MH

1353
UNPL

Bell Laboratories **Cover Sheet for Technical Memorandum**

The information contained herein is for the use of employees of Bell Laboratories and is not for publication. (See GEI 13.9-3)

Title- **Semantics of the C programming language, part 0: prelude**

Date- **January 2, 1979**

TM- **79-1271-2**

Other Keywords-

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

ABSTRACT

This is the first of a sequence of papers defining the semantics of the C programming language. Of the three methods — operational, denotational, and axiomatic — that have been used to specify the semantics of reasonably complete languages, the denotational method has been chosen to specify C. In this prelude, a very simple language with assignments and while loops will be used to illustrate the semantic method.

Pages Text 23	Other 7	Total 30	WORKING COPY
No. Figures 10	No. Tables 0	No. Refs. 59	

BELL TELEPHONE LABORATORIES, INC.

TM-79-1271-2

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 E
BARCLAY,DAVID K
BARDOFSKY,ALLEN
BASEIL,RICHARD J
BAUER,ANDREW E
BAUER,BARBARA T
BAUER,HELEN A
<BAUGH,C R
BEDNAR,JOSEPH A,JR
BEMER,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 I
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 R
BOIVIE,RICHARD H
BONANNI,L E
BORDELON,EUGENE P
BORISON,ELLEN A
BOURNE,STEPHEN R
BOWERMAN,REBECCA ELAINE
BOWYER,L RAY
<BOYCE,W M
BOYER,PHYLLIS J
BOYLE,W S
BRADLEY,A H
BRAM,ALAN
<BRANDT,RICHARD B
BRAUNE,DAVID P
BRIGGS,GLORIA A
BROSS,JEFFREY D

COVER SHEET ONLY TO

BROWN,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 R
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
<CROME,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 A
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
FEEIDUN,K K
FINUCANE,J J
FISCHER,HERBERT B
FISCHER,MICHAEL T
FISHER,EDWARD R
FISHMAN,DANIEL H
FLANDRENA,R
FLEISLEBER,RAYMOND C.
FONG,K T
FORTNEY,V J
FOUNTOUKIDIS,A
FOWLER,GLENN D
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
GALENZ,RENATO N
GALLANT,R J
GAMA,JORGE L
GARST,BLAINE,JR
<GATES,G W
GAY,FRANCIS A
GEARY,M J
GEERS,T J,JR
GERGSEN,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,R L
GROSS,ARTHUR G
GRUENWALD,JOHN
GRZELAKOWSKI,MAUREEN E
GUIDI,PIER V
GUNTHER,F L
GUSTAVSON,J H
GUTTMAN,NEWMAN
HAFER,R 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
HEAGENHAN,C B
<HESSELGRAVE,MARY E
HOCHBERG,GLENN A
HOEHN,MARIE J
HOLTMAN,JAMES P
HOREWEDEL,J R
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,C D
IRVINE,M M
ISMAN,MARSHALL A
<IVIE,EVAN L
JACKOWSKI,D J
JACKSON,H M,2ND
JACOBS,H S
JAMES,J W
<JENSEN,PAUL D

...
564 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

CCPLGP = COMPUTING/PROGRAMMING LANGUAGES/GENERAL PURPOSE

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.
4. INDICATE WHETHER MICROFICHE OR PAPER IS DESIRED.

MC CORRESPONDENCE FILES
MC 5C101

TM-79-1271-2
TOTAL PAGES

PLEASE SEND A COMPLETE

( ) MICROFICHE COPY ( ) PAPER COPY

TO THE ADDRESS SHOWN ON THE OTHER SIDE.

Bell Laboratories

Subject: **Semantics of the C programming language, part 0: prelude**
Case- 39199 -- File- 39199-11

date: **January 2, 1979**

from: **Ravi Sethi**

TM: **79-1271-2**

MEMORANDUM FOR FILE

1. Introduction

By far the most popular method for specifying a programming language is to specify the syntax using some variant of BNF notation [bac59] and then give an informal description of the semantics. This was the method used for the influential Algol 60 report [nau63]. While natural language descriptions are very good at conveying the spirit of a language construct, important details may get left out, either because they are overlooked, or because they are hard to express.

It is for cleanly and concisely specifying the precise details of the meaning of a language that we turn to formal specifications of programming languages.

1.1. *An example.* Here we consider the example of the `for` statement in the C programming language [ker78], where it is difficult to express the meaning of the `for`, even if we take the meaning of the `while` statement as being understood. The `for` statement in C has the form

`for` (exp_1 ; exp_2 ; exp_3) *stm*

The intent of the `for` statement is similar to that of the following program fragment, but as we shall see, an exact analogy is frustrated by `continue` statements:

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

Assignments may be embedded within expressions in C, and an expression followed by a semicolon is a statement. In the statement *exp ;* the expression is evaluated for its side-effects — the value of the expression is thrown away.

Thus the first expression exp_1 specifies initialization for the loop; the second exp_2 specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0; the third expression exp_3 often specifies an incrementation which is performed after each iteration.

The difference between the `for` statement and the above program fragment is that a `continue` statement within *stm* is treated differently in the two cases. In the `while`, when a `continue` in *stm* is encountered, the statements following the `continue` (including exp_3) are

skipped and the test exp_2 is reevaluated. In the `for` statement, however, when a `continue` in *stm* is encountered, exp_3 is evaluated before the next iteration begins. (Notice that we do not have a precise notion of what an iteration is.)

Using `goto` statements and labels we could indeed give the meaning of the `for` statement, in effect by compiling it into an intermediate language. Quite apart from the fact that the meaning of the intermediate language would then have to be specified, we would have lost the simplicity of a natural language description.

1.2. *Abstract.* The purpose of this work is to provide a readable and precise specification of the semantics of the C programming language. We would like the specification to be precise enough that a compiler can be constructed from the specification. Of the three methods — operational, denotational, and axiomatic — that have been used to specify the semantics of reasonably complete languages, the denotational method seems best suited for our purposes.

The exact boundary between operational and denotational semantics is neither very clear, nor very important. In this prelude we will survey the operational and denotational approaches to the semantics of programming languages. The language we will use to illustrate both these approaches has statements with the following syntax.

stm ::= *id* `:=` *exp* `;`
| *stm stm*
| `if (` *exp* `) {` *stm* `}`
| `while (` *exp* `) {` *stm* `}`

In the discussions that follow, it will sometimes be convenient to drop { and } and write the conditional and while statements as

`if (` *exp* `)` *stm*

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

Expressions will be discussed in some detail in section 2. We intend *exp* to generate the usual arithmetic expressions, with no side-effects.

Some bibliographic notes appear in section 8.

1.3. *Operational semantics.* In practice, when a question arises about the meaning of a program, it is often resolved by actually compiling and running the program. A compiler on a given computer clearly constitutes a precise specification of a language. Unfortunately, this specification is not very usable since it depends too intimately on the details of the particular compiler on the particular computer.

A machine independent specification of a language can be constructed by abstracting away unnecessary details and defining a simplified hypothetical computer: the language can then be specified by writing an interpreter for the language using the machine language of the hypothetical computer.

McCarthy [mcc62] suggested in 1962 that the semantics of a language be specified as follows:

1. Isolate the information that is needed to describe the *state* of a computation. For the purposes of this prelude, a state s will give the current value of each program identifier.
2. "The meaning of a program is defined by its effect on the state vector." "In the case of ALGOL we should have a function $s' = algol(p,s)$ which gives the value s' of the state vector after the ALGOL program p has stopped, given that it was started at its beginning and that the state vector was initially s." (In these quotes from [mcc62] we have used the symbols p and s instead of π and ξ.)

McCarthy's interpreter for LISP [mcc60] became the de facto standard for the language [mcc78], so LISP might be viewed as the first language to be defined interpretively. Landin defined Algol 60 by first translating Algol 60 programs into expressions in a modified λ-calculus, and then interpreting the resulting expressions on an abstract machine called the SECD machine [lan64,lan65]. McCarthy's and Landin's work was a starting point for the Vienna definition of PL/1 [luc69].

1.4. *Denotational semantics.* The beginnings of denotational semantics are evident in the 1964 paper [str64] by Strachey. Although Strachey was considerably influenced by Landin, he preferred to use an approach that did not depend in any way on the workings of a machine, however abstract the machine might be.

Here we will give the barest hint of the method of denotational semantics [scs71], which will be discussed in section 5.

Let a *state s* map an identifier to a value. Exploiting the structure of an expression, it is easy to devise a function which maps an expression *exp* and a state *s* to a value. (We have assumed that there are no side-effects during expression evaluation.)

Let us take the meaning of a statement *stm* to be a function f from states to states. Suppose *stm* is the assignment *id* := *exp* ; and we are given a state s. Then $f(s)$ will be some state s', which maps all identifiers except *id* to the values that the original state s would have: *id* will be mapped to the value of *exp* in the original state s. For a compound statement stm_1 stm_2, if f_1 is the meaning of stm_1 and f_2 the meaning of stm_2, then the meaning of stm_1 stm_2 will be the function $f_2 \circ f_1$ from states to states. In general, for each syntactic rule that constructs a statement from its constituents, a semantic rule will be given to construct the meaning of the statement from the meanings of the constituents.

Taking the meaning of a statement to be a function from states to states, and giving the meaning of the compound statement stm_1 stm_2 by functional composition as above, leads to difficulties when `goto` statements are included in the language. These difficulties can be resolved by using *continuations*, which will be examined in section 6.

Denotational specifications have been given for a number of languages, including Algol 60 [mss74,hen78] and Pascal [ten77].

1.5. *Distinguishing operational and denotational specifications.* A multiplicity of notations have been used for specifying the semantics of programming languages. In some cases the notation is so extensive that the basic concepts underlying the specification are hopelessly lost. The situation is complicated by the fact that some notations can be used to express both operational and denotational specifications.

We will initially use distinct notations for operational and denotational semantics so that it will be clear at a glance which style is being discussed. The operational semantics in section 3 will be expressed using both notations in order to compare and contrast it with the denotational semantics in section 5.

An easy and effective way of telling operational and denotational specifications apart is to look at the semantics of while statements. (If the language does not have while statements, then look for the semantics of any construct that may lead to an infinite computation.) Regardless of which method is used, we expect `while` (*exp*) *stm* to have the same meaning as

`if` (*exp*) { *stm* `while` (*exp*) *stm* }

Operational methods specify the meaning of the while statement using some formalization of: if *exp* evaluates to false, then skip to the end of the while statement; otherwise execute *stm* and reexecute the while statement. In this way, the meaning of a while statement is specified *in terms of* the meaning of the while statement.

Denotational methods synthesize the meaning of the while statement just in terms of the

meanings of *exp* and *stm*, and *not* in terms of the entire while statement. Since execution of a while statement need not terminate, denotational methods need some mechanism for finitely expressing the meaning of a possibly infinite computation. This point will be discussed further in section 5.1 after recursive definitions are discussed in section 4.

