AN ELEMENTARY FORMAL SEMANTICS
FOR THE PROGRAMMING LANGUAGE
PL/CS†.

Robert L. Constable
James E. Donahue

76-271

March 1976

Department of Computer Science
Cornell University
Ithaca, New York 14853

†This research was supported in part by the National Science Foundation under grant DCR-74-14701.
AN ELEMENTARY FORMAL SEMANTICS FOR THE PROGRAMMING LANGUAGE PL/CS†

Robert L. Constable
James E. Donahue
Cornell University

Abstract:

The PL/CS language is an instructional variant of PL/C designed to provide a simple, easy-to-understand tool to teach a disciplined style of programming (see [Conway 1976]). This report gives a complete formal semantic specification of the language, following the style of [Scott and Strachey 1972]. In keeping with the goal of simplicity in the design of PL/CS, the formal definition is presented in an hierarchical fashion and uses only elementary mathematical concepts, such as set, relation, and recursive definition.

Key Words: programming language semantics, denotational semantics, recursive functions, PL/I, PL/C, PL/CS.

†This research was supported in part by the National Science Foundation under grant DCR-74-14701
§ 1 Introduction

(1.1) The PL/CS programming language, origins and overview.

The programming language PL/CS was designed as an instructional variant of PL/C, constrained by the decision to implement it by modifying the existing PL/C compiler [Cook 1971]. Within these limits, one goal of the language design was to develop a clear and simple semantic description. This was done in part by limiting the language features to those that could be clearly defined in a formal way, as has been suggested by [Hoare 1969], [Hoare and Wirth 1973], and Dijkstra. PL/CS goes farther in the direction of restricted constructs than most real programming languages, e.g. functions do not have side effects, goto's are forward only, global variables are explicitly declared, procedure and function parameters are not allowed. The language design was also influenced by a concern for a formal axiomatic description and by an awareness of what language features are helpful in expressing correctness proofs for programs.

The syntax of PL/CS was chosen to be highly correctible and PL/I-like. Although the syntax does not concern us in this report, certain syntactic considerations have influenced the semantics. For instance, although block structure is easy to describe in the formal semantics, it causes severe error detection problems, pedagogical problems and axiomatic description problems. So some constructs have either been omitted or left for consideration in future versions of the language.
1.0 The semantic style.

The definition given below follows the dogma and style of "mathematical semantics", as exemplified by [Scott and Strachey 1972]. The major difference between the form of this definition and other definitions in the same style (e.g., Tarski 1944) is the adoption of an hierarchical structure in its presentation. We begin at the level of an abstract (or simple level) language and progressively introduce new forms of control constructs, basic statements, or declarations to build up finally to the definition of the full PL/CS language.

Moreover, this progression to more detail follows a logical course: the "next most general class of constructs" is determined by what can be specified by "the simplest" refinement of the underlying semantic model.

This method of presentation appears to be a pedagogically sound way to organize ideas about a programming language. It is the style found in most informal expositions of programming languages, but is not common in most of the more formal literature. We feel that the organization of the definition presented in this report may make formal methods less formidable to the uninitiated and permit a more rigorous approach to programming instruction.

Although this report is basically an application of existing theory, there are some technical points of interest (for example, our method of treating simple gotos without the full apparatus of continuations). As the report proceeds, we
will remark on these and on issues arising from proposed modifications of the language. We will also discuss some shortcomings of the language brought to light by our attempt to specify formally its semantics.

§ 2 Formal Semantics of PL/CS

(2.1) Organization.

As noted above, we present the semantics of the entire PL/CS language by isolating several important sublanguages and presenting, in turn, the semantics of each of these sublanguages. The order will be the following.

We begin with the language of while schemes. Essentially, while schemes include the control constructs basic to structured programming – sequencing, binary selection, and iteration. This subset has a particularly elegant semantics illustrating the basic aspects of the denotational approach.

The next subset is the PL/CS flowchart schemes, formed by extending the while schemes with the simple (forward) gotos allowed in PL/CS. The introduction of the goto complicates the semantics in a fundamental manner and causes changes in the interpretations of our earlier while schemes. As we describe below, however, the simple jumps of PL/CS do have a much more intuitive semantics than the more complicated gotos of full PL/I (or PL/C).

We next introduce procedure declaration, call, and return to form the language of PL/CS recursion schemes. To define the
semantics of recursion schemes, we are forced to give a more
detailed description of the underlying "state" manipulated by
PL/CS programs and the "meanings" of identifiers in programs.

In the next PL/CS sublanguage, we introduce the notion of
program variables, causing a further refinement of the "state"
to include a "machine store" component. In this sublanguage
we define the meanings of variable declarations and parameter-
passing to PL/CS procedures.

Finally, we complete our description of PL/CS by giving
a detailed definition of the meaning of the value-producing
expressions in the language, as well as the semantics of assignment
function call and return. The last section discusses the
meanings of the subscripted variables (arrays) and operations
(the indexed do) allowed in the language, which depend on a
further refinement of the domain of values to include the integers.
(A final section, dealing with input and output
operations will be added later.)

(2.2) While schemes.

We can introduce many of the concepts and issues in
formal semantics in terms of very simple program schemes. At
first, we will consider the following subset of PL/CS
statements: 

\[\text{\textsuperscript{1}We modify the presentation of productions in a grammar by}
\text{using a capitalized name, as Exp, to denote the class of all}
\text{expressions. We then write "exp", a non-terminal, to denote}
\text{an element of the class.}\]
Stmt = SimpleStmt
    | if Exp then Stmt else Stmt fi
    | while Exp do Stmt od
    | Stmt; Stmt
    | do Stmt end

For the present, we will leave the class of simple statements unspecified. However, we note in passing that simple statements compose the basic actions of programs in the language, including assignment, procedure call, and input/output commands.

The meaning of PL/CS while schemes is given in terms of functions on sets called domains of the language. We think of each basic statement as a function which transforms a state s into a new state s'. The collection of states is denoted S and an individual state is denoted s. This set S is one of the basic domains of the language.

Intuitively, a state will correspond to the data space with all of its named values. We will refine its definition later as required to define the semantics of PL/CS recursion schemes. At present, however, we will keep our understanding of "state" strictly informal.

The meaning of each of the statements in this language is defined in terms of a function Ms which maps each statement in the language into its associated state transformation, i.e., Ms has functionality

\[ Ms : Stmt \rightarrow [S + S] \]
The function $M_s$ is defined by presenting a clause of the definition for each syntactic category of the class of statements.

$$M_s[\text{simpleStmt}] = \text{some particular } f:S \rightarrow S \text{ associated with each simple statement}$$

$$M_s[\text{if exp then stmt}_1 \text{ else stmt}_2 \text{ fi}] (s) =$$

$$\begin{align*}
\text{if } M_e[\text{exp}] (s) \text{ then } & M_s[\text{stmt}_1] (s) \\
\text{else } & M_s[\text{stmt}_2] (s)
\end{align*}$$

where $M_e$:Exp + S + {true,false}

$$M_s[\text{while exp do stmt od}] (s) =$$

$$\begin{align*}
\text{if } M_e[\text{exp}] (s) \text{ then } & M_s[\text{while exp do stmt od}] (M_s[\text{stmt}] (s)) \\
\text{else } & s
\end{align*}$$

$$M_s[\text{stmt}_1; \text{stmt}_2] (s) =$$

$$M_s[\text{stmt}_2] (M_s[\text{stmt}_1] (s))$$

$$M_s[\text{do stmt end}] (s) = M_s[\text{stmt}] (s)$$

This semantics relies only on the reader's familiarity with function application, composition, and recursive functions over an arbitrary set $S$.

