Semantics of Simpl

(Simpl = simple imperative programming language)

1 Abstract syntax of Simpl

```
V     = Variables (identifiers)
Nb    = integers  (decimal notation)
Op    = + | - | * | / | =| < | ≤ | ≠
Expr  = Nb [1] | V [2] |succ Expr [3] | pred Expr [4] |
        Expr Op Expr [5] | procedure Instr [6]
Instr = V := Expr |  call Expr |  Instr ; Instr |
        if Expr then Instr else Instr |  while Expr do Instr
Prog  = read V ; Instr ; write Expr
```

Note

The syntax enforces no type distinction between