2. Expressions: a small example

In order to set up a programming language we must begin by setting up the syntax, which specifies what programs in the language look like. After specifying the syntax, we still do not have a programming language until we set up its semantics, which specifies what programs in the language mean.

In this section we introduce some basic concepts and notations for specifying semantics, by considering arithmetic expressions over - (subtraction) and * (multiplication). The language of expressions is simple enough that we can focus on the semantic method without being sidetracked by the intricacies of the language being defined. The language is in fact so simple that there is no distinction between its operational and denotational semantics. Substantive distinctions between the two methods can be made after `while` statements are included.

The starting point for a semantic specification is a syntax for the language. We would like to defer a discussion of the choice of syntax on which semantics are based until section 2.4.

Meanwhile, as a further simplification, we will write expressions in prefix notation e.g. `*-abc` instead of `(a-b)*c`. Later in this section infix expressions like `(a-b)*c` will be treated.

The syntax of prefix expressions is given by

exp ::= - *exp exp*
| * *exp exp*
| *id*

Here *id* represents an arbitrary identifier. In examples, the letters `a`, `b`, and `c` will be used for identifiers. In productions we will sometimes use subscripts, as in *exp* ::= - exp_1 exp_2, with the sole intent of distinguishing the instances of *exp*.

Even though there is no real distinction between operational and denotational semantics for prefix expressions, a natural progression is to introduce operational semantics and then denotational semantics for expressions.

2.1. *Domains.*

Each syntactic object, like an expression or an identifier, *denotes* a semantic object: in the case of expressions, a value chosen from say the set of integers. Instead of the term "set", the term *domain* is generally used, so we talk about the domain of expressions or the domain of values. The reasons for separating the terms "set" and "domain" will become clear in section 4.4 when we impose some structure on domains.

Even modest programming languages encompass a number of distinct domains. Some of these domains, like those of expressions and statements, are clear from the syntax, but others reflect semantic choices made during the design of the language and are not nearly as obvious. Strachey [str72] suggests that a language designer start by considering the domains relevant to a language, since we cannot give the semantics of a language without knowing what the semantic objects are. Furthermore, deep semantic differences between languages can show up even at this stage [str72].

The language of prefix expressions has three syntactic domains: **Exp** the domain of *expressions* is constructed from the domain **Ide** of *identifiers* and the domain **Bop** of *binary operator symbols.*

The first step in defining the semantics of prefix expressions is a specification of the domains that the meanings of the syntactic objects will lie in. An identifier *id* will *denote* a

value belonging to an appropriate domain V of *values*. We will use functions called *states* to map identifiers to their values. The domain of states will be S.

2.2. *Operational semantics.* We will set up a procedure $val(exp,s)$ that will return a value in V, given an expression *exp* and state s. In order to define procedure $val(exp,s)$ we need a metalanguage that the procedure will be written in. For brevity, in this paper we will use a metalanguage that is simple enough that a precise definition of the metalanguage will be omitted. The metalanguage includes the operator $\ominus$, which subtracts its second argument from its first, and may be viewed as perfect subtraction. Also included is the operator $\circledast$, which multiplies its first and second arguments.

The procedure $val(exp,s)$ can be written as follows:

$$
\begin{aligned}
&val(exp,s) = \\
&\quad \textit{if } exp \textit{ is } id \textit{ then } s(id) \\
&\textit{else if } exp \textit{ is } -exp_1 exp_2 \textit{ then } val(exp_1,s) \ominus val(exp_2,s) \\
&\textit{else if } exp \textit{ is } *exp_1 exp_2 \textit{ then } val(exp_1,s) \circledast val(exp_2,s)
\end{aligned}
$$

Figure 1. Operational semantics for prefix expressions.

Using $val(exp,s)$, we need both *exp* and s before we can begin to give the meaning of an expression. In Figure 1, the meaning of an expression is therefore inextricably linked with states.

2.3. *Denotational semantics.* The mapping of syntactic to semantic domains is done by functions that are called *valuations*. Valuations play a role similar to that of procedures like *val* in section 2.2.

Given an expression *exp* and a state s the valuation **ev** determines a value in V. Determination of a value proceeds in two stages: $\mathbf{ev}[\![exp]\!]$ is a function from states to values. The special brackets $[\![$ and $]\!]$ are used just to enclose syntactic objects. When the function $\mathbf{ev}[\![exp]\!]$ is applied to state s the value $(\mathbf{ev}[\![exp]\!])(s)$ is determined. We will avoid parentheses and write $\mathbf{ev}[\![exp]\!]s$ for $(\mathbf{ev}[\![exp]\!])(s)$.

Later in this prelude we will consider two different valuations, which will give two different meanings, for statements. For expressions, however, there will be just one valuation.

Since there is just one valuation **ev** for expressions, we can drop **ev** without loss of information and abbreviate $\mathbf{ev}[\![exp]\!]$ to $[\![exp]\!]$. Thus for nonterminal *exp* in the syntax, the meaning of the expression generated by *exp* will be written as $[\![exp]\!]$.

We declare (to the reader) that $[\![exp]\!](s)$ is an element of V by writing

$$[\![exp]\!](s)\colon \mathbf{V}$$

Elsewhere we can declare that s is an element of S by writing

$$s\colon \mathbf{S}$$

All uses of s will then refer to elements of S. We can distinguish between distinct states by decorating s with subscripts and/or primes.

The semantic specification in Figure 2 emphasizes the syntax directed nature of the definition of $[\![exp]\!]$. For example, associated with the syntactic rule

$$exp ::= - exp_1\ exp_2$$

is the semantic rule

$$[\![exp]\!](s) = [\![exp_1]\!](s) \ominus [\![exp_2]\!](s)$$

Syntactic Domains

id:	**Ide**	identifiers
exp:	**Exp**	expressions
–, ⋆:	**Bop**	binary operators

Semantic Domains

v:	**V**	values
s:	**S = Ide → V**	states

Semantic Rules

$[\![exp]\!](s)$: **V**

| *id*
— $s(id)$

| – exp_1 exp_2
— $[\![exp_1]\!](s) \ominus [\![exp_2]\!](s)$

| ⋆ exp_1 exp_2
— $[\![exp_1]\!](s) \oplus [\![exp_2]\!](s)$

;

Figure 2. Denotational semantics for prefix expressions. Lines beginning with "|" specify the syntactic rules for constructing prefix expressions, and are followed by lines beginning with "—" which give the corresponding semantic rules.

2.4. *Abstract syntax.* When specifying the semantics of a programming language, a distinction is made between the "concrete" syntax of the language, which specifies exactly how programs in the language can be constructed, and the "abstract" syntax on which the semantic specification is based. For the language of expressions over – and ⋆ the concrete syntax, which handles precedence of ⋆ over – correctly, might be

exp ::= *exp* – *term* | *term*

term ::= *term* ⋆ *factor* | *factor*

factor ::= (*exp*) | *id*

Since an identifier *id* is generated by the derivation

exp ⇒ *term* ⇒ *factor* ⇒ *id*

there is clearly no essential semantic difference between the strings generated by *exp*, *term*, and *factor*.

McCarthy [mcc62] advocated the use of "abstract syntax" where instead of basing the semantics of expressions on the above grammar, special procedures like *isdiff* in [mcc64] are used to decide if the expression finally generated by a nonterminal is really the difference of

two subexpressions. If *isdiff* says yes, then there are further procedures, like *subtrahend* and *minuend* in [mcc64], to extract the subexpressions whose difference is being taken. These special procedures constitute an abstract syntax of the language.

Almost all the methods base semantic specifications on an abstract syntax of the language in question. In both the Vienna definition of PL/I [luc69] and the ANSI Standard PL/I definition [ans76,mar77], the abstract syntax differs sufficiently from the concrete syntax that the conversion of concrete programs to abstract programs has to be specified in detail. In denotational specifications in the style of [mil76,sto77,ten76] the abstract syntax is fairly close to the concrete syntax: the major difference between the two is that the abstract syntax is a compact, generally ambiguous grammar for the language being defined. Alternately, denotational specifications can be based on parse trees in which nonterminals like *exp*, *term*, and *factor* which are semantically similar have been grouped together. The notation in [mss76] integrates parse trees into the metalanguage.

The syntax of statements in the sample language in this prelude is ambiguous because of the production

stm ::= *stm* *stm*

This syntax is a convenient starting point for introducing the semantics for the language. The semantics of statements in C [set78] will be based on an unambiguous grammar.

The grammar for infix expressions given above is larger than the grammar for prefix expressions given early in section 2. The semantic specification for infix expressions will be proportionately larger than Figure 2, but manageably so. We will need valuations not only for *exp*, but for *term* and *factor* as well. As with *exp*, there will be just one valuation for *term* and *factor*, so we need not explicitly identify the valuation, as in the following rules for *term*:

$[\![term]\!](s)$: **V**

| *term* * *factor*

— $[\![term]\!](s) \circledast [\![factor]\!](s)$

| *factor*

— $[\![factor]\!](s)$

:

3. An Operational Semantics

McCarthy [mcc62] suggested (see section 1.3) that "the meaning of a program is defined by its effect on the state vector." In this section we will examine some of the basic concepts of this view of operational semantics. We will not attempt to introduce the notations that have been used to give operational specifications for complete programming languages.

Recall from section 1.2 that the syntax of statements in our sample language is given by

stm ::= *id* := *exp* ;

| *stm* *stm*

| `if` (*exp*) { *stm* }

| `while` (*exp*) { *stm* }

The syntax of expressions is not of importance here. We assume that a procedure like *val* in section 2.2 yields the value *val*(*exp*,*s*) of expression *exp* relative to state *s*.

The value of an expression was found in the last section by decomposing the expression into subexpressions and first evaluating the subexpressions. A similar idea works for assignments, conditionals, and compound statements, but not for while statements.

3.1. *A statement interpreter.* Procedure *comp*(*stm*,*s*) in Figure 3 returns a final state *s'*, given a statement *stm* and a starting state *s*. As in procedure *val*(*exp*,*s*) in Figure 1, there is a case for each production in the syntax for *stm*.

comp(*stm*,*s*) =

if *stm* ***is*** *id* := *exp*; ***then***

$s[val(exp,s)/id]$

else if *stm* ***is*** stm_1 stm_2 ***then***

$comp(stm_2,comp(stm_1,s))$

else if *stm* ***is*** `if` (*exp*) { stm_1 } ***then***

if $val(exp,s)\neq 0$ ***then*** $comp(stm_1,s)$ ***else*** *s*

else if *stm* ***is*** `while` (*exp*) { stm_1 } ***then***

if $val(exp,s)\neq 0$ ***then*** $comp(stm,comp(stm_1,s))$ ***else*** *s*

Figure 3. Given a statement *stm* and a state *s*, *comp*(*stm*,*s*) is the state *s'* that is the result of starting with state *s* and executing *stm*. We specify that parameters be evaluated before a recursive call occurs, so $comp(stm_2,comp(stm_1,s))$ is evaluated by first evaluating $comp(stm_1,s) = s'$ and then evaluating $comp(stm_2,s')$. Keywords like ***then*** in the metalanguage are in a different font from keywords like `while` in the programming language.