(2.3) Flowchart schemes.

This simple definition of $M_s[\ ]$ in 2.2 fails as soon as we add more flexible control as powerful as the goto. The
difficulty then is that statement composition is not function composition. This is immediately obvious for \texttt{goto l; stmt}, but is more insidious in the case of \texttt{while p do stmt\textsubscript{1} od; stmt\textsubscript{2}} in which the \texttt{stmt\textsubscript{1}} contains a \texttt{goto}.

There are methods of dealing with arbitrary \texttt{gotos} in the spirit of the mathematical semantics, in particular, the methods of \texttt{continuations} in [Strachey and Wadsworth 1974]. There are also methods that rely on transforming the program text slightly, for instance regarding all statements as procedure calls (see (2.4) of this report).

None of these methods are both simple and adequate, but in the case of forward \texttt{gotos} there is a simple definition technique. The idea is to carry along the program segment following a statement. It is into this segment that the program must continue. Because of the list structure of this segment, we can easily locate the labels to which forward \texttt{gotos} will branch.

The syntax of the extensions to the PL/CS language to allow forward \texttt{gotos} is the following:

\[
\text{Stmt} = \begin{cases} 
\text{Id:Stmt} \\
\text{goto Id}
\end{cases}
\]

The first rule allows statements to be labelled, the second provides the syntax to allow transfers to labelled statements.

To give the semantics of our extended language (the language of PL/CS flowchart schemes), we extend the meaning function
We defined above to take an additional argument, the context in which goto statements are to be evaluated. Because of the restrictions placed on goto statements in PL/CS, this context is simply the remainder of the program, or an element of the syntactic domain Stmt. Thus, our new meaning function is of type

\[ \text{Ms} : \text{Stmt} \rightarrow \text{Stmt} \times S \times S \]

where the second argument is the context used to give meaning to gotos. The clauses of our definition are given below.

\[ \text{Ms}([\text{simple\_stmt}])(\text{stmt})(s) = \text{Ms}([\text{stmt}])(\text{null})(f(s)) \]

where \( f : S \rightarrow S \) is the state transformation associated with the simple-statement and null is the empty statement list, i.e., no statements remain to be evaluated.

\[ ^{\dagger}\text{We associate type equations to the right so that this means} \]

\[ \text{Stmt} \rightarrow [\text{Stmt} \rightarrow [S \times S]]. \]
\[ Ms \left[ \text{if } \exp \text{ then } \text{stmt}_1 \text{ else } \text{stmt}_2 \text{ fi } \right] (\text{stmt})(s) = \]
\[ \quad \text{if } Ms[\exp](s) \]
\[ \quad \text{then } Ms[\text{stmt}_1](\text{stmt})(s) \]
\[ \quad \text{else } Ms[\text{stmt}_2](\text{stmt})(s) \]

\[ Ms[\text{while } \exp \text{ do } \text{stmt}_1 \text{ od }](\text{stmt}_2)(s) = \]
\[ \quad \text{if } Ms[\exp](s) \]
\[ \quad \text{then } Ms[\text{stmt}_1](\text{while } \exp \text{ do } \text{stmt}_1 \text{ od } \text{stmt}_2)(s) \]
\[ \quad \text{else } Ms[\text{stmt}_2](\text{null})(s) \]

\[ Ms[\text{stmt}_1; \text{stmt}_2](\text{stmt}_3)(s) = \]
\[ Ms[\text{stmt}_1](\text{stmt}_2; \text{stmt}_3)(s) \]

\[ Ms[\text{goto } id](\text{stmt})(s) = \]
\[ Ms[\text{LocateLabel}(id, \text{stmt})](\text{null})(s) \]

where LocateLabel: [Id \times \text{Stmt}] \to \text{Stmt} produces the remainder of the statement sequence given by its second argument beginning with the statement labelled with the label given by its first argument. Note that the label must occur on a statement in the compound statement given by the second argument (it cannot be a label inside such a statement). This means that it is not possible to jump into the bodies of iteration or conditional statements. If there is no statement in the list with this label, then the identity transformation, \text{as}s, is used.
\( M[s \text{do } \text{stmt}_1 \text{ end }] (\text{stmt}_2) (s) = \)
\( M[s \text{stmt}_1 ] (\text{stmt}_2) (s) \)

\( M[s \text{id} : \text{stmt}_1 ] (\text{stmt}_2) = M[s \text{stmt}_1 ] (\text{stmt}_2) \)

Labels have no effect on the meaning of statements. Finally,

\( M[s \text{null }] (\text{null}) (s) = s \)

If there is nothing remaining to be done, the state produced is the final result of the program.

(2.3) PL/CS recursion schemes.

The final major control constructs to be added to the PL/CS language are procedure declaration, call, and return. To give the semantics of this sublanguage, the language of PL/CS recursion schemes, we will need to make our first refinement of the intuitive, informal notion of "state" used in the previous definition. (For the moment, we will ignore the possibilities of parameter passing; this aspect of procedures will be discussed in (2.4).)

The syntax of PL/CS recursion schemes is the following. A PL/CS recursion scheme is a main procedure body followed by a finite list of procedure declarations.

\[
\text{Program} = \text{Id}:\text{MainProc} (: \text{Id}:\text{ProcDef})
\]

\[
\text{ProcDef} = \text{procedure} , \text{Stmt} \text{end}
\]

\[
\text{MainProc} = \text{procedure options} (\text{main}) ; \text{stmt} \text{end}
\]

Procedure bodies are simply statements, as defined above for
flowchart schemes, with the following additions to allow procedure call and return:

\[
\text{Stmt} = \text{call Id} \quad \text{return}
\]

To define the semantics of this sublanguage, we obviously need some way to associate identifiers with the "meaning" of declared procedures. The important aspect of procedure declaration is that, because of the static scope rules for PL/CS, the meaning associated with procedure identifiers is fixed throughout the execution of a program. This contrasts with the dynamically changing values of the program variables. Thus, we separate the "program state" used earlier into:

1. a static environment, which for the present will simply associate "procedure values" with procedure identifiers, and

2. a "store" component containing the values of the program variables. For now, we will leave further details of the store unspecified.

