'[c'/con];
    in se⟦stm⟧(e'',c')
```

The meaning of `while` statements is almost identical to that of `do` statements, since a jump into *stm* bypasses the loop test.

```
se⟦while ( exp ) stm⟧(e,c) =
    let c' = λs. sc⟦while ( exp ) stm⟧(e,c,s);
        e' = e[c/brk];
        e'' = e'[c'/con];
    in se⟦stm⟧(e'',c')
```

A little more care is needed with `for` statements because of the initialization and incrementation expressions exp_1 and exp_3. Based on the semantics in §3.9,

```
for ( exp₁ ; exp₂ ; exp₃ ) stm
```

is equivalent to

```
exp₁ ; for ( 1; exp₂ ; exp₃ ) stm
```

since `1;` is equivalent to a null statement. Furthermore, the intent of the `for` statement is similar to that of the following program fragment:

```
exp₁ ; if ( exp₂ ) { stm
                     exp₃ ;
                     for ( 1; exp₂ ; exp₃ ) stm
                     }
```

A jump to a label within *stm* will bypass exp_1 ; and the test of exp_2 so we need only consider the first instance of *stm* and the two statements that follow it in the above program fragment:

```
se⟦for ( exp₁ ; exp₂ ; exp₃ ) stm⟧(e,c) =
        let c′ = λs. sc⟦for ( 1 ; exp₂ ; exp₃ ) stm⟧(e,c,s);
            e′ = e[c/brk];
            c″ = λs. sc⟦exp₃ ;⟧(e,c′,s);
            e″ = e′[c″/con];
        in se⟦stm⟧(e″,c″)
```

4.5. *Interdependence of* **se** *and* **sc**. The specification of valuation **se** is not enough for setting up the environment needed for the semantics of §3. Here we will discuss what needs to be done, but the environment will not be set up until declarations are discussed.

Recall that given a labeled statement *id* : *stm*, valuation **se** determines continuation c' for *stm* and its following statements, and then modifies the environment so that continuation c' is associated with label *id*. The continuation c' is of course determined using valuation **sc** of §3. The problem is that *stm* may well be a compound statement containing a `goto`, so an environment is needed to determine c'.

This circularity is resolved by starting with an intial environment e_0 that is used by **sc** to determine continuations. These continuations in turn are used by valuation **se** to construct a new environment e_1. Iterating this process, we will get a sequence of environments $e_0, e_1, \cdots$. This sequence will actually be a chain of environments, with least upper bound, say *en*.

In constructing e_{i+1} from e_i we have to be sure to include the continuation for each label, so the valuations **se** and **sc** will have to be applied to the collection of all statements in a C-function. The only time we are handed the collection of all statements in a C-function is when the C-function is defined. For this reason, the construction of the environment *en* will not become explicit until the semantics of C-function definitions are given.

The construction of *en* also ensures that proper continuations are associated with any labels that might be in loops set up by `goto` statements.

5. Selecting `case`'s in `switch` Bodies

The `switch` statement, §3.10, causes control to be transferred to one of several statements depending on the value of an expression. In

`switch ( exp ) stm`

the value of *exp* determines the `case` that control flows to. We will therefore construct a function, referred to as k_f in §3.10.2, that will map the value of *exp* to the continuation of the selected `case`.

Determination of the expression continuation k_f is similar to, and simpler than, the determination of an appropriate environment for `goto` statements in §4. Instead of entering continuations into environments when labeled statements are encountered, we will enter continuations into an expression continuation when a `case` or `default` prefix is encountered. In fact, Appendix C containing a concise specification of the valuation **sk** to be discussed here was created by minimally editing Appendix B, which contains a specification of valuation **se**, §4.

Valuation **sk** will be such that:

$$\mathbf{sk}[\![stm]\!](e,k,c):\ \mathrm{K}\quad where\quad \mathrm{K} = \mathrm{V} \rightarrow \mathrm{C}$$

The construction of an expression continuation by valuation **sk** will of course depend on valuation **sc**, which is used to construct statement continuations, but **sc** and **se** will not depend on **sk**. Intuitively, there is no circularity in the specification of **sk** because control flows from a `switch` to a `case` in the switch body, so a `switch` by itself cannot cause a loop.

5.1. `case` *and* `default` *prefixes.* The following meaning for `case` and `default` prefixes strongly influences the meaning of the other statements.