If *stm* is the assignment *id* := *exp*; then the state *s* is modified to *s'*, where *s'*(*id*) will be *val*(*exp*,*s*). All other identifiers are mapped to the same value by both *s* and *s'*. More precisely,

$$s'(x) = \textbf{if } x=id \textbf{ then } val(exp,s) \textbf{ else } s(x)$$

The definition of a new function like *s'* from an old one like *s* takes place often enough that we will introduce some notation for it. We will write $f[a/y]$ for the function *f'* defined below. (The use of λ notation is explained in a footnote[1].)

$$f' = \lambda x.\ \textbf{if } x=y \textbf{ then } a \textbf{ else } f(x)$$

We therefore have, $s' = s[val(exp,s)/id]$.

If *stm* is the compound statement stm_1 stm_2, then *comp*(*stm*,*s*) is evaluated by first evaluating $s' = comp(stm_1,s)$ and then evaluating $comp(stm_2,s')$. Alternately, $comp(stm_1\ stm_2,s))$ is given by $comp(stm_2,comp(stm_1,s))$ and the understanding that the inner $comp(stm_1,s)$ evaluated first.

If *stm* is the conditional `if` (*exp*) { stm_1 } then the state *comp*(*stm*,*s*) depends on *val*(*exp*,*s*). If *val*(*exp*,*s*) is nonzero (which corresponds to true in C [ker78]) then *comp*(*stm*,*s*) is $comp(stm_1,s)$; otherwise it is *s*.

[1]Ordinary notation does not permit us to talk about functions like *s'* and *f'* without dragging in their arguments. Since we wish to talk about functions independently of their arguments, it will be convenient to use the following alternate notation to specify the function *s'*, where the argument of *s'* is listed on the right hand side and preceded by a "λ"

$$s' = \lambda x.\ \textbf{if } x=id \textbf{ then } val(exp,s) \textbf{ else } s(x)$$

While this notation is motivated by the λ-calculus [chu41,cur58] we will use it purely as a device for defining functions.

If *stm* is `while ( exp ) { stm1 }` then *comp*(*stm*,*s*) is the same as it would be if *stm* were

`if ( exp ) {stm1 while ( exp ) { stm1 }}`

3.2. *Remark on the metalanguage.* The operational semantics of the sample language is given using the procedures *comp* in Figure 3 and *val* in Figure 1. By stating that "nested" or "innermost" calls are done first while evaluating *comp*(*stm*,*s*), we have indicated an unambiguous evaluation. Evaluation of recursive definitions, of which *comp* and *val* are examples, is discussed in section 4.1.

3.3. *Alternate notation.* Figure 4 shows what the recursive definition in Figure 3 looks like if it is expressed using the notation associated with denotational semantics. The denotational specification in section 5 will differ from Figure 4 *only* in the semantic rule for `while` statements.

Domains

id:	**Ide**	identifiers
exp:	**Exp**	expressions
stm:	**Stm**	statements
−, ∗:	**Bop**	binary operators
v:	**V**	values
s:	**S = Ide → V**	states

Semantic Rules

$[\![stm]\!](s)$: **S**

| *id* := *exp*;
— $s[v/id]$ *where* $v = [\![exp]\!](s)$

| stm_1 stm_2
— $[\![stm_2]\!]([\![stm_1]\!](s))$

| `if ( exp ) { stm1 }`
— *if* $[\![exp]\!](s) \neq 0$ *then* $[\![stm_1]\!](s)$ *else* s

| `while ( exp ) { stm1 }`
— *if* $[\![exp]\!](s) \neq 0$ *then* $[\![stm]\!]([\![stm_1]\!](s))$ *else* s

Figure 4. The recursive definition of the statement interpreter in Figure 3, expressed in the notation we have been using for denotational semantics. Note that $[\![stm]\!]$ appears on the right hand side in the semantic rule for the `while` statement. The semantics of expressions are as in section 2. Recall that a nonzero expression value corresponds to *true* in C [ker78].

For clarity, large expressions have been partitioned into subsidiary expressions using the *where* and *let-in* constructions. For example, s_1, s_2, and s_3 below are all equivalent.

$s_1 = s[\,[\![exp]\!](s)\,/id]$

$s_2 = s[v/id]$ *where* $v = [\![exp]\!](s)$

$s_3 =$ *let* $v = [\![exp]\!](s)$
in $s[v/id]$

The ***where*** construction will be used only when both parts of the construction fit on one line. The ***let-in*** construction will be used either when there is more than one subsidiary expression, or when the construction spans more than one line.

4. Recursive Definitions

Recursive definitions have figured prominently in the semantic specifications in sections 2 and 3. In this section we look more closely at recursive definitions and examine how we can base computation sequences on them and how we can use them to define functions.

A *recursive definition* of $F(x_1, \ldots, x_n)$ will be written as

$$F(x_1, \ldots, x_n) = T$$

where T is a term constructed using known functions like +, ***if-then-else***, ***is***; the unknown function represented by F; the formal parameters $x_1, \ldots, x_n$; and constants. F will be referred to as the *unknown function symbol.*

For simplicity, the treatment in this section will be informal and will use simple examples like

$$F(x,y) = \textit{if } x=0 \textit{ then } y \textit{ else } F(x-1,y+1) \tag{4.1}$$

The discussion carries over to the case when two or more unknown functions are defined simultaneously. (Recall that *comp* in section 3 was defined using *val* from section 2.)

In section 4.1 we show how computation sequences can be based on recursive definitions. For some recursive definitions, like that of *comp* in Figure 3, more than one computation sequence is possible. An example in section 4.2 shows that the values computed by different computation sequences need not always agree. This example suggests that we seek alternate methods for associating meaning with recursive definitions. Fixed points of recursive definitions will be introduced in section 4.3. In order to talk about fixed points we need to be precise about what an undefined value is: this need for precision motivates the discussion of domains in section 4.4.

Familiarity with sections 4.3-5 will help greatly in understanding denotational semantics in section 5.

4.1. *Computation rules.* Reflection on the recursive definition (4.1) suggests that $F(x,y)$ will be $x+y$ if x is nonnegative. Starting with $F(5,2)$ we will show how 7 might be computed. Later in this section we will define "computation sequences" in terms of "expansion" and "simplification" of terms.

Given a term like $F(5,2)$ we will be allowed to *expand* $F(5,2)$ using the definition (4.1). Expansion is performed by textually substituting the actual parameters 5 and 2 for x and y, to give

$$\textit{if } 5=0 \textit{ then } 2 \textit{ else } F(5-1,2+1) \tag{4.2}$$

After expansion, we will be allowed to *simplify* the intermediate term (4.2) as follows. Any subterm like $5-1$ or $5=0$, which does not contain unknown function symbols, can be replaced by the appropriate constant. Thus $5-1$ simplifies to 4 and $5=0$ simplifies to ***false***. Furthermore, *if* p *then* t_1 *else* t_2 simplifies to t_1 if p is ***true*** and to t_2 if p is ***false***. Simplification of (4.2) yields

$$F(4,3) \tag{4.3}$$

The term $F(4,3)$ above can be expanded to yield

$$\textit{if}\ 4=0\ \textit{then}\ 3\ \textit{else}\ F(4-1,3+1)$$

which simplifies to

$$F(3,4) \tag{4.4}$$

Four expansion and simplification steps later we will get 7.

The procedures *val*(*exp*,*s*) in Figure 1 and *comp*(*stm*,*s*) in Figure 3 are just larger examples of recursive definitions. These recursive definitions can also be evaluated using alternate expansion and simplification steps.

Given a recursive definition like (4.1) and a starting term t_0 like $F(5,2)$ a *computation sequence* is a sequence of terms $t_0, t_1, t_2, \cdots$, where t_{i+1} is determined from t_i by (i) expanding some subterm in t_i, and then (ii) simplifying as far as possible to yield t_{i+1}.

Coming up in section 4.2 is an example of a recursive definition and a starting term for which the computation sequence is not unique. Such a situation occurs if at some expansion step, there is a choice of subterms to expand. This point must be kept in mind while reading the following definition of "computation rule".

Given a recursive definition and a starting term t_0, a (deterministic) *computation rule* picks a unique computation sequence $t_0, t_1, t_2, \cdots$.

In addition to supplying recursive definitions like *comp* in Figure 3 and *val* in Figure 1, an operational specification must supply a computation rule: otherwise, the operational specification may not be deterministic.

4.2. *Comparing computation rules.* As the next example due to Morris [mos68] shows, different computation rules may sometimes lead to different results. Consider the recursive definition

$$M(x,y) = \textit{if}\ x=0\ \textit{then}\ 1\ \textit{else}\ M(x-1,M(x-y,y))$$

When started with $M(1,0)$ we get the following computation sequence if we expand the "innermost" subterm at every stage.

$$M(1,0) \Rightarrow M(0,M(1,0)) \Rightarrow M(0,M(0,M(1,0))) \Rightarrow \cdots$$

Since this sequence is infinite, the value computed by this sequence is undefined.

If instead of expanding the "innermost" subterm at each stage, we expand the "outermost" subterm, then we get a sequence that computes 1.

$$M(1,0) \Rightarrow M(0,M(1,0)) \Rightarrow 1$$

Starting with *stm* and *s*, depending on the computation rule, the code for procedure *comp* in Figure 3 may yield quite different computation sequences. (The operational semantics of section 3 chose one particular sequence as giving the "meaning" of *stm*.) The existence of distinct computation sequences does raise some questions about the relation between these sequences. These questions are best studied by taking a more functional approach and considering the "fixed points" of a recursive definition.

4.3. *Fixed points.* The recursive definition (4.1) of $F(x,y)$ was used to determine computation sequences in section 4.1. In this section we will take a different view of the same definition. For convenience, the definition is repeated below.

$$F(x,y) = \textit{if}\ x=0\ \textit{then}\ y\ \textit{else}\ F(x-1,y+1) \tag{4.1}$$

In the above definition we can view F as a variable representing a function, just like x is

viewed as a variable representing a number in the following equation.

$$x+6 = (x+1)^3-x^2$$

It is easy to verify that $x=1$ satisfies the above equation.

Returning to (4.1), let us view F as a variable representing a function. If we set F to the function

$$f = \lambda x,y.\ x+y$$

then we find that f satisfies (4.1). In other words if we substitute $x+y$ for $F(x,y)$ and $x-1+y+1 = x+y$ for $F(x-1,y+1)$ in (4.1), then the two sides of the equality remain equal, since for all x and y, $x+y = y$ when $x=0$.