Thus, the new definition will use the following domains:

\[
\begin{align*}
\text{Env} &= \text{Id} \times \text{Proc} & \text{environments} \\
\text{Proc} &= S \times S & \text{procedure value} \\
S &= & \text{stores}
\end{align*}
\]

and meaning functions
\[ Ms: Stmt \rightarrow Stmt + Env \rightarrow S + S \]

\[ Mp: [Program + ProcDef] \rightarrow Env \rightarrow Proc \]

The clauses of the new meaning functions are given below.

\[ Ms [\text{simple statement}] (stmt)(e, s) = \]
\[ Ms [\text{stmt}] (return)(e, f(e, s)) \]

where \( f: Env \rightarrow S \rightarrow S \) is the state transformation associated with the basic statement. Since all PL/CS statements are parts of procedures, rather than using the \textit{null} continuation as meaning "nothing remains to be done", we use \textit{return} to signify "the end of the enclosing procedure body".
\[ \text{Ms}[\text{if exp then stmt}_1 \text{ else stmt}_2 \text{ fi}] (\text{stmt}_3) (e, s) = \]
\[ \text{if Ms}[\text{exp}] (e, s) \]
\[ \text{then Ms}[\text{stmt}_1] (\text{stmt}_3) (e, s) \]
\[ \text{else Ms}[\text{stmt}_2] (\text{stmt}_3) (e, s) \]
\[ \text{Ms}[\text{while exp do stmt}_1 \text{ od}] (\text{stmt}_2) (e, s) = \]
\[ \text{if Ms}[\text{exp}] (e, s) \]
\[ \text{then Ms}[\text{stmt}_1] (\text{while exp do stmt}_1 \text{ od}; \text{stmt}_2) (e, s) \]
\[ \text{else Ms}[\text{stmt}_2] (\text{return}) (e, s) \]
\[ \text{Ms}[\text{stmt}_1; \text{stmt}_2] (\text{stmt}_3) (e, s) = \]
\[ \text{Ms}[\text{stmt}_1] (\text{stmt}_2; \text{stmt}_3) (e, s) \]
\[ \text{Ms}[\text{goto id}] (\text{stmt}) (e, s) = \]
\[ \text{Ms}[\text{LocateLabel(id, stmt)} \text{ (return)} (e, s) \]
\[ \text{Ms}[\text{call id}] (\text{stmt}) (e, s) = \]
\[ \text{Ms}[\text{return}] (\text{return}) (e, e) \]

For procedure call, we simply take the procedure value associated with the identifier in the environment and apply it to the current state to produce the result of a call.

\[ \text{Ms}[\text{return}] (\text{stmt}) (e, s) = s \]

Return statements produce the final state resulting from a procedure invocation.

The final question to be resolved by the semantics of re-
cursive schemes is how procedure declarations bind identifiers to procedure values. Here, because PL/CS procedures may be mutually recursive, we must use a simultaneous recursive definition, i.e., a recursive definition which simultaneously gives the meaning of all of the declared procedures.

\[ \text{mp} \left[ \text{procedure; stmt end} \right] (e) = \lambda s : \text{Ms} \left[ \text{stmt} \right] \text{return} (e, s) \]

\[ \text{mp} \left[ \text{mainproc; Id}_1 \text{procdef}_1; \right. \]

\[ \left. \text{Id}_2 \text{procdef}_2; \ldots; \text{Id}_n \text{procdef}_n \right] (e) = \text{mp} \left[ \text{mainproc} \right] (e') \]

where

\[ e' = e[\text{id}_1 + \text{mp} \left[ \text{procdef}_1 \right] (e')]; \]

\[ \text{id}_2 + \text{mp} \left[ \text{procdef}_2 \right] (e'); \]

\[ \vdots \]

\[ \text{id}_n + \text{mp} \left[ \text{procdef}_n \right] (e') \]

Two points are of particular importance in understanding the preceding definition of the meaning of PL/CS recursion scheme programs. First, the meaning function uses a recursive definition of \( e' \), that is, for each procedure identifier we have

\[ e'(\text{id}_1) = \text{mp} \left[ \text{procdef}_1 \right] (e') \]

\[ ^\dagger \text{Remark: We use } e' = e[\text{id + f}] \text{ to mean that } e' \text{ is } e \text{ altered to have value } f \text{ associated with the identifier id.} \]
This is necessary because in PL/CS any procedure can call all other declared procedures (including itself, of course).

The second point is that this definition of $e'$ is no more complex than an ordinary recursive function definition of the form

$$
\begin{align*}
  f_1 &= \lambda s : M . \text{stmt}_1(f_1, \ldots, f_n)(s) \\
  \vdots \\
  f_n &= \lambda s : M . \text{stmt}_n(f_1, \ldots, f_n)(s)
\end{align*}
$$

Our decision to give a definition of recursive procedures in terms of a recursive definition of an environment is simply a matter of notational convenience. By rewriting the definition slightly, we can focus on the fact that the crux of the matter is the mutually recursive definition of a set of procedures, e.g.

$$
\begin{align*}
  e'(id_1) &= \lambda s : M . \text{stmt}_1 \{ e(id_1 + e'(id_1); \ldots; id_n + e'(id_n) \\
  \vdots \\
  e'(id_n) &= \lambda s : M . \text{stmt}_n \{ e(id_1 + e'(id_1); \ldots; id_n + e'(id_n)
\end{align*}
$$

This form also shows more clearly that $e'$ may be a total function (i.e., always well-defined) even if one or more of the procedures defined in a program happens to be non-terminating.

(2.4) PL/CS variable declaration and parameter passing.

We now make the final major refinement of the underlying "state" used to give meaning to PL/CS programs to introduce the concepts of variable declaration and parameter passing.

A "program state" merely indicates the status of the pro-
gram's world. So far our abstract states have only indicated the form of this world without mention of its content. In general, the execution of a program will create objects and examine them to detect certain properties.

In PL/CS only simple objects can be directly created (objects like numbers, strings or arrays of numbers and strings, but not objects like sets or functions or graphs). We will call these objects values. The language provides basic operations for building values, such as addition of numbers, concatenation of strings, and certain tests of properties of objects, such as equality of numbers and strings. (These operations and tests are written as expressions and boolean expressions in PL/CS.) But regardless of exactly how we build objects in a language (i.e., what semantics are given to expressions), we have the more basic problem of how to name them and pass them around to be modified or tested. So before discussing the way we build values, we will examine how we name them and move them regardless of what they are.

We first introduce a domain \( V \) of values which are produced by the expressions in the language and are stored in the state as values of the program variables. Of major importance in designing a language is deciding how values are associated with the identifiers used to denote variables in a program. There are two essential ways to do this. One way is to allow identifiers to reference values directly, so identifiers are reference objects. Another way is to assume a mathematical ob-
ject called a reference object (or location, or address) and let identifiers name such objects. These reference objects in turn name the values.

The first method, having identifiers directly reference values, is characteristic of ordinary mathematical notation. One says, "let x range over set s", or "let x be an integer"; and this creates a "variable" x whose values belong to s. These mathematical variables are treated quite informally. We can distinguish different uses according to the frequency with which the values of the variable might change. For instance, when we say "let f be the function ..." we rarely change the meaning of f in that context. Even when we do change its value, we do it by changing context, not by computation.

In ordinary mathematical notation we do not have an operation for explicitly changing the value of a variable. We might say, "let y = f(x)", and then when x changes value so will y, but x is usually given a relatively fixed meaning. (To understand this more fully, look at formal mathematical languages like the predicate calculus.)

In programming languages we do have operations for explicitly changing the value associated with an identifier. This is a basic operation in computation because we are concerned with abbreviations and space managing devices.

The concept of a reference object or location is convenient for discussing ideas associated with dynamic changes of identifier meaning (especially scope rules for identifiers and parameter passing mechanisms). Since locations are currently standard in
programming language semantics and are naturally related to implementation, we adopt them in this report.