```
| case constant_exp : stm
    = let   k' = sk[[stm]](e,k,c);
            c' = λs. sc[[stm]](e,c,s);
            v = [[constant_exp]](e);
            k" = k'[c'/v];
      in k"

| default : stm
    = let   k' = sk[[stm]](e,k,c);
            c' = λs. sc[[stm]](e,c,s);
            k" = λv. k'(v) = DUMMY → c', k'(v);
      in k"
```

5.2. *Other statements.* As in §4, statements without prefixes will not affect the expression continuation constructed by **sk**. If a `switch` occurs within the body of an outer `switch`, then control cannot flow from the outer `switch` to a `case` in the inner `switch`. Continuations for `case` prefixes in the body of the inner `switch` are therefore not visible to the outer `switch`, so `switch` statements do not change the expression continuation:

$$\mathbf{sk}[\![\texttt{switch}\ (\ exp\ )\ stm]\!](e,k,c) = k$$

For all remaining statements, valuation **sk** behaves similarly to valuation **se**, §4.

Acknowledgments.

It has been a pleasure discussing the semantics of C with F. T. Grampp and A. R. Koenig. S. C. Johnson, B. W. Kernighan, and D. M. Ritchie have patiently fielded a number of questions about C. M. D. McIlroy's comments on an earlier draft were very helpful.

§3 of this paper was prepared by extensively editing §9 of the C Reference Manual in [ker78]. I want to thank B. W. Kernighan and D. M. Ritchie for their permission to use the material from [ker78].

Ravi Sethi

MH-1271-RS-unix

Ravi Sethi

Att.
Appendices A, B, C,
References

Appendix A. Statement Continuations

Let V, E, and S be the domains of values, environments, and states, respectively. Since C-functions have side-effects, we will view C-functions as returning *answers* which are elements of (V,S). Corresponding to the statements in a C-function is a (statement) continuation in domain C, which maps the state s, that the statements are reached with, to the answer of the C-function.

The continuation for the statements in a C-function will be determined using valuation **sc**, §2-3, which is specified below. In $\mathbf{sc}[\![stm]\!](e,c,s)$, s is the state that *stm* is reached with, and c is the continuation for the statements that follow *stm*.

Figure 1 contains an index to the detailed discussion of each construct.

```
sc⟦stm⟧(e,c,s): (V,S)
  | return ;
      — (grb,s)
  | return exp ;
      — ⟦exp⟧(e,s)
  | ;
      — c(s)
  | exp ;
      — c(s′)   where  (v,s′) = ⟦exp⟧(e,s)
  | if ( exp ) stm
      — let (v,s′) = ⟦exp⟧(e,s);
        in  v ≠ 0 — sc⟦stm⟧(e,c,s′) ,  c(s′)
  | if ( exp ) stm₁ else stm₂
      — let (v,s′) = ⟦exp⟧(e,s);
        in  v ≠ 0 — sc⟦stm₁⟧(e,c,s′) ,  sc⟦stm₂⟧(e,c,s′)
  | break ;
      — c′(s)   where  c′ = e(brk)
  | continue ;
      — c′(s)   where  c′ = e(con)
  | do stm while ( exp ) ;
      — let    f = λs′.sc⟦while ( exp ) stm⟧(e,c,s′);
               e′ = e[c/brk];
               e″ = e′[f/con];
        in sc⟦stm⟧(e″,f,s)
```

```
| while ( $exp$ ) $stm$
    — let   $f = fix\ \lambda g.$
                          $\lambda s'.$   let  $e' = e[c/\texttt{brk}];$
                                               $e'' = e'[g/\texttt{con}];$
                                               $(v,s'') = [\![exp]\!](e,s');$
                                          in $v \neq 0 \rightarrow sc[\![stm]\!](e'',g,s''),\ c(s'');$
      in  $f(s)$

| for ( $exp_1$ ; $exp_2$ ; $exp_3$ ) $stm$
    — let   $f = fix\ \lambda g.$
                          $\lambda s'.$   let  $g' = \lambda t.\, sc[\![exp_3\ ;]\!](e,g,t);$
                                               $e' = e[c/\texttt{brk}];$
                                               $e'' = e'[g'/\texttt{con}];$
                                               $(v,s'') = [\![exp_2]\!](e,s');$
                                          in $v \neq 0 \rightarrow sc[\![stm]\!](e'',g',s''),\ c(s'');$
      in  $sc[\![exp_1\ ;]\!](e,f,s)$