Since the function f satisfies the recursive definition (4.1) we will call f a *fixed point* of the recursive definition.[2]

Once we start looking for fixed points of recursive definitions, we find an abundance of them. Another fixed point of (4.1) is the function g given by

$$g = \lambda x,y.\ \textit{if}\ x<0\ \textit{then}\ 0\ \textit{else}\ x+y$$

While both f and g are fixed points of (4.1), functions f and g are different since $f(-1,5)=4$, but $g(-1,5)=0$.

Changing our viewpoint momentarily, if the recursive definition of F is taken to be a scheme for *computing* $F(x,y)$, then it is clear that for negative values of x, the idea of decrementing x, incrementing y, and repeating the process, leads to a nonterminating computation.

Let us use the symbol "$\perp$" (read bottom) for the result of such a nonterminating computation. Using the symbol $\perp$ we can express functions like the following:

$$h = \lambda x,y.\ \textit{if}\ x<0\ \textit{then}\ \perp\ \textit{else}\ x+y$$

Again, it can be verified that h is a fixed point of (4.1). Not only is h a fixed point of (4.1), but, informally speaking, h corresponds most closely to the computation of $F(x,y)$, since $h(x,y)$ is $\perp$ exactly when the computation of $F(x,y)$ does not terminate, and $h(x,y)$ and $F(x,y)$ both yield $x+y$ otherwise.

The relationship between h and the other fixed points of (4.1) will be discussed further in section 4.5, after $\perp$ is discussed in section 4.4.

4.4. *Domains.* If a computation does not terminate, we have no information about the computed value. Loosely speaking, the computed value is "undefined".

The notion of "undefined value" can be formalized by including a special value, written "$\perp$", as one of the elements of a set **B** of basic values. Without a special value like $\perp$ it is difficult to distinguish the notion of "undefined value" from the following quite distinct notions: uninitialized values; and the results of operations like 1/0 (which is really an error).

The value, $\perp$, has somewhat different properties from the remaining elements of **B**. The intuitive relationship between $\perp$ and other values in **B** is suggested by Figure 5.

---

[2] The term "fixed point" is generally used in the following sense. Let ϕ be a function from some domain **D** to **D**. An element x in **D** is a *fixed point* of ϕ if and only if $x = \phi(x)$. The function f is actually a fixed point of the function

$$\lambda F.\ \lambda x,y.\ \textit{if}\ x=0\ \textit{then}\ y\ \textit{else}\ F(x-1,y+1)$$

When the meaning of `while` statements is specified in section 5 we will define it as the least fixed point of a suitable function.

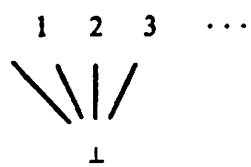

Figure 5. A value in **B** is either ⊥ or is some integer.

In section 4.3, we used ⊥ to express the result of a nonterminating computation. Another example of its use follows. Consider the declaration

```
char a[];
```

which might appear in a C [ker78] "function", where a is a formal parameter of the C "function". Here a is declared to be an array of characters, but we have no information about the number of elements in the array. We will therefore say that the number of elements in the array is ⊥. When an actual parameter is supplied, then the number of elements in the array will be known, and will be some integer.

In the denotational semantics that we will consider, a *domain* will be a pair $(\mathbf{D},\sqsubseteq)$, where **D** is a set, and $\sqsubseteq$ is a partial order on **D**. (There are other conditions that a domain must satisfy, but these will be reserved until the mathematical details are discussed in appendix A.) The partial order $\sqsubseteq$ on the domains we will consider is a formalization of the notion "less defined than". Each domain will also have a least element under $\sqsubseteq$ which will be written as ⊥.

In discussions we will sometimes refer to ⊥ as the undefined element of the domain under discussion. For the domain **B** in Figure 5, all elements other than ⊥ will be referred to as the "defined" elements of **B**.[3]

4.5. *Least fixed points.* The difference between h and the functions f and g is that $h(x,y)$ is ⊥ for all negative values of x, but neither $f(x,y)$ nor $g(x,y)$ is ⊥ for negative values of x. The important point to note is that when $h(x,y)$ is defined, then $h(x,y)$, $f(x,y)$, and $g(x,y)$ are all equal. Thus h is less defined than f or g. In fact, h is the *least* defined fixed point, in the sense that for any other fixed point f^*, of (4.1), when $h(x,y)$ is defined, $h(x,y)$ and $f^*(x,y)$ give the same defined value.

In order to read the body of the paper, all the reader needs to accept is that for each recursive definition, there exists a unique least defined fixed point. In the sequel we will follow common terminology, and refer to the least defined fixed point as the least fixed point.

Least fixed points are interesting because they, of all fixed points, correspond most closely to the "function computed" by a recursive definition.

[3] The domain **B** in Figure 5 has a very simple structure: the element ⊥ is less defined than all the other equally defined elements. Now consider the domain of all functions from **B** to **B**, written **B**→**B**. Assuming **B** has an infinite number of elements, there will be an infinite number of "levels of definedness" for elements of **B**→**B**. After all we can have a sequence of functions $f_0, f_1, f_2, \cdots$, where $f_i(x)$ is ⊥ for all but i elements of **B**, and when $f_i(x)$ is not ⊥ then $f_i(x)=f_{i+1}(x)$. In this case we have $f_i \sqsubseteq f_{i+1}$ but $f_i \neq f_{i+1}$. Such sequences of functions will be encountered in section 5 when denotational semantics for `while` statements are discussed.

5. Denotational Semantics

Concluding section 3, Figure 4 expressed the recursive definition of the statement interpreter in the notation we have been using for denotational semantics. Moving from operational to denotational semantics requires a change in viewpoint: from specifying computations to specifying functions.

This change in viewpoint can be appreciated by examining the treatment of `while` statements.

5.1. *While statements.* In both operational and denotational semantics, the meaning of a while statement is given by setting up a recursive definition motivated by the following program fragment:

`if (` *exp* `) {` stm_1 `while (` *exp* `)` stm_1 `}`

Let us suppose that *stm* is `while (` *exp* `)` stm_1. From the recursive definition in Figure 4:

$$[\![stm]\!](s) = \textit{if } [\![exp]\!](s) \neq 0 \textit{ then } [\![stm]\!]([\![stm_1]\!](s)) \textit{ else } s$$

The above equality will hold in denotational semantics as well.

For clarity, let us rewrite $[\![stm]\!]$ as F and switch the *then* and *else* parts, to get the following recursive definition.

$$F(s) = \textit{if } [\![exp]\!](s) = 0 \textit{ then } s \textit{ else } F([\![stm_1]\!](s)) \tag{5.1}$$

In denotational semantics, the function $[\![stm]\!]$ will be the least fixed point of the recursive definition (5.1). In order to give the reader some feeling for the least fixed point of (5.1), we will construct a computation sequence (see section 4.1) using (5.1). In a sense to be made precise, the least fixed point of (5.1) will be exactly the function computed by this computation sequence.

Explicitly computing a function. The first three terms in a computation sequence for (5.1) are:

$$t_0 = F(s)$$

$$t_1 = \textit{if } [\![exp]\!](s) = 0 \textit{ then } s \textit{ else } F([\![stm_1]\!](s))$$

$$t_2 = \textit{if } [\![exp]\!](s) = 0 \textit{ then } s \textit{ else if } [\![exp]\!]([\![stm_1]\!](s)) = 0 \textit{ then } [\![stm_1]\!](s) \textit{ else } F([\![stm_1]\!]([\![stm_1]\!](s)))$$

Informally, the terms t_0, t_1, t_2 correspond to stages in the process of expanding the while statement, or loop. We start out with t_0 corresponding to the unexpanded while loop: t_1 corresponds to the loop expanded once; t_2 corresponds to the loop expanded two times; and so on for subsequent terms. Since there is no a priori bound on the number of times a while statement will be executed, if we want to capture the meaning of a while statement by loop expansion, this process will have to produce an infinite sequence of terms $t_0, t_1, t_2, \cdots$. The fully expanded loop will be the "limit" of the infinite sequence of terms.

Since we will give the semantics of statements using functions rather than using the sequence $t_0, t_1, t_2, \cdots$ directly, we will construct a related sequence of functions $f_0, f_1, f_2, \cdots$

from $t_0, t_1, t_2, \cdots$.[4]

$$f_0 = \lambda s.\ \perp$$

$$f_1 = \lambda s.\ \textit{if } [\![exp]\!](s) = 0 \textit{ then } s \textit{ else } \perp$$

$$f_2 = \lambda s.\ \textit{if } [\![exp]\!](s) = 0 \textit{ then } s \textit{ else if } [\![exp]\!]([\![stm_1]\!](s)) = 0 \textit{ then } [\![stm_1]\!](s) \textit{ else } \perp$$

In general, for $i \geq 0$,

$$f_{i+1} = \lambda s.\ \textit{if } [\![exp]\!](s) = 0 \textit{ then } s \textit{ else } f_i([\![stm]\!](s))$$

Just as the fully expanded while loop corresponds to the "limit" of the sequence of terms $t_0, t_1, t_2, \cdots$, the function $[\![stm]\!]$ for the while loop will be the "limit" of the sequence of functions $f_0, f_1, f_2, \cdots$.

Given any state s, $f_0(s)$ will be $\perp$. However, if $[\![exp]\!](s)=0$ then $f_1(s)$ will be s itself.

The sequence of functions $f_0, f_1, f_2, \cdots$ has a very useful property. For $i<j$, $f_i(s)$ and $f_j(s)$ can never be incomparable states. More precisely, if $f_0, f_1, f_2, \cdots$ are elements of some domain $(D, \sqsubseteq)$ then it can be shown that

$$f_i \sqsubseteq f_{i+1} \qquad i \geq 0$$

Given such a sequence of functions $f_0, f_1, f_2, \cdots$ our informal notion of "limit" of the sequence can be made precise by taking the "least upper bound" of the set $\{f_0, f_1, f_2, \cdots\}$, written $\sqcup \{f_0, f_1, f_2, \cdots\}$. A precise treatment will be given in appendix A.

For our present purposes it suffices to say that we will write

$$f = \sqcup \{f_0, f_1, f_2, \cdots\} \tag{5.2}$$

to capture the informal notion of "limit" of the sequence $f_0, f_1, f_2, \cdots$.

The function f in (5.2) has an additional property which allows the computation sequence of terms $t_0, t_1, t_2, \cdots$ to be related to denotational semantics. The function f can be shown to be the least defined solution of the recursive definition (5.1).

Implicitly defining a function. There is a more direct way of constructing the sequence of functions $f_0, f_1, f_2, \cdots$ than the one used above. If we start with f_0, the function τ below is such that $f_1 = \tau(f_0)$: in general, for all $i \geq 0$, $f_{i+1} = \tau(f_i)$.

$$\tau = \lambda F.\ \lambda s.\ \textit{if } [\![exp]\!](s) = 0 \textit{ then } s \textit{ else } F([\![stm_1]\!](s))$$

As stated in a footnote in section 4, the term "fixed point" is generally used in the following sense. Let ϕ be a function from some domain D to D. An element x in D is a *fixed point* of ϕ if and only if $x = \phi(x)$.