In order to present the PL/CS language features associated with the manipulation and referencing of values and allocation of locations, we need to introduce the domain Val of values and the domain Loc of locations. In a typical programming language there are several types of values, e.g. integers, reals, characters, integer arrays, etc. But at this level we consider a single homogeneous domain of values. We refine the store so that it consists of mappings from locations to values, and we extend environments to include binding identifiers to locations.

Definitions:

Val is a set called the values
Loc is a set called the locations
S = Loc → Val
Env = Id → [Proc + Loc]
Proc = S + S

Given this interpretation of variables, we can ask two questions about the binding of a particular location to an identifier:

1. What is the scope of such a binding, i.e., over what region of a program is it the case that this identifier is bound to this location? In PL/CS, there are two obvious choices for the scope of a variable. First, scope could be local to the procedure in which the variable is declared. This is the default scope in PL/CS.
Second, scope could be global to the program, i.e., the binding could potentially be known to all procedures forming a program. In PL/CS, this is EXTERNAL scope. The bindings of EXTERNAL variables are known in all procedures in which the variable is declared to be EXTERNAL.

2. What is the extent to such a binding, i.e., how long does it persist? Again, in PL/CS there are two possibilities. First, the bindings of local variables of a procedure could be established upon each procedure invocation (this is also the default in PL/CS). The second choice is to bind local identifiers to locations prior to evaluation of the program. In PL/CS, this is the meaning used if a variable is declared to be STATIC.

One final wrinkle in our new notion of environment and store is introduced by the parameter-passing mechanisms of PL/CS. In the language, two different types of parameters exist:

1. Parameters which may be both read and changed within the procedure body (the default case), and

2. READONLY parameters which may be read (and not changed) inside the procedure.

In the case of READONLY parameters, we can regard these identifie:
as denoting constant values over the execution of the procedure (since the value cannot be changed). In this case, a direct binding of values to identifiers in the environment seems to reflect most accurately the semantics of this form of parameter passing.

Our new PL/CS sublanguage includes the following syntactic elements (we give the syntax of the entire language to refresh the reader's memory):

```
Stmt = call Id(Exp*)
   | if Exp then Stmt else Stmt fi
   | while Exp do Stmt od
   | Stmt;Stmt
   | do Stmt end
   | goto Id
   | return
   | Id:Stmt
```

```
ProcDef = procedure(Id*); Body end
Body = Decl*; Stmt
Decl = declare Id* Attr [Access]
Access = external | static | readonly
Program = Id:MainProc (; Id:ProcDef)*
MainProc = procedure options (main); Body end
```

The various semantic domains and functions used to define the language include the following:
Val values
Loc locations
$S = Loc + [Val \times Tag]$ stores
$Tag = \{true, false\}$ allocation tags

Allocation tags are used to record whether
the location is currently accessible, i.e.,
is bound to a variable currently in use.

$Env = Id + [Val + Proc + Loc]$ environments
$Proc = [Loc + Val]^* + S + S$ procedures

In this subset of the language, procedures
are now parameterized.

$M_s: Stmt + Stmt + Env + S + S$
$M_b: [Id^* \times Body] + Env + S + S$

The meaning of a procedure body is also
dependent on which identifiers appear as
parameters to the procedure.

$M_p: ProcDef \to S \to [(Env + Proc) \times S]$
$M_{proq}: Program + Env + S + S$

The new $M_p$ must not only produce the procedure value spe-
cified by a procedure declaration, but must also allocate
static variables declared within the procedure.

Finally, the clauses of the definition are given below.
\( M_s [ \text{call id(arg\text{*})}] (\text{stmt})(e,s) = \)
\( M_s [ \text{stmt}] (\text{return})(e,e(id)(M_a [\text{arg\text{*}}](e,s))(s)) \)

where \( M_a : \text{Exp} \rightarrow \text{Env} \rightarrow S \rightarrow [\text{Loc} + \text{Val}] \) is defined by

\( M_a [\text{id}](e,s) = e(id) \)
\( M_a [\text{exp}](e,s) = M_e [\text{exp}](e,s) \)

If the argument is an identifier, pass its associated location; otherwise, pass the value of the expression.
\[ Ms[\text{if } \text{exp} \text{ then } \text{stmt}^1 \text{ else } \text{stmt}^2 \text{ fi}](\text{stmt}^3)(e,s) = \]
\[ \text{if } Ms[\text{exp}](e,s) \]
\[ \text{then } Ms[\text{stmt}^1](\text{stmt}^3)(e,s) \]
\[ \text{else } Ms[\text{stmt}^2](\text{stmt}^3)(e,s) \]

\[ Ms[\text{while } \text{exp} \text{ do } \text{stmt}^1 \text{ od}](\text{stmt}^2)(e,s) = \]
\[ \text{if } Ms[\text{exp}](e,s) \]
\[ \text{then } Ms[\text{stmt}^1](\text{while } \text{exp} \text{ do } \text{stmt}^1 \text{ od}; \text{stmt}^2)(e,s) \]
\[ \text{else } Ms[\text{stmt}^2](\text{return})(e,s) \]

\[ Ms[\text{stmt}^1;\text{stmt}^2](\text{stmt}^3)(e,s) = \]
\[ Ms[\text{stmt}^1](\text{stmt}^2;\text{stmt}^3)(e,s) \]

\[ Ms[\text{do } \text{stmt}^1 \text{ end}](\text{stmt}^2)(e,s) = \]
\[ Ms[\text{stmt}^1](\text{stmt}^2)(e,s) \]

\[ Ms[\text{goto } \text{id}](\text{stmt}(e,s) = \]
\[ Ms[\text{LocateLabel}(\text{id},\text{stmt})](\text{return})(e,s) \]

\[ Ms[\text{return}](\text{stmt})(e,s) = s \]

\[ Ms[\text{id};\text{stmt}^1](\text{stmt}^2) = \]
\[ Ms[\text{stmt}^1](\text{stmt}^2) \]

Now for procedure declarations, recall from above that \( Mb \) (used to give meaning to procedure bodies) is of type

\[ Mb: [\text{Id}^* \times \text{Body}] \rightarrow \text{Env} \rightarrow S \rightarrow S \]

The definition of \( Mb \) is as follows:
\[ M \mathbb{C} \mathbb{I} d^* \times (\text{decl}^*; \text{stmt}) \mathbb{E} (e, s) = \]
\[ M \mathbb{C} \text{stmt} \mathbb{D} \text{return} \mathbb{J} (e', s') \]

where

\[ e', s' = \text{Alloc}(\text{Locals}(\text{decl}^*) \not\in id^*, e, s) \]

where:

1. \textbf{Locals:Decl}^* \rightarrow \text{Decl}^*
   produces a list of all declarations not declaring STATIC or EXTERNAL variables, i.e., the declarations of local variables. Note that because of the PL/CS syntax, we must be careful to remove declarations of parameters.

2. \textbf{Alloc:}([\text{Decl}^* \times \text{Env} \times S] \rightarrow [\text{Env} \times S]
   produces a new environment and store such that the identifiers referred to in the declaration are bound to previously unallocated storage locations, and these locations have been tagged as allocated in the resulting store.