| switch ( $exp$ ) $stm$
    — $\lambda s.$   let  $k = \lambda a{:}\mathrm{V}.\ (\mathrm{DUMMY{:}C});$
                   $e' = e[c/\texttt{brk}];$
                   $k' = sk[\![stm]\!](e',k,c);$
                   $k_f = \lambda a.\ k'(a) = \mathrm{DUMMY} \rightarrow c,\ k'(a);$
                   $(v,s') = [\![exp]\!](e',s);$
                   $c'' = k_f(v);$
              in  $c''(s')$

| case $constant\_exp$ : $stm$
    — $sc[\![stm]\!](e,c,s)$
| default : $stm$
    — $sc[\![stm]\!](e,c,s)$
| { $stm\_s$ }
    — $sc[\![stm\_s]\!](e,c,s)$
| goto $identifier$ ;
    — $c'(s)$   where   $c' = e(identifier)$
| $identifier$ : $stm$
    — $sc[\![stm]\!](e,c,s);$

;
```

```
sc⟦stm_s⟧(e,c,s): (V,S)
    | /* empty */
        — c(s)
    | stm_s stm
        — sc⟦stm_s⟧(e,c′,s)   where   c′ = λt. sc⟦stm⟧(e,c,t)
    ;
```

Appendix B. Environments for goto Statements

Flow of control due to say `goto` *id* ; is formalized by using the continuation for the point that control flows to. In this case, if there is a labeled statement *id* : x, and c_x is the continuation for x and its following statements, then

$$\mathbf{sc}[\![\texttt{goto}\ id\ ;]\!](e,c,s) = c_x(s)$$

We want environment e to be such that $c_x = e(id)$.

This appendix specifies valuation **se**, §4, which is used in the construction of environments like e.

```
se[stm](e,c): E

        /*  The meaning of labeled statements influences the meaning of
            of all other statements, §4.1.
            */

    | identifier : stm
        — let   e' = se[stm](e,c);
                c' = λs. sc[stm](e,c,s);
          in  e'[c'/id]

        /*  Statements without labels cannot change the environment, §4.2. */

    | return ;
        — e
    | return exp ;
        — e
    | ;
        — e
    | exp ;
        — e
    | break ;
        — e
    | continue ;
        — e
    | goto identifier ;
        — e
    /*  The effect of embedded labels needs to be propagated, §4.3. */
| if ( exp ) stm
      — se⟦stm⟧(e,c)
| switch ( exp ) stm
      — se⟦stm⟧(e,c)
| case constant_exp : stm
      — se⟦stm⟧(e,c)
| default : stm
      — se⟦stm⟧(e,c)
| { stm_s }
      — se⟦stm_s⟧(e,c)
| if ( exp ) stm₁ else stm₂
      — se⟦stm₂⟧(e′,c)   where  e′ = se⟦stm₁⟧(e,c)

    /*  A jump into a loop causes execution of the loop body followed by
        execution of the entire loop (provided no other jumps take place).
        This intuitive idea is used to give meaning for the looping constructs, §4.4.
        */

| do stm while ( exp ) ;
      — let    c′ = λs. sc⟦while ( exp ) stm⟧(e,c,s);
               e′ = e[c/brk];
               e″ = e′[c′/con];
        in  se⟦stm⟧(e″,c′)
| while ( exp ) stm
      — let    c′ = λs. sc⟦while ( exp ) stm⟧(e,c,s);
               e′ = e[c/brk];
               e″ = e′[c′/con];
        in  se⟦stm⟧(e″,c′)
| for ( exp₁ ; exp₂ ; exp₃ ) stm
      — let    c′ = λs. sc⟦for ( 1 ; exp₂ ; exp₃ ) stm⟧(e,c,s);
               e′ = e[c/brk];
               c″ = λs. sc⟦exp₃ ;⟧(e,c′,s);
               e″ = e′[c″/con];
        in  se⟦stm⟧(e″,c″)
;
```

```
se[stm_s](e,c): E
    | /* empty */
        → e
    | stm_s stm
        → let   c' = λs. sc[stm](e,c,s);
                e' = se[stm](e,c);
          in se[stm_s](e',c')
    ;
```

Appendix C. Selecting `case`'s in `switch` Bodies

In `switch ( `*exp*` ) `*stm*, §3.10, the value of *exp* determines the `case` that control flows to. Valuation **sk**, §3.10.2,§5, will be used to construct an expression continuation k that will map the value of *exp* to the continuation for the selected `case`.