There are a number of functions that are fixed points of τ. As stated in section 4.4, each domain D has an associated partial order $\sqsubseteq$ which formalizes the intuitive notion of "less

[4] The connection between the terms t_i and the functions f_i can be formalized by setting up a suitable interpretation. This interpretation replaces the unknown function symbol F by the function $\perp_F = \lambda s.\ \perp$ which is undefined everywhere.

defined than". There is a similar partial order $\sqsubset'$ on the domain that the arguments of, and fixed points of, τ are drawn from. A result in appendix A guarantees that among the fixed points of τ there is unique least defined fixed point under $\sqsubset'$.

We will write

$$\mathit{fix}\,(\tau)$$

for the least defined fixed point of τ. It will sometimes be convenient to drop parentheses and just write $\mathit{fix}\,\tau$ for $\mathit{fix}\,(\tau)$.

It is the least fixed point of

$$\tau = \lambda F.\ \lambda s.\ \mathit{if}\ [\![exp]\!](s) = 0\ \mathit{then}\ s\ \mathit{else}\ F([\![stm_1]\!](s))$$

that we choose to be the meaning of the `while` statement. In other words, when *stm* is `while ( exp )` stm_1, then

$$[\![stm]\!] = \mathit{fix}\,\lambda F.\ \lambda s.\ \mathit{if}\ [\![exp]\!](s)=0\ \mathit{then}\ s\ \mathit{else}\ F([\![stm_1]\!](s)) \qquad (5.4)$$

5.2. *A connection.* The function f in equality (5.2) was constructed by taking the "limit" of a sequence of functions. On the other hand, no explicit construction was given for $[\![stm]\!]$ in (5.4). A natural question to ask is whether $[\![stm]\!]$ can even be computed.

A result in appendix A shows that f in (5.2) and $[\![stm]\!]$ in (5.4) are in fact exactly the same function!

5.3. *Denotational semantics.* Rather than repeat most of Figure 4 we will just indicate the change that needs to be made to the figure to construct a denotational specification. Replace the lines for `while` statements in Figure 4 by the lines in Figure 6.

$[\![stm]\!](s)$: S

...

| `while ( exp ) { stm1 }`

— *let* $f = \mathit{fix}\,\lambda F.$

$\lambda s'.\ \mathit{if}\ [\![exp]\!](s')\neq 0\ \mathit{then}\ F([\![stm_1]\!](s'))\ \mathit{else}\ s'$

in $f(s)$

Figure 6. If the lines for `while` statements in Figure 4 are replaced by the above text, then we will get a denotational specification.

6. Continuations

All denotational specifications of realistic programming languages use "continuations", which are a device for handling `break`, `goto` or related statements that result in transfer of control. Since `goto` statements can occur within C programs, we will use continuations in giving the semantics of statements in C [set78]. While the approach in this section will be denotational, continuations can also be used in operational semantics. In fact the first published use of continuations as we will use them was in a survey of operational semantics [rey72].

So far we have used the *direct* approach of taking the meaning of a statement to be a function from states to states. The direct approach cannot handle `goto` statements because they disrupt the connection between control flow and the static text of a program. The problem can be traced to the direct semantics for compound statements.

In this section we will need to talk about both direct and continuation semantics, so we need to differentiate between these two kinds of meanings of statements. We will therefore explicitly identify valuations. Valuation **ss** will map a statement *stm* to a function **ss**⟦*stm*⟧ from states to states. In **ss** the first **s** is from statement, and the second **s** is from state. Another valuation **sc** will be introduced in a moment. It will be convenient to drop parentheses and write for example **ss**⟦*stm*⟧*s* for **ss**⟦*stm*⟧(*s*).

Based on the semantics of compound statements in Figure 4, we can write

$$\mathbf{ss}[\![stm_1\ stm_2]\!]s = \mathbf{ss}[\![stm_2]\!](\mathbf{ss}[\![stm_1]\!]s) \tag{6.1}$$

The problem with equation (6.1) is that it expresses the case where stm_2 is *always* executed after stm_1, since the function **ss**⟦stm_2⟧ will always be applied to the state **ss**⟦stm_1⟧s. If it is possible for stm_1 to be a `goto`, then equation (6.1) fails, since rather than applying **ss**⟦stm_2⟧ to **ss**⟦stm_1⟧s, we need to apply the function appropriate to the place where "control" goes after stm_1.

This problem is resolved by using continuations [abd76,mor70,stw74].

6.1. *An example.* The use of continuations will be illustrated by considering the following program fragment in the C language [ker78]. Note that = and not := is the assignment symbol in C. This discussion is for illustrative purposes only, and is based on a very simple subset of C in which there are no side-effects.

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

We will indicate why the statement `return (next)`, which can never be reached, does not affect the continuation semantics of the above program fragment.

Procedures in C are called "functions" since they return values and may be called within expressions. We will refer to such procedures as C-functions. `square` in the program fragment above is therefore a C-function. Statements in C must occur within the body of a C-function definition, and it is not possible to jump out of a C-function.

The value v returned by `square` depends on the statements in `square`. From these statements we will construct a function f to determine the value v returned by `square`. When f is applied to the state s with which the statements in `square` are reached, we will get

$$f(s) = v$$

A function like f which maps a state to the "answer" of a C-function will be called a (*statement*) *continuation.*

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.

A parse tree for the statements in `square` is shown in Figure 7. The continuation f for these statements will be determined during a post-order traversal of the parse tree. We start the traversal with a starting continuation c_0 which corresponds to a C-function containing just the null statement: for all states s,

$$c_0(s) = grb$$

When a `return` is executed, control returns to the caller of `square`, regardless of the statements that follow the `return`. Moreover, the value returned to the caller is the value of

the expression following the keyword `return`.

Continuation c_1 will therefore be such that for all states s,

$c_1(s) = v$ *where* $v = s(\texttt{next})$

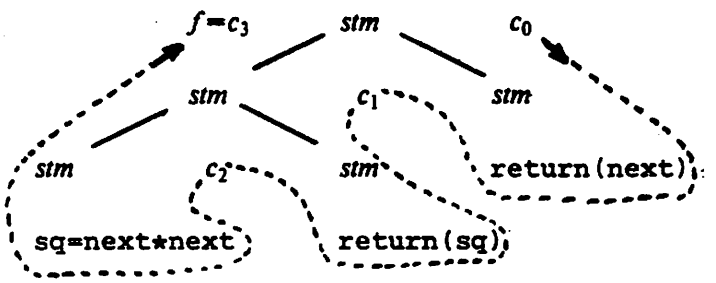

Figure 7. The continuation for the entire tree will be determined during a post-order traversal of the tree.

The interesting case is `return (sq)`, which is the next statement to be traversed. Since control returns to the caller when `return (sq)` is reached, regardless of the statements following `return (sq)`, we want continuation c_2 to be such that

$c_2(s) = v$ *where* $v = s(\texttt{sq})$

Note that c_2 does not depend in any way on c_1. Thus `return (next)`, which can never be reached, cannot affect the continuation f for the whole tree in Figure 7.

Let *st* be any of the statements in Figure 7. Suppose we know the continuation c_{st} for the statements to the right of *st* and the state s_{st} that *st* is reached with. With c_{st} and s_{st}, we can predict the "answer" of the C-function. (Note that `return (next)` can never be reached, so the fact that we may possibly predict an incorrect "answer" at this statement does not matter.)

6.2. *Continuation semantics.* For the remainder of this section we revert back to the simple language of while statements introduced in section 1.2. The treatment in this subsection is based on [sta78].

Suppose we have a sequence of statements

$stm_1 \;\; stm_2 \;\; \cdots \;\; stm_k$

Starting with state s_s before stm_1 is "executed", suppose s_f is the state after all the statements are "executed". We assume that a continuation c_{init} is given, where c_{init} will map the final state s_f to the "answer" of the program.

The objective is to determine c_{all} which takes into account the effect of all the statements, and maps the starting state s_s to the "answer".

Figure 8 shows one stage in the construction of a continuation for a sequence of statements. Suppose we are given continuation c, corresponding to the computation following the execution of *stm*. The continuation c' then corresponds to prefixing the execution of *stm* to this computation.

Associated with a statement *stm* is a function $\mathbf{sc}[\![stm]\!]$ that maps a continuation like c in Figure 8 to its corresponding c'. The valuation sc gives continuation semantics for our simple language of while statements. In the remainder of this section we will consider each production in the syntax of statements and will show how $\mathbf{sc}[\![stm]\!]$ can be specified in each case.

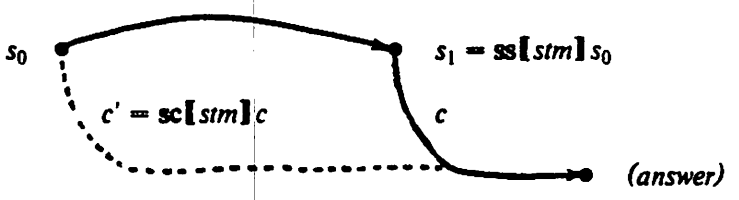

Figure 8. A *continuation* is a function from the current state to the answer.

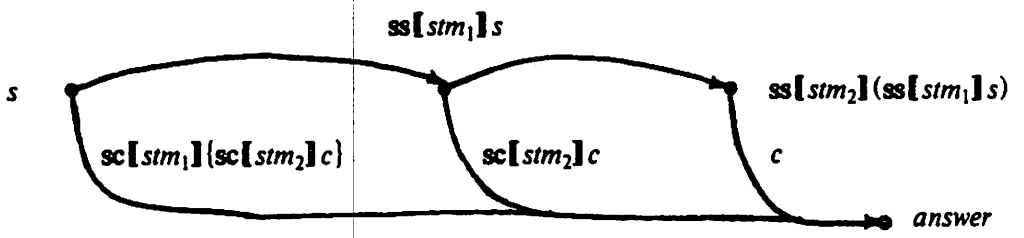

Figure 9. Semantics of $stm_1\ stm_2$ in terms of the semantics of stm_1 and stm_2. Note the order in which stm_1 and stm_2 appear in $\mathbf{ss}[\![stm_2]\!](\mathbf{ss}[\![stm_1]\!]s)$ and $\mathbf{sc}[\![stm_1]\!]\{\mathbf{sc}[\![stm_2]\!]c\}$. Whenever convenient, we will use braces "{" and "}" to enclose continuations.