From earlier, the type of \( M_p \), the meaning function for procedure declarations was of type

\[ M_p; \text{ProcDef} + S + [[\text{Env} \rightarrow \text{Proc}] \times S] \]

The definition of \( M_p \) is given by

\[ M_p \llbracket \text{procedure}(id^*); \text{body end} \rrbracket (s) = <p, s'> \]
where

\[ p = \lambda e:la^* : s p : \]

\[ Mb( id^* \times body \{(e[id^* \rightarrow ValArg(id^*, body, a^sp); id' \rightarrow i'])(s') \}

where

\[ id', i', s' = New(Statics(body), s) \]

where

1. \( \text{New} : [\text{Declaration}^* \times S] \rightarrow [\text{Id}^* \times \text{Loc}^* \times S] \)

allocates new locations for each of the identifiers appearing in the sequence of declarations and returns

a. the identifiers for which storage was allocated,

b. the locations allocated, and

c. the altered store.

Note that we can define the previously used \( \text{Alloc} \) function in terms of \( \text{New} \) by

\[ \text{Alloc}(\text{decl}^*, e, s) = \langle e[\text{id} \rightarrow i], s' \rangle \]

where

\[ \text{id}, t, s' = \text{New}(\text{decl}^*, s) \]

2. \( \text{ValArg} : [\text{Id}^* \times \text{Body} \times [\text{Loc} + \text{Val}]^* \times S] \rightarrow [\text{Loc} + \text{Val}]^* \)

is used to bind arguments to \text{READONLY} (or \text{value}) parameters within a procedure body. If the argument is a location and the corresponding parameter is de-
clared to be READONLY, then a value \((s(1))\) is pro-
duced. Otherwise, the argument is unchanged.

\[
\text{Mprog}[ \text{id:procedure options(main); body end;}
\]

\[
\text{id}_1: \text{procdef}_1; \ldots; \text{id}_n: \text{procdef}_n \implies (e, s) = \\
\text{M}[ \text{body} \implies (e', s') \text{ where}
\]

\[
\begin{align*}
\text{e}_{\text{ext}}' & = \text{Alloc(Externals(Program), e, s)}; \\
\text{P}_1, \text{s}_1 & = \text{M}[ \text{procdef}_1 \implies \text{s}_{\text{ext}}]; \\
\vdots & \quad \vdots \\
\text{P}_n, \text{s}_n & = \text{M}[ \text{procdef}_n \implies \text{s}_{\text{n-1}}]; \\
s' & = \text{s}_n; \\
e' & = \text{e}_{\text{ext}}[\text{id}_1 + \text{p}_1(e'); \ldots; \text{id}_n + \text{p}_n(e')]
\end{align*}
\]

Some remarks on the meaning given to programs are clearly in
order. The important point is that the definition includes
both "sequential" and "simultaneous" components. The sequential
component of the definition is the meaning given to procedures.
Because procedures now may contain local static variables and
references to external variables, it is necessary to sequentially
allocate storage for the external variables and the local static
variables of each procedure. Thus, we form the sequence of
stores \(s_{\text{ext}}, s_1, s_2, \ldots, s_n\) and the sequence of \(\text{Env} + \text{Proc values}
\]

\[
\text{P}_1, \ldots, \text{P}_n
\]
The simultaneous component of the definition is the formation of the environment $e'$, similar to the simultaneous recursive definition of the environment used in the preceding section on recursion schemes. It is the environment formed by this simultaneous definition that is used to evaluate the main procedure of the program.

(2.5) Expressions and assignment.

We have now built up the semantic model of PL/CS to define most of the language. One aspect which remains undiscussed is the way in which values (elements of the semantic domain $V$) may be produced. The evaluation of expressions will form the content of the next two sections. In this section, we will introduce a new meaning function $M_e$ to define the meaning of expressions and discuss the semantics of PL/CS function declaration, call, and return and the semantics of assignment. In the next section, we refine Val to include a number of component domains and define the few remaining PL/CS constructs for which the semantics are dependent on particular components of Val.

In PL/CS, values can be constructed using certain basic functions, $f_i: Val^i \rightarrow Val$, and programmer-defined function procedures. These values are denoted by expressions. For example, various expressions for boolean values (used as selectors in conditional and iteration statements) can be built up in terms of atomic predicates, $p_j: Val^j \rightarrow \{\text{true, false}\}$
and other programmer-defined predicates.

The basic syntax of PL/CS expressions is given below:

\[
\text{Exp} = \text{Id} \mid \text{Const} \mid \text{Exp Op Exp} \mid \text{Id}(\text{Exp}^+) \\
\text{Op Exp}
\]

In PL/CS, all programmer-defined function procedures have the set theoretic type \(0^n \rightarrow 0\), that is of \(n\)-tuples of individuals to individuals. The declaration must specify the exact type of the function, e.g. number of arguments, their attributes and attribute of the returned value.

\[
\text{FuncDef} = \text{Id:procedure(\text{Id}^*) returns(\text{Attr});} \\
\text{Stmt; end Id}
\]

The meaning of a function declaration is similar to that of a procedure declaration, except that functions produce members of Val, rather than S, as final values. Thus, there must be a method for specifying the value of the statement which makes up the function body. This is done by executing \text{return(exp)}, which has the effect of storing the value of the expressions exp in a special component of the state called \text{fvalue}. Then after execution of the function procedure body, a special function, \text{ValueOf:S + Val} is applied to return the value of the \text{fvalue} component. More precisely, we now have the following definitions of function declaration and return. First, we give the new semantic domains and meaning functions involved.
Env = Id + [Val + Proc + Loc + Fun]

Fun = Val* + S + Val

S = Loc + [Val × Tag] × Val

the second component of S will be used as the
fvalue component to return the value produced
by a function body.

Mf: FuncDef → S → ([Env + Fun] × S)

the meaning function for function declaration.

M[e][return](exp) [] (stmt) (e, s) =
  s[fvalue + Me[exp] (e, s)]

Mf: FuncDef → S → ([Env + P] × S)
is defined by

M[e][procedure(id*) returns (attr); body end] (s) =
  \<λv:λv*:λs*:ValueOf(Mb[id* × body] (e[id* = v*; id* = ''] (s'))), s'\>

where

id*, l*, s' = New(Statics(body), s[fvalue + undefined])

We now define a PL/CS program to be a main procedure,
followed by a sequence of procedure and function procedure
declarations, i.e., we have the new syntactic definitions

Program = MainProc (; Id:ProcDef)* (; Id:FuncDef)*

with the following new semantic definition of programs:
\[\text{Mprog} \{ \text{id: procedure options(main); body end; } \]
\[\text{id}_1: \text{procdef}_1; \ldots; \text{id}_n: \text{procdef}_n; \]
\[\text{id}_{n+1}: \text{funcdef}_1; \ldots; \text{id}_{n+m}: \text{funcdef}_m \} (e, s) = \]
\[\text{Mb} \{ \text{body} \} (e', s') \]

where

\[e_{\text{ext}}' = \text{Alloc(Externals(program), e, s)};\]
\[P_1, s_1 = \text{Mp} \{ \text{procdef}_1 \} (s_{\text{ext}});\]
\[
\vdots
\]
\[P_n, s_n = \text{Mp} \{ \text{procdef}_n \} (s_{n-1});\]
\[f_1, s_{n+1} = \text{Mf} \{ \text{funcdef}_1 \} (s_n);\]
\[
\vdots
\]
\[f_m, s_{n+m} = \text{Mf} \{ \text{funcdef}_m \} (s_{n+m-1});\]

\[s' = s_{n+m};\]
\[e' = e_{\text{ext}}'[\text{id}_1 + P_1(e'); \ldots; \text{id}_n + P_n(e'); \]
\[\text{id}_{n+1} + f_1(e'_{\text{NOP}}); \ldots; \text{id}_{n+m} + f_m(e'_{\text{NOP}})]\]

where

\[e_{\text{NOP}} = \lambda i: \text{if } e[i] \text{ is Proc then error else } e[i]\]

i.e., \(e_{\text{NOP}}\) masks all procedure values in \(e\). This is necessary in PL/CS to disallow the possibility of procedure calls inside the body of functions.