```
sk[stm](e,k,c): K

        /* The primary purpose of sk is to enter the continuations for
           case and default prefixes into k, §5.1.
           */

    | case constant_exp : stm
        — let   k' = sk[stm](e,k,c);
                c' = λs. sc[stm](e,c,s);
                v = [constant_exp](e);
                k'' = k'[c'/v];
          in k''
    | default : stm
        — let   k' = sk[stm](e,k,c);
                c' = λs. sc[stm](e,c,s);
                k'' = λv. k'(v) = DUMMY → c', k'(v);
          in k''

        /* Statements without prefixes cannot change k. */

    | return ;
        — k
    | return exp ;
        — k
    | ;
        — k
    | exp ;
        — k
    | break ;
        — k
    | continue ;
        — k
    | goto identifier ;
        — k
    /* Control may not flow from an outer switch
       into the body of an inner switch.
       Since sk is applied to the body of an outer switch, any case
       prefixes in stm are not visible to the outer switch.
       k will therefore remain unchanged.
       */

| switch ( exp ) stm
    = k

    /* The effect of embedded labels needs to be propagated. */

| if ( exp ) stm
    = sk⟦stm⟧(e,k,c)
| identifier : stm
    = sk⟦stm⟧(e,k,c);
| { stm_s }
    = sk⟦stm_s⟧(e,k,c)
| if ( exp ) stm₁ else stm₂
    = sk⟦stm₂⟧(e,k',c)   where   k' = sk⟦stm₁⟧(e,k,c)
```

```
        /*  A jump into a loop causes execution of the loop body followed by
            execution of the entire loop (provided no other jumps take place).
            This intuitive idea is used to give meaning for the looping constructs.
            */

      | do stm while ( exp ) ;
            — let   c′ = λs. sc⟦while ( exp ) stm⟧(e,c,s);
                    e′ = e[c/brk];
                    e″ = e′[c′/con];
              in sk⟦stm⟧(e″,k,c′)
      | while ( exp ) stm
            — let   c′ = λs. sc⟦while ( exp ) stm⟧(e,c,s);
                    e′ = e[c/brk];
                    e″ = e′[c′/con];
              in sk⟦stm⟧(e″,k,c′)
      | for ( exp1 ; exp2 ; exp3 ) stm
            — let   c′ = λs. sc⟦for ( 1 ; exp2 ; exp3 ) stm⟧(e,c,s);
                    e′ = e[c/brk];
                    c″ = λs. sc⟦exp3 ;⟧(e,c′,s);
                    e″ = e′[c″/con];
              in sk⟦stm⟧(e″,k,c″)
      ;

   sk⟦stm_s⟧(e,k,c): K
      | /* empty */
            — k
      | stm_s stm
            — let   c′ = λs. sc⟦stm⟧(e,c,s);
                    k′ = sk⟦stm⟧(e,k,c);
              in sk⟦stm_s⟧(e,k′,c′)
```

References

joh78 S. C. Johnson, "A portable compiler: theory and practice," *Fifth ACM Symposium on Principles of Programming Languages*, pp. 97-104 (January 1978).

ker78 B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N.J. (1978).

mss76 P. D. Mosses, "Compiler generation using denotational semantics," pp.436-441 in *Mathematical Foundations of Computer Science 1976*, Lecture Notes in Computer Science 45, Springer-Verlag, Berlin (1976).

rit78a D. M. Ritchie, "A tour through the UNIX C compiler," unpublished manuscript, Bell Laboratories, Murray Hill, NJ (1978).

rit78b D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "UNIX time-sharing system: the C programming language," *Bell Sys. Tech. J.* 57(6) pp. 1991-2019 (1978).

scs71 D. Scott and C. Strachey, "Towards a mathematical semantics for computer languages," pp.19-46 in *Proceedings of the Symposium on Computers and Automata*, Polytechnic Press, Brooklyn, N.Y. (April 1971).

set79a R. Sethi, "Semantics of the C programming language, part 0: prelude," unpublished manuscript, Bell Laboratories, Murray Hill, N.J. (1979).

sto77 J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, MA (1977).

ten76 R. D. Tennent, "The denotational semantics of programming languages," *Comm. ACM* 19 (8) pp.437-453 (August 1976).