Compound statements. Continuation semantics for the sequence $stm_1\ stm_2$ is suggested by Figure 9:

$$\mathbf{sc}[\![stm_1\ stm_2]\!]c = \mathbf{sc}[\![stm_1]\!]\{\mathbf{sc}[\![stm_2]\!]c\} \tag{6.2}$$

Assignment statements. Let *stm* in Figure 8 be the statement $id := exp;$. Given continuation c, we need to determine $c' = \mathbf{sc}[\![stm]\!]c$. From Figure 8 we expect the answer $c'(s_0)$ to be given by:

determine s_1 from s_0 as in direct semantics;
then apply c to s_1 to give the answer;
In the direct semantics in Figure 4, s_1 is such that

$$s_1 = s_0[v/id] \quad \textit{where} \quad v = [\![exp]\!](s_0)$$

Therefore,

$$c'(s_0) = \textit{let}\ \ v = [\![exp]\!](s_0);$$
$$s_1 = s_0[v/id];$$
$$\textit{in}\ \ c(s_1)$$

Conditional statements. Let *stm* in Figure 8 be the statement `if ( exp ) { stm1 }`.

Given any state s_0, if *exp* has a zero value (***false*** in C) then the conditional *stm* does not change the state s_0. In this case $s_1 = s_0$, so the answer will be $c(s_0)$.

On the other hand, if *exp* has a nonzero value (*true* in C) then execution of the conditional has the same effect as execution of stm_1. So if

$$c'' = \mathbf{sc}[\![stm_1]\!]c$$

the answer in this case is given by $c''(s_0)$.

Summarizing,

$$c'(s_0) = \textit{let}\ \ c'' = \mathbf{sc}[\![stm_1]\!]c \quad \textit{in}\ \ \textit{if}\ [\![exp]\!](s_0) \neq 0\ \textit{then}\ c''(s_0)\ \textit{else}\ c(s_0)$$

While statements. Let *stm* in Figure 8 be the statement `while ( exp ) { stm1 }`. Once again, we expect *stm* to have the same meaning as

`if ( exp ) { stm1 while ( exp ) stm1 }`

which is the same as

`if ( exp ) { stm1 stm }`

Using the semantics for conditionals and compound statements we will arrive at the semantics of while statements. The objective is to determine $c' = \mathbf{sc}[\![stm]\!]c$, which must be the same as

$$c' = \mathbf{sc}[\![\texttt{if (} exp \texttt{) \{ } stm_1\ stm \texttt{ \}}]\!]c \tag{6.3}$$

It will be clearer if we use the symbol F rather than c' in the following discussion.

From (6.3) and the semantics of conditionals

$$F(s_0) = c'(s_0) = \textit{let}\ \ c'' = \mathbf{sc}[\![stm_1\ stm]\!]c \quad \textit{in}\ \ \textit{if}\ [\![exp]\!](s_0) \neq 0\ \textit{then}\ c''(s_0)\ \textit{else}\ c(s_0) \tag{6.4}$$

From the semantics of compound statements and the fact that $F = \mathbf{sc}[\![stm]\!]c$, we can rewrite $\mathbf{sc}[\![stm_1\ stm]\!]c$ in (6.4) to give

$$F(s_0) = \textit{let}\ \ c'' = \mathbf{sc}[\![stm_1]\!](F) \quad \textit{in}\ \ \textit{if}\ [\![exp]\!](s_0) \neq 0\ \textit{then}\ c''(s_0)\ \textit{else}\ c(s_0) \tag{6.5}$$

We now have a recursive definition of F and can determine an appropriate function as in section 4.

A treatment similar to that of `while` statements in section 5.1 suggests that we define $\mathbf{sc}[\![stm]\!]c$ to be the least fixed point of the following function

$$\lambda F.\ \lambda s_0.\ \textit{let}\ \ c'' = \mathbf{sc}[\![stm_1]\!](F) \quad \textit{in}\ \ \textit{if}\ [\![exp]\!](s_0) \neq 0\ \textit{then}\ c''(s_0)\ \textit{else}\ c(s_0)$$

We leave it to the interested reader to write a continuation semantic specification in the style of Figure 4. Continuations will be discussed again in [set78].

7. Discussion

Operational and denotational methods specify the semantics of a language by writing semantic rules for each syntactic rule. The notations used for these rules are more complex than the notations for syntax, but much of this complexity is due to the fact that programming languages are compact notations for expressing subtle concepts.

The precision of operational and denotational specifications is expected to be of considerable assistance to a language implementer. Aside from serving as a reference manual, there is evidence that a specification can automatically be converted into an implementation of the

language [and76,mss78]. This implementation may be inefficient, but at least it is true to the specification.

One advantage of a denotational over an operational specification is that the meaning of each construct is specified just in terms of the meanings of the subconstructs. This property makes it possible to compile a program in the defined language into a metalanguage expression and may indeed make automatic compiler generation easier.

8. Bibliographic Notes

There is a large body of various shades of published literature on the subject of programming language semantics. A minimal number of references will be supplied below. Both Milner [min76] and Bjorner [bjo77] survey much of the material covered in this prelude.

Syntax. BNF notation, introduced by Backus in [bac59], allowed syntax to be specified in a precise, concise, and readable manner. The easy success with syntax raised expectations of a quick solution to the problem of specifying semantics. In fact, Backus wrote, "The author had hoped to complete a formal description of the set of legal [Algol] programs and of their meanings in time to present it here. Only the description of legal programs has been completed however. Therefore the formal treatment of the semantics of legal programs will be included in a subsequent paper. [bac59]" It would be grossly unfair to imply that Backus was alone in underestimating the difficulty of specifying semantics.

Lisp interpreter. The interpreter for LISP in [mcc60] is often held up as an example of a concise and readable specification of a language. This first operational specification was somewhat of an accident. McCarthy's intent in [mcc60] was to show that LISP was "neater than a Turing machine" by writing a universal LISP function *eval* and showing that *eval* "was briefer and more comprehensible than the description of a universal Turing machine [mcc78]." The interesting fact is that "S. R. Russell noticed that *eval* could serve as an interpreter for LISP, promptly hand coded it, and now we had a programming language with an interpreter. The unexpected appearance of an interpreter tended to freeze the form of the language [mcc78]."

Operational semantics. In sections 11-13 of [mcc62] (less than two pages), McCarthy suggested an operational approach to semantics. The concept of abstract syntax was introduced and a semantic function for determining the value of an expression relative to a state was given. Elsewhere, semantics were given for Micro-Algol, which allows assignments, conditionals, and goto's to labels [mcc64].

Landin's specification of Algol 60 was the first to actually construct an abstract machine on which translated programs in the defined language were interpreted. In [lan64], Landin specified a class of *applicative expressions* based on the λ-calculus, and an abstract machine called the SECD machine, which was capable of interpreting applicative expressions. Landin also showed how Algol 60 programs might be translated into an extended class of applicative expressions, and modified the SECD machine to interpret the extended applicative expressions [lan65]. Using continuations, Abdali [abd76] was able to translate Algol 60 programs into the pure λ-calculus, or the unextended class of applicative expressions.

The work of McCarthy and Landin considerably influenced the definition of PL/I at the IBM Vienna Laboratory between 1965 and 1968. The notation used to specify PL/I came to be called VDL (for Vienna Definition Language) and was described in [luc69]. An overview of storage models was given in [bek71]. A number of reviews of VDL are available [weg72,oll75,mar76]. The notation used in the more recent [ans76] operational specification of PL/I is similar in spirit to VDL and is reviewed in [mar77]. Further references and a discussion of subsequent work by the Vienna group may be found in [bjo78].

A useful starting point for studying operational semantics is the survey by Reynolds [rey72].

SEMANOL [and76,and77] is a metalanguage that has been used to write interpreters for Minimal BASIC and JOVIAL. The interesting point is that SEMANOL metaprograms can be executed: given a program in the defined language and its input, execution of the metaprogram yields the program's output. Unlike the metalanguages of most of the previous operational specifications, SEMANOL allows assignment as a basic operation.

The specification of Algol 68 [vaw75] is operational, but has little in common with the operational methods mentioned above: the specification is an elaborate string rewriting system, which as noted in [deb69] is an extension of Markov algorithms. A simple language is specified using this approach in [mar76].

Denotational semantics A historical account of the development of denotational semantics can be pieced together from the foreword by Scott in [sto77] and from Scott's Turing lecture [sco77]. The groundwork for denotational semantics was laid by Strachey in a 1964 paper [str64]. We have already mentioned that Strachey's ideas were placed on a secure mathematical foundation by Scott [sco70,sco76], and [scs71] covers the same ground as [str64] but with more precision.

The treatment in [scs71] could not cope satisfactorily with goto statements. Although glimmerings of the idea can be seen in earlier work, Morris [mor70] and Wadsworth [stw74] are generally given the credit for independently introducing continuations for handling goto statements and error exits from procedures. At about the same time Abdali [abd76] independently discovered continuations.

A good starting point for studying denotational semantics is the text by Stoy [sto77]. Once the reader is familiar with [scs71] the tutorial by Tennent [ten76] shows how the concepts can be applied to construct semantic specifications for complete programming languages. Milne and Strachey [mil76] is an advanced reference. Mosses [mss76] has constructed a useful program called SIS which essentially constructs an interpreter for the defined language directly from a denotational specification.

Denotational specifications have been constructed for a number of languages, including: Algol 60 [mss74,hen78]; Algol 68 [mil72]; Gedanken [ten76]; Pascal [ten77]; Snobol 4 [ten73]; Sal [mil76].

Recursive definitions. Conditional expressions were introduced by McCarthy and used to formulate recursive definitions, which were studied in [mcc63].

An example due to Morris [mos68] in section 4.2 showed that different computation rules for the same recursive definition may sometimes lead to different results. This raises the question of how computation rules might be compared. It is usual to compare computation rules relative to the least fixed point of a recursive definition, because with each computation rule we can associate the function computed by the rule. The expository paper [man73] considers some of these questions.

Based on the results of Knaster and Tarski [tar55] and Kleene [kle52] it follows that each recursive definition has a unique least fixed point (provided all functions are continuous — see appendix A). The importance of this work was recognized by Scott [sco70] who placed Landin's [lan64] and Strachey's [str64] use of fixed points on a firm mathematical basis.

Two properties of computation rules have been studied extensively.

The first is whether a computation rule computes the least fixed point of the recursive definition in question. Morris [mos68] showed that "innermost" expansion does not compute the least fixed point. Cadiou [cad72] proved that the function computed by a computation rule must be dominated by the least fixed point. Vuillemin [vui74] defined a class of "safe" computation rules, where any safe computation rule is guaranteed to compute the least fixed point (proved in [dow76]). Downey and Sethi [dow76] give necessary and sufficient conditions for deciding if a computation rule computes the least fixed point.

Since innermost evaluation corresponds to "call-by-value" in Algol 60, and innermost evaluation does not compute the least fixed point of a recursive definition, there was some concern about the properties of "call-by-value". de Bakker [deb76] shows that the function computed by "call-by-value" is also a least fixed point, but of a different recursive definition.

The second property of computation rules that has been studied is the amount of work that needs to be done in order to compute the least fixed point. See [vui74,ber79].