The meaning of expressions is given by a function
\[
Me: \text{Exp} \rightarrow \text{Env} + S + \text{Val}
\]

defined as follows:

\[
Me[\text{id}](e,s) =
\]

\[
\begin{align*}
\text{if } e[\text{id}] \text{ is Val then } e[\text{id}] \\
\text{else } s(e[\text{id}]) & \quad /* e[\text{id}] must be a location */
\end{align*}
\]

\[
Me[\text{const}](e,s) = \text{const}
\]

\[
Me[\text{op exp}](e,s) = \text{Uop}_{\text{op}}(Me[\text{exp}](e,s))
\]

where \text{Uop:Val} \rightarrow \text{Val} defines each unary operation

\[
Me[\text{exp}_1 \text{ op exp}_2](e,s) =
\]

\[
\text{Bop}_{\text{op}}(Me[\text{exp}_1](e,s), Me[\text{exp}_2](e,s))
\]

where \text{Bop:[val \times Val]} \rightarrow \text{Val} defines each binary operation in PL/CS

\[
Me[\text{id}(\text{exp}^*)](e,s) =
\]

\[
(s)(e[\text{id}](Me[\text{exp}^*](e,s)))(s)
\]

Now, given this definition of \(Me\), we can define one of the basic statements of PL/CS, the simple assignment statement, as follows:

\[
Me[\text{id}=\text{exp}](\text{stmt})(e,s) =
\]

\[
Me[\text{stmt}](e,s[e[\text{id}] = Me[\text{exp}](e,s))]
\]

(2.6) Integers, arrays, and bounded iteration.

The final aspect of PL/CS to be discussed involves a further refinement of the domain \text{Val}, so that arrays and bounded
iterations can be defined. The domain of basic values in PL/CS is made up of the following elements

\[ \text{BVal} = \text{Char} + \text{Num} + \text{Bool} \]

where

\[ \text{Char} = \text{the domain of PL/CS character strings} \]
\[ \text{Num} = \text{the domain of PL/CS numbers} \]
\[ \text{Bool} = \{\text{true}, \text{false}\} \text{ the domain of boolean values} \]

An important subset of the elements of \text{Num} is the set of integers, denoted by \text{Int} (or \text{N}). Using the domain \text{Int}, we can define the domain of PL/CS array values as

\[ \text{Aval} = \text{Int}^n \rightarrow \text{Bval} \text{ for } n = 1, 2, \ldots \]

Also, in PL/CS one can declare array variables, which we define as elements of the domain

\[ \text{Ar} = [\text{Int}^n \times \text{Loc}] \times \text{Int}^n \]

where the second component of an array variable defines the upper bound allowed in each subscript position (the lower bound in PL/CS is always assumed to be 1). Thus, in addition to our earlier declaration of simple variables, we now have the following syntax for array declarations.

\[ \text{Decl} = \text{declare Id} (\text{Exp} , \text{Exp}) \text{Attr} [\text{Access}] \]

The semantics of the allocation of array variables can be
simply given by the obvious extensions of the functions Alloc and New to allocate a distinct location for each valid subscript specified in the array declaration.

The final use for integer-valued expressions in PL/CS is to serve as the indices for bounded iterations of the form

\[
do \text{id:=}\text{exp}_1 \text{ to } \text{exp}_2 \text{ by } \text{exp}_3 \text{ stmt end}
\]

where none of the variables appearing in the expressions \text{exp}_1, \text{exp}_2, and \text{exp}_3 may appear in the statement body stmt. The meaning of this new statement is given by the following clause of \(\text{Ms}^3\):\[
\text{Ms} \left[ \begin{array}{c}
do \text{id:=}\text{exp}_1 \text{ to } \text{exp}_2 \text{ by } \text{exp}_3 \text{ stmt_1 end} \\
\end{array} \right] (\text{stmt}_2)(e,s) = \\
\text{if Ms} \left[ \begin{array}{c}
\text{exp}_1 \\
\end{array} \right] (e,s) \leq \text{Ms} \left[ \begin{array}{c}
\text{exp}_2 \\
\end{array} \right] (e,s) \text{ then} \\
\text{Ms} \left[ \begin{array}{c}
\text{id:=}\text{exp}_1; \text{stmt}_1 \\
\end{array} \right] (\text{do id:=exp}_1 + \text{exp}_3 \text{ to } \text{exp}_2 \text{ by } \text{exp}_3; \\
\text{stmt}_1 \text{ end}; \text{stmt}_2)(e,s) \\
\text{else Ms} \left[ \begin{array}{c}
\text{stmt}_2 \\
\end{array} \right] (\text{return}(e,s(e(id) = \text{undefined}))
\]

§3 Conclusion

One of the major difficulties of language design is attempting to describe the final result to a potential user (and in some cases, to the designer himself). The problem is that language definitions must simultaneously be:

1. Precise. The user must be told exactly what can and cannot be done by each construct of the language.
2. Concise. Few users are willing to struggle through several hundred pages of text to find out where to put the semicolons. Note that the requirement of precision often is in contradiction with concision.

3. Easy to read. This should be obvious, but is frequently overlooked (e.g., the descriptions of Algol 68 with the myriad levels of Report, Introduction to, Informal Introduction to, Very Informal Introduction to, ...). This means that not only should the prose be highly polished, but the definition should be given in terms common to the intended audience (how many of us have ever heard of meta-, proto-, and hyper-notions before?).

In this report, we have attempted to give a description of PL/CS which meets these requirements. Our use of mathematical semantics as a method of presentation is not new; what is new is primarily the restraint used in its application. PL/CS is a simple language and its semantics should be presented in as simple a manner as possible, consistent, of course, with the goal of precision.

In this regard, we have also used the results of tentative simple formal definitions as a guide to proscribing certain PL/CS features. For example, the choice of providing only a forward goto
in PL/CS was motivated mainly by an intuitive belief in the need to restrict the use of jumps. Our decision was reinforced, however, with the realization that such simple gotos could be described using a much simpler technique than that required for the full complexity of PL/I gotos. Also, the fact that PL/CS functions have no side-effects allowed us to give them a meaning consistent with the ordinary mathematical meaning of functions. Thus, in PL/CS there is one less departure from the ordinary to confuse the neophyte programmer.

We feel that our approach of levels of language and successive refinement of the domains used in the definition provides one way of achieving one of the major goals of formal semantics - improving the communication between language designer and user. We welcome any comments or criticisms.
REFERENCES

[Conway 1976]

[Conway and Gries 1975]

[Conway and Wilcox 1973]

[Donahue 1976]

[Hoare 1969]

[Hoare 1971]

[Hoare and Wirth 1973]

[Scott and Strachey 1971]
[Strachey and Wadsworth 1974]
# APPENDIX

## Syntactic Domains

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>identifiers</td>
</tr>
<tr>
<td>Const</td>
<td>constants</td>
</tr>
<tr>
<td>UnOp = (+, -, -)</td>
<td>unary operators</td>
</tr>
<tr>
<td>BinOp = (+, -, &amp;[,],*,/)</td>
<td>binary operators</td>
</tr>
<tr>
<td>Var = Id</td>
<td>variables</td>
</tr>
<tr>
<td>Exp = Var + Id(Exp,...,Exp)</td>
<td>subscripted variables</td>
</tr>
<tr>
<td>Exp = Var + Const</td>
<td>expressions</td>
</tr>
<tr>
<td>Exp = Var + [UnOp Exp]</td>
<td>labels</td>
</tr>
<tr>
<td>Exp = Var + Id(Exp*)</td>
<td>statements</td>
</tr>
<tr>
<td>Lab = Id*</td>
<td>gotos</td>
</tr>
<tr>
<td>Stmt = [Id:=Exp]</td>
<td>procedure return</td>
</tr>
<tr>
<td>Stmt = [goto Id]</td>
<td>function return</td>
</tr>
<tr>
<td>Stmt = [return (Exp)]</td>
<td>conditionals</td>
</tr>
<tr>
<td>Stmt = [if Exp then Stmt else Stmt fi]</td>
<td>unbounded iterations</td>
</tr>
<tr>
<td>Stmt = [while Exp do Stmt od]</td>
<td>sequences</td>
</tr>
<tr>
<td>Stmt = [do Stmt end]</td>
<td>do groups</td>
</tr>
<tr>
<td>Stmt = [call Id(Exp*)]</td>
<td>procedure calls</td>
</tr>
<tr>
<td>Stmt = [do Id:=Exp to Exp by Exp; Stmt end]</td>
<td>bounded iterations</td>
</tr>
<tr>
<td>ProcDef = procedure(Id*); Body end</td>
<td>labelled statements</td>
</tr>
<tr>
<td>Body = Decl*; Stmt</td>
<td>procedure definitions</td>
</tr>
<tr>
<td>Decl = declare Id* Attr [Access]</td>
<td>procedure/function bodies</td>
</tr>
<tr>
<td>Access = (external, static, readonly)</td>
<td>variable declarations</td>
</tr>
<tr>
<td>FuncDef = procedure(Id*) returns (Attr); Body end</td>
<td>access attributes</td>
</tr>
<tr>
<td></td>
<td>function definitions</td>
</tr>
</tbody>
</table>
Program = Id:MainProc
    (; Id:ProcDef)*
    (; Id:FuncDef)*
MainProc = procedure options(main); Body end

Semantic Domains

Bval = Int + Char + Bool + Real
Int
Char
Bool = (true,false)
Real
Aval = [Int^n + Bval] x Int^n
Val = Bval + Aval
Loc
Ar = [Int^n + Loc] x Int^n
S = [Loc + [Bval x Tag]] x Bval
Fun = Val* + S + Bval
Proc = [Loc + Val]* + S + S
Env = Id + [Val + Proc + Func + Loc + Ar]

Meaning Functions

Meaning of expressions:
    Me:Exp + Env + S + Val

Meaning of statements
    Ms:Stmt + Stmt + Env + S + S

Meaning of procedure and function definitions and programs
    Mp:ProcDef + S + [[Env + Proc] x S]
    MF:FuncDef + S + [[Env + Fun] x S]
    Mprog:Program + Env + S + S

Meaning of procedure arguments
    Ma:Exp-> Env + S + [Loc + Val]

Meaning of function and procedure bodies
    Mb:[Id* x Body] + Env + S + S
Auxiliary Functions

Alloc: Decl* → [Env × S] → [Env × S]
variable declaration semantics

New: [Decl* × S] → [Id* × Loc* × S]
storage allocation function

Val: [(Loc* + Val)* × Id* × Body* × S] → [Loc + Val]*
coercion of readonly arguments to constant values

Statics: Body → Decl*

Externals: Program → Decl*

Locals: Body → Decl*
syntactic functions producing declarations of all
static, external, or local variables in a program
or procedure or function body

Uop: Op → Val → Val

Bop: Op → [Val × Val] → Val
define unary and binary operators

LocateLabel: Id + Stmt + Stmt
finds the statement in its second argument
labelled by its first argument

Clauses of the definitions

Expressions

Me[ id ](e,s) =
  if e[ id ] is Val then e[ id ] else s(e[ id ])

Me[ const ](e,s) = const

Me[ op exp ](e,s) = Uop 오( Me[ exp ](e,s) )

Me[ exp1 op exp2 ](e,s) =
  Bop 오( Me[ exp1 ](e,s), Me[ exp2 ](e,s) )

Me[ id(exp*) ](e,s) = (e[ id ])( Me[ exp* ](e,s) )
Statements

\[
\begin{align*}
\mathtt{Ms}[\mathtt{id}=\mathtt{exp}] & \ (\mathtt{stmt})(e,s) = \\
\mathtt{Ms}[\mathtt{stmt}] & \ (e,s[e[\mathtt{id}]=\mathtt{Ms}[\mathtt{exp}](e,s)]) \\
\mathtt{Ms}[\mathtt{goto id}] & \ (\mathtt{stmt})(e,s) = \\
\mathtt{Ms}[\mathtt{LocateLabel}(id,\mathtt{stmt})] & \ (\mathtt{return})(e,s) \\
\mathtt{Ms}[\mathtt{return}] & \ (\mathtt{stmt})(e,s) = s
\end{align*}
\]