Recursive definitions and computation rules as we have discussed them can be applied to λ-calculus like languages, but the formalism is not very natural. The work by Berry and Levy [ber79] seems promising in this regard. The discussion in [rey72,plo75] is related to the subject of computation rules for λ-calculus like languages: the question is whether arguments must be computed before a function is applied. Using continuations, recursive definitions can be made independent of whether arguments must be computed before function application.

Conclusion. The above references for recursive definitions have been included to indicate the extent of work that is being done in the broad area of programming language semantics. This is neither the time or the place for a complete survey of the techniques for specifying semantics and their underlying mathematics. We trust that the interested reader will find a wealth of material through the references above and their bibliographies.

Acknowledgments

I am indebted to numerous colleagues at Bell Laboratories and Princeton University for comments on this and earlier drafts of this paper. The referees did a remarkably thorough job and made me much more careful in my choice of terms. Of people outside my immediate circle at Bell Laboratories, the comments and questions of R. E. Milne, A. Tang, R. D. Tennent, and an anonymous referee have helped clarify my thoughts.

Ravi Sethi

Ravi Sethi

MH-1271-RS-unix

Att.
Appendix A
References

Appendix A. Domains

A number of concepts that will be clarified here were introduced somewhat hastily in section 4. In this appendix we will start with the element $\perp$, pronounced "bottom", and explore some of the logical consequences of its introduction. $\perp$ as a value is intended to be a formalization of the notion of "don't know" or "no information". Some perspective on the definitions and results that follow will be provided by the discussion in section A.1.

The ideas in this section are due primarily to Scott [sco70].

A.1. *Discussion of domains.* The running example in this subsection uses the following recursive definition, which is motivated by the factorial function:

$$F(n) = \textit{if } n{=}0 \textit{ then } 1 \textit{ else } n{\times}F(n{-}1)$$

For the moment, values will be drawn from the set **N** of integers.

For nonnegative values of n, the computation of $F(n)$ halts with result $n!$, while for negative values of n the computation does not halt. Writing $\perp$ for the result of a nonterminating computation, the above recursive definition computes the function

$$f(n) = \textit{if } n \geqslant 0 \textit{ then } n! \textit{ else } \perp$$

Since $\perp$ can be the result of a function application, rather than being elements of **N** the set of integers, results will be elements of $\mathbf{N} \cup \{\perp\}$ which we will refer to as $\mathbf{N}_\perp$. In fact we want to allow $\perp$ as an argument as well: if $f(-1) = \perp$, then $f(f(-1))$ is $f(\perp)$, so we are forced to consider functions from $\mathbf{N}_\perp$ to $\mathbf{N}_\perp$ instead of from **N** to **N**.

The use of $\perp$ will be clarified by considering the sequence of functions

$$f_j(n) = \textit{if } 0{\leqslant}n{<}j \textit{ then } n! \textit{ else } \perp$$

The computational intuition behind this sequence of functions is as follows: if a computation of $F(n)$ is limited to making no more than j "calls" of F, then the function f_j is computed. For example if at most one call of F is allowed then we can determine that $f_1(n)$ is 1 if $n = 0$, but is $\perp$ otherwise, because we have no information about the result for all other values of n.

As the value of j increases, f_j becomes more and more like the function f, so it makes intuitive sense to view the sequence $f_0, f_1, \cdots$ as a sequence of "better and better approximations" to f.

Consider $f_1(5)$ and $f(5)$. Since $f_1(5) = \perp$, but $f(5) = 120$, if we view f_1 as an "approximation" of f, then $\perp$ must be an "approximation" of 120. By similar reasoning, $\perp$ is an "approximation" of every integer in **N**, as illustrated in Figure A.1 (which is a repetition of Figure 5 in section 4).

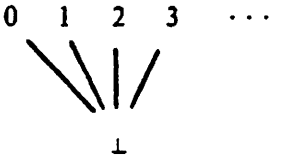

Figure A.1 The flat domain $\mathbf{N}_\perp$ with a partial order $\sqsubseteq$.

The notion of "approximation" is formalized by imposing a partial order $\sqsubseteq$ on $\mathbf{N}_\perp$, defined by

$$x \sqsubseteq y \quad \text{if and only if} \quad (x = \perp) \vee (x = y) \tag{A.1}$$

Partial orders like the one defined by (A.1) are called *flat* or *discrete* because whenever $x \sqsubseteq y \sqsubseteq z$,

either $x=y$ or $x=z$.

Once we accept the flat partial order in Figure A.1 as being natural, we are immediately led to accept richer partial orders as being natural.

A function with two arguments can be viewed as taking an ordered pair from $\mathbf{N}_\perp \times \mathbf{N}_\perp$ as an argument. The natural extension of $\sqsubseteq$ on $\mathbf{N}_\perp$ to $\sqsubseteq'$ on $\mathbf{N}_\perp \times \mathbf{N}_\perp$ is defined by

$$(u,v) \sqsubseteq' (x,y) \quad \text{if and only if} \quad (u \sqsubseteq x)\ (v \sqsubseteq y) \tag{A.2}$$

The partial order defined by (A.2) is not flat, because not only do we have $(0,\perp) \sqsubseteq' (0,1)$ and $(\perp,1) \sqsubseteq' (0,1)$, but $(0,\perp)$ and $(\perp,1)$ are not comparable under $\sqsubseteq'$.

As another example, suppose that the infinite sequence of functions f_j defined above is drawn from some domain $\mathbf{D}$. The partial order $\sqsubseteq''$ on $\mathbf{D}$ must be such that $f_0 \sqsubseteq'' f_1 \sqsubseteq'' \cdots$, which is an infinite "chain" of distinct functions.

When we define "domains" to be partial orders with a unique least element, $\perp$, in section A.2, there will be one more condition on "domains". This condition will require that "limits" exist. For example, given the infinite sequence $f_0 \sqsubseteq'' f_1 \sqsubseteq'' \cdots$ of elements in a $\mathbf{D}$ we want f, their "limit", to be in $\mathbf{D}$ as well.

A.2. *Domains.* A *chain-complete partial order* is a system $(\mathbf{D}, \sqsubseteq, \perp)$ where $\mathbf{D}$ is a set, $\sqsubseteq$ is a partial order on $\mathbf{D}$, and $\perp$ is an element of $\mathbf{D}$, subject to the axioms:

(i) ($\perp$ is a zero for $\mathbf{D}$) $\forall x \in \mathbf{D},\ \perp \sqsubseteq x$

(ii) ($\mathbf{D}$ is complete) for every linearly ordered subset (chain) $C \subseteq \mathbf{D}$, there exists a *least upper bound* $\sqcup C$ such that

$\forall x \in C,\ x \sqsubseteq \sqcup C$, and

$(\forall x \in C,\ x \sqsubseteq y)$ *implies* $(\sqcup C \sqsubseteq y)$

The term "chain-complete partial order" will be abbreviated to "cpo". All the domains we will consider will be cpo's.

Given domains $(\mathbf{D}, \sqsubseteq, \perp)$ and $(\mathbf{D}', \sqsubseteq', \perp')$ it will be convenient to drop primes while writing $\sqsubseteq'$ and $\perp'$ and let the context of $\sqsubseteq$ and $\perp$ determine which domain is involved. Furthermore, we will take $\sqsubseteq$ and $\perp$ as being understood and will simple write $\mathbf{D}$ and $\mathbf{D}'$ for the domains $(\mathbf{D}, \sqsubseteq, \perp)$ and $(\mathbf{D}', \sqsubseteq, \perp)$.

A.3. *Discussion of functions.* We have considered a number of examples of recursive definitions, the most recent being that of F in section A.1, where on some argument a, the computation of $F(a)$ does not terminate. It follows that the computation of, for example, $F(a)+5$ will not terminate either.

Since $\perp$ is an element of $\mathbf{N}_\perp$, functions like $+$ which previously applied to integers in $\mathbf{N}$ will have to be extended to permit $\perp$ as an argument as well. Our computational intuition suggests that $\perp+5$ should also be $\perp$. The function $+$ has the following "monotonicity" property.

A function g from domain $\mathbf{D}$ to $\mathbf{D}'$ is *monotonic* if for all x and y in $\mathbf{D}$, $x \sqsubseteq y$ implies $g(x) \sqsubseteq g(y)$. We require all functions to be monotonic. When $x \sqsubseteq y$, the y is "more defined" or has "more information" than x, so we expect $g(y)$ to be similarly "more defined" than $g(x)$.

Actually, we will impose a stronger condition than monotonicity on functions. This new condition, called *continuity*, becomes important when a domain $\mathbf{D}$ can have infinite chains of distinct elements under $\sqsubseteq$.

A function g from domain $\mathbf{D}$ to $\mathbf{D}'$ is *continuous* if for all chains $C \subseteq \mathbf{D}$,

$$g(\sqcup C) = \sqcup \{g(x) | x \in C\}$$

Note. Continuity implies monotonicity, so all continuous functions are monotonic.

As an example, recall the chain of functions $f_0 \sqsubseteq f_1 \sqsubseteq \cdots$ in section A.1, with least upper bound f. For any j, f_j gives a non-$\perp$ result for a finite number of arguments, but f gives a non-$\perp$ result for an infinite number of arguments. Thus f must be different from f_j, for all j. The case of interest is when some higher order function τ takes f as an argument. Since f is the least upper bound of the sequence $f_0, f_1, \ldots$, we expect f to have no more "information" than that contained in the sequence of approximations to f. Continuity of τ requires that

$$\tau(f) = \sqcup \{\tau(f_j) | j = 0,1, \cdots\}$$

since

$$f = \sqcup \{f_j | j = 0,1, \cdots\}$$

A.4. *Functions.* Only continuous functions between domains will be considered.

A.5. *Least fixed points.* With the definition of domains and the restriction to continuous functions we can prove the following theorem due to Knaster and Tarski [tar55].

Fixed point theorem. Let f be a continuous function from domain **D** to domain **D'**. There exists a unique least element of **D** under $\sqsubseteq$, such that $x = f(x)$ i.e. x is the least *fixed point* of f.

Moreover,

$$x = \bigsqcup_{i=0}^{\infty} f^i(\perp)$$

Proof. First we will show that $\sqcup_{i=0}^{\infty} f^i(\perp)$ is a fixed point of f. Here $f^i(u)$ is defined to be u if $i=0$ and to be $f(f^{i-1}(u))$ otherwise.

From the properties of $\perp$, $\perp \sqsubseteq u$, for all u in **D**, so in particular $\perp \sqsubseteq f(\perp)$. From the monotonicity of f (which is implied by the continuity of f) $f(\perp) \sqsubseteq f(f(\perp)) = f^2(\perp)$. Iterating this process we get $\perp = f^0(\perp) \sqsubseteq f^1(\perp) \sqsubseteq \cdots$.