\[
\begin{align*}
\mathtt{Ms}[\mathtt{return}](\mathtt{exp}) & \ (\mathtt{stmt})(e,s) = s[\mathtt{fvalue} + \mathtt{Ms}[\mathtt{exp}](e,s)] \\
\mathtt{Ms}[\mathtt{call id}(\mathtt{arg}^*)] & \ (\mathtt{stmt})(e,s) = \\
\mathtt{Ms}[\mathtt{stmt}] & \ (\mathtt{return})(e, e[\mathtt{id}](\mathtt{Ms}[\mathtt{arg}^*])(e,s)) \\
\mathtt{Ms}[\mathtt{if}] & \ (\mathtt{exp}\ \mathtt{then}\ \mathtt{stmt}_1\ \mathtt{else}\ \mathtt{stmt}_2\ \mathtt{fi}](\mathtt{stmt}_3)(e,s) = \\
& \mathtt{if} \ \mathtt{Ms}[\mathtt{exp}](e,s) \\
& \ \mathtt{then} \ \mathtt{Ms}[\mathtt{stmt}_1](\mathtt{stmt}_3)(e,s) \\
& \ \mathtt{else} \ \mathtt{Ms}[\mathtt{stmt}_2](\mathtt{stmt}_3)(e,s) \\
\mathtt{Ms}[\mathtt{while}] & \ (\mathtt{exp}\ \mathtt{do}\ \mathtt{stmt}_1\ \mathtt{od}](\mathtt{stmt}_2)(e,s) = \\
& \mathtt{if} \ \mathtt{Ms}[\mathtt{exp}](e,s) \\
& \ \mathtt{then} \ \mathtt{Ms}[\mathtt{stmt}_1](\mathtt{while}\ \mathtt{exp}\ \mathtt{do}\ \mathtt{stmt}_1\ \mathtt{od}\ \mathtt{stmt}_2)(e,s) \\
& \ \mathtt{else} \ \mathtt{Ms}[\mathtt{stmt}_2](\mathtt{return})(e,s) \\
\mathtt{Ms}[\mathtt{stmt}_1;\mathtt{stmt}_2] & \ (\mathtt{stmt}_3)(e,s) = \\
\mathtt{Ms}[\mathtt{stmt}_1] & \ (\mathtt{stmt}_2;\mathtt{stmt}_3)(e,s) \\
\mathtt{Ms}[\mathtt{do}] & \ (\mathtt{stmt}_1\ \mathtt{end}](\mathtt{stmt}_2)(e,s) = \\
\mathtt{Ms}[\mathtt{stmt}_1] & \ (\mathtt{stmt}_2)(e,s) \\
\mathtt{Ms}[\mathtt{id}=\mathtt{exp}_1\ \mathtt{to}\ \mathtt{exp}_2\ \mathtt{by}\ \mathtt{exp}_3;\mathtt{stmt}_1\ \mathtt{end};\mathtt{stmt}_2](e,s) = \\
\mathtt{if} \ \mathtt{Ms}[\mathtt{exp}_1] = \mathtt{Ms}[\mathtt{exp}_2](e,s) \\
& \ \mathtt{then} \ \mathtt{Ms}[\mathtt{id}=\mathtt{exp}_1;\mathtt{stmt}_1] \ (\mathtt{do}\ \mathtt{id}=\mathtt{exp}_1\ \mathtt{+}\ \mathtt{exp}_3\ \mathtt{to}\ \mathtt{exp}_2\ \mathtt{by}\ \mathtt{exp}_3;\mathtt{stmt}_1\ \mathtt{end};\mathtt{stmt}_2)(e,s) \\
& \ \mathtt{else} \ \mathtt{Ms}[\mathtt{stmt}_2](\mathtt{return})(e,s[e[\mathtt{id}]=\mathtt{undefined}]) \\
\mathtt{Ms}[\mathtt{id};\mathtt{stmt}_1] & \ (\mathtt{stmt}_2) = \mathtt{Ms}[\mathtt{stmt}_1](\mathtt{stmt}_2)
\end{align*}
\]
\[ Mp \llbracket \text{procedure}(id^*) \rrbracket; \text{body end} \rrbracket (s) = \langle p, s' \rangle \]
where
\[ p = \lambda e: a^*: \lambda s^*: \text{Mb} \llbracket id^* \times \text{body} \rrbracket \]
\[ (e[id^* + \text{Val}(a^*, id^*, \text{body}, s^*); id^* + 1'], s^*) \]
where
\[ id^*, l', s' = \text{New}(\text{Statics(body)}, s) \]

\[ Mf \llbracket \text{procedure}(id^*) \text{returns}(attr); \text{body end} \rrbracket (s) = \langle f, s' \rangle \]
where
\[ f = \lambda e: \lambda v^*: \lambda s^*: \text{ValueOf}(\text{Mb} \llbracket id^* \times \text{body} \rrbracket (e[id^* + v^*; id^* + 1'], s^*)) \]
where
\[ id^*, l', s' = \text{New}(\text{Statics(body)}, s[fvalue + \text{undefined}]) \]

\[ Mproq \llbracket \text{id:procedure options(main); body end}; \]
\[ \text{id}_1: \text{procdef}_1; \ldots; \text{id}_n: \text{procdef}_n; \]
\[ \text{id}_{n+1}: \text{funcdef}_1; \ldots; \text{id}_{n+m}: \text{funcdef}_m \rrbracket (e, s) = \]
\[ \text{Mb} \llbracket \text{body} \rrbracket (e', s') \]
where
\[ e_{\text{ext}}', s_{\text{ext}} = \text{Alloc}(\text{Externals} \text{(program)}, e, s); \]
\[ p_1, s_1 = \text{Mb} \llbracket \text{procdef}_1 \rrbracket (s_{\text{ext}}); \]
\[ \ldots \]
\[ p_n, s_n = \text{Mb} \llbracket \text{procdef}_n \rrbracket (s_{n-1}); \]
\[ f_1, s_{n+1} = \text{Mb} \llbracket \text{funcdef}_1 \rrbracket (s_n); \]
\[ \ldots \]
\[ f_m, s_{n+m} = \text{Mb} \llbracket \text{funcdef}_m \rrbracket (s_{n+m-1}); \]
\[ s' = s_{n+m}'; \]
\[ e' = e_{\text{ext}}[\text{id}_1 + p_1(e'); \ldots; \text{id}_n + p_n(e'); \text{id}_{n+1} + f_1(e'_{\text{NOP}}); \ldots; \text{id}_{n+m} + f_m(e'_{\text{NOP}})] \]
where
\[ e'_{\text{NOP}} = \lambda i: \text{if } e'[i] \text{ is Proc then error else } e[i] \]
\[ M_b \left[ \text{id} \times (\text{decl} ; \text{stmt}) \right] (e, s) = M_n \left[ \text{stmt} \right] (\text{return}) (e', s') \]

where
\[
e', s' = \text{Alloc} (\text{Locals} (\text{decl}) \text{ not in } \text{id}, e, s)\]

Array assignment

We have omitted a definition of array assignment from the equations given above because the current PL/CS definition is extremely cumbersome. We give a definition here so the reader can judge for himself; but we feel that the awkwardness of this definition is ample reason to prefer the simpler idea that array assignment should result in the state
\[ s(e(id)) = M_e \left[ \text{exp} \right] (e, s) \]

where \( e(id) \) is an array expression.

\[ M_s \left[ \text{id} \right] \left[ \text{exp} \right] (\text{stmt})(e, s) = \]

if \( \text{id} \) is simple then \[ M_s \left[ \text{stmt} \right] (e, s(e(id)) + M_e \left[ \text{exp} \right] (e, s)) \]

else let \( \text{id} \) be \( \text{id}(b_1, \ldots, b_n) \) in

\[ M_s \left[ \text{do } i_1 = 1 \text{ to } b_1 \text{ do } i_2 = 1 \text{ to } b_2 \right. \]

\[ \ldots \]

\[ \text{do } i_n = 1 \text{ to } b_n \]

\[ \text{id}(i_1, \ldots, i_n) := \text{exp} \]

end

end \[ \left[ \text{stmt} \right] (e, s) \]

where \( i_1, \ldots, i_n \) do not occur otherwise in the program.