Let $y = \sqcup_{i=0}^{\infty} f^i(\perp)$, and consider $f(y)$. From continuity of f,

$$f(y) = f\left(\bigsqcup_{i=0}^{\infty} f^i(\perp)\right) = \bigsqcup_{i=0}^{\infty} f^{i+1}(\perp) = \bigsqcup_{i=1}^{\infty} f^i(\perp) = \bigsqcup_{i=0}^{\infty} f^i(\perp) = y$$

where the second last equality is based on the observation that the first term $f^0(\perp) = \perp$ of the limit is superfluous.

Having shown that y is a fixed point of f, we now need to show that y is the unique least fixed point of f.

Suppose z is some other fixed point of f. Clearly $\perp \sqsubseteq z$, and from monotonicity of f, $f(\perp) \sqsubseteq f(z) = z$. Iterating, we get $f^i(\perp) \sqsubseteq z$. From the definition of least upper bound, it follows that

$$y = \bigsqcup_{i=0}^{\infty} f^i(\perp) \sqsubseteq z$$

Since z is an arbitrary fixed point of f, it follows that y must be the unique least fixed point of f. □

A.6. *Conclusion.* Typical of the operations that can be used to construct new domains from old are the following:

$\mathbf{D_0}+\mathbf{D_1}$	the *sum* of $\mathbf{D_0}$ and $\mathbf{D_1}$
$\mathbf{D_0}\times\mathbf{D_1}$	the *product* of $\mathbf{D_0}$ and $\mathbf{D_1}$

$D_0 \rightarrow D_1$ the domain of all continuous functions from D_0 to D_1

Just as functions can be recursively defined, domains can also be recursively defined. Scott's major contribution was demonstrating the existence of recursively defined domains, including for example domains satisfying

$$D = D \rightarrow D$$

The reader is referred to a text book like [sto77] for further study.

References

abd76 S. K. Abdali, "A lambda calculus model of programming languages: I. simple constructs; II. jumps and procedures," *Computer Languages* **1** (4) pp.287-301, 303-320 (1976).

and76 E. R. Anderson, F. C. Belz, and E. K. Blum, "Semanol (73) A metalanguage for programming the semantics of programming languages," *Acta Informatica* **6** pp.109-131 (1976).

and77 E. R. Anderson, F. C. Belz, and E. K. Blum, "Issues in the formal specification of programming languages," pp.1-30 in *Formal Descriptions of Programming Concepts*, ed. E. J. Neuhold, North-Holland, Amsterdam (1978).

ans76 ANSI X3.53, *American National Standard Programming Language PL/I*, American National Standards Institute, New York, N.Y. (1976).

bac59 J. W. Backus, "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference," pp.125-132 in *Proceedings International Conference on Information Processing, Unesco, Paris, June 1959*, Published by Unesco (Paris), R. Oldenburg (Munich), and Butterworths (London) (1960).

bek71 H. Bekic and K. Walk, "Formalization of storage properties," pp.28-61 in *Symposium on Semantics of Algorithmic Languages*, ed. E. Engeler, Lecture Notes in Mathematics 188, Springer-Verlag, Berlin (1971).

ber79 G. Berry and J.-J. Levy, "Minimal and optimal computations of recursive programs," *J. ACM* **26** (1) (January 1979).

bjo77 D. Bjorner "Programming languages: linguistics and semantics," pp.511-536 in *International Computing Symposium 1977*, ed. E. Morlet and D. Ribbens, North-Holland, Amsterdam (1977).

bjo78 D. Bjorner and C. B. Jones, (eds), *The Vienna Development Method: The Meta-Language*, Lecture Notes in Computer Science 61, Springer-Verlag, Berlin (1978).

cad72 J. M. Cadiou, "Recursive definitions of partial functions and their computations," Ph.D. Thesis, STAN-CS-266-72, Computer Science Department, Stanford University, Stanford CA (March 1972).

chu41 A. Church, *The Calculi of Lambda Conversion*, Annals of Math. Studies 6, Princeton University Press, Princeton, N.J. (1941), 2nd ed (1951).

con76 R. L. Constable and J. E. Donahue, "An elementary formal semantics for the programming language PL/CS," TR 76-271, Department of Computer Science, Cornell University (March 1976).

cur58 H. B. Curry and R. Feys, *Combinatory Logic, Volume 1*, North-Holland, Amsterdam (1958).

deb69 J. W. de Bakker, "Semantics of programming languages," pp.173-227 in *Advances in Information Systems Science, Vol. 2*, ed. J. T. Tou, Plenum Press, New York, N.Y. (1969).

deb76 J. W. de Bakker, "Least fixed points revisited," *Theoretical Computer Science* **2** pp.155-181 (1976).

don76 J. E. Donahue, *Complementary Definitions of Programming Language Semantics*, Lecture Notes in Computer Science **42** Springer-Verlag, Berlin (1976).

dow76 P. J. Downey and R. Sethi, "Correct computation rules for recursive languages," *SIAM J. Computing* **5**, (3) pp.378-401 (September 1976).

hen78 W. Henhapl and C. B. Jones, "A formal definition of Algol 60 as described in the 1975 modified report," pp.305-336 in *The Vienna Definition Method: The Meta-Language*, ed. D. Bjorner and C. B. Jones, Lecture Notes in Computer Science 61, Springer-Verlag, Berlin (1978).

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

kle52 S. C. Kleene, *Introduction to Metamathematics,* Van Nostrand, Princeton, N.J. (1952).

lan64 P. J. Landin, "The mechanical evaluation of expressions," *Computer J.* **6** (4) pp.308-320 (January 1964).

lan65 P. J. Landin, "A correspondence between Algol 60 and Church's lambda-notation," *Comm. ACM* **8** (2) pp.89-101 (February 1965) and *Comm. ACM* **8** (3) pp.158-165 (March 1965).

luc69 P. Lucas and K. Walk, "On the formal description of PL/I," *Annual Review in Automatic Programming* **6** (3) pp.105-182 (1970).

man73 Z. Manna, S. Ness, and J. Vuillemin, "Inductive methods for proving properties of programs," *Comm. ACM* **16** (8) pp.491-502 (August 1973).

mar76 M. Marcotty, H. F. Ledgard, and G. V. Bochman, "A sampler of formal definitions," *Computing Surveys* **8** (2) pp.191-276 (June 1976).

mar77 M. Marcotty and F. G. Sayward, "The definition mechanism for standard PL/I," *IEEE Trans. Software Engineering SE-3,* **6** pp.416-450 (November 1977).

mcc60 J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, Part I," *Comm. ACM* **3** (4) pp.184-195 (April 1960).

mcc62 J. McCarthy, "Towards a mathematical science of computation," pp.21-28 in *Information Processing 1962,* ed. C. M. Popplewell, North-Holland, Amsterdam (1963).

mcc63 J. McCarthy, "A basis for a mathematical theory of computation," pp.33-70 in *Computer Programming and Formal Systems,* ed. P. Braffort and D. Hirschberg, North-Holland, Amsterdam (1963).

mcc64 J. McCarthy, "A formal description of a subset of Algol," pp.1-12 in *Formal Language Description Languages for Computer Programming,* ed. T. B. Steel, Jr., North-Holland, Amsterdam (1966).

mcc78 J. McCarthy, "History of Lisp," Preprints of ACM-SIGPLAN History of Programming Languages Conference, *SIGPLAN Notices* **13**, (8) pp.217-223 (August 1978).

mil72 R. E. Milne, "The mathematical semantics of Algol 68," unpublished manuscript, Oxford University (1972).

mil76 R. E. Milne and C. Strachey, *A Theory of Programming Language Semantics, 2 Vols.,* Chapman and Hall, London, and John Wiley, New York (1976).

min76 R. Milner, "Program semantics and mechanized proof," *Mathematical Centre Tracts* **82**, pp.3-44 (1976).

mor70 F. L. Morris, "The next 700 programming language descriptions," unpublished manuscript (1970).

mos68 J. H. Morris, Jr., "Lambda calculus models of programming languages," MAC-TR-57, MIT Project MAC, Cambridge, MA. (1968).

mss74 P. D. Mosses, "The mathematical semantics of Algol 60," Technical Monograph PRG-12, Programming Research Group, Oxford University (1974).

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).

nau63 P. Naur (ed), J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger, "Revised report on the algorithmic language Algol 60," *Comm. ACM 6* (1) pp.1-17 (January 1963), *Computer J.* **5** (4) pp.349-367 (January 1963). *Numer. Math.* **4** (5) pp.420-453 (1963).

oll75 A. Ollongren, *A Definition of Programming Languages by Interpreting Automata*, Academic Press (1975).

plo75 G. D. Plotkin, "Call-by-name, call-by-value and the λ-calculus," *Theoretical Computer Science* **1** (2) pp.125-159 (1975).

rey72 J. C. Reynolds, "Definitional interpreters for higher-order programming languages," *25th ACM Annual Conference*, pp.717-740 (August 1972).

sco70 D. Scott, "Outline of a mathematical theory of computation," *Fourth Annual Princeton Conference on Information Sciences and Systems*, pp.169-176 (March 1970).

sco76 D. Scott, "Data types as lattices," *SIAM J. Computing* **5** (3) pp.522-587 (September 1976).

sco77 D. Scott, "Logic and programming languages," *Comm. ACM* **20** (9) pp.634-640 (September 1977).

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).

set78 R. Sethi, "Semantics of the C programming language, part 1: statements," unpublished manuscript, Bell Laboratories, Murray Hill, N.J. (1978).

sta78 R. Sethi and A. Tang, "Transforming direct into continuation semantics for a simple imperative language," unpublished manuscript, Bell Laboratories, Murray Hill, N.J. (1978).

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

str64 C. Strachey, "Towards a formal semantics," pp.198-220 in *Formal Language Description Languages for Computer Programming*, ed. T. B. Steel, Jr., North-Holland, Amsterdam (1966).

str72 C. Strachey, "Varieties of programming language," pp.222-233 in *International Computing Symposium Proceedings*, Cini Foundation, Venice (April 1972).

stw74 C. Strachey and C. Wadsworth, "Continuations: a mathematical semantics which can deal with full jumps," Technical Monograph PRG-11, Programming Research Group, Oxford University (1974).

tar55 A. Tarski, "A lattice-theoretical fixpoint theorem and its applications," *Pacific J. Math.* **5** (2) pp.285-309 (June 1955).

ten73 R. D. Tennent, "Mathematical semantics of Snobol 4," *ACM Symposium on Principles of Programming Languages*, pp.95-107 (October 1973).

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

ten77 R. D. Tennent, "A denotational definition of the programming language Pascal," Tech. Rep. 77-47, Department of Computing and Information Science, Queen's University, Kingston, Ontario (July 1977).

vaw75 A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker, "Revised report on the algorithmic language Algol 68," *Acta Informatica* **5** (1-3) pp.1-236 (1975).

vui74 J. Vuillemin, "Correct and optimal implementations of recursion in a simple programming language," *JCSS* **9** (3) pp.332-354 (December 1974).

weg72 P. Wegner, "The Vienna definition language," *Computing Surveys* **4** (1) pp.5-63 (March 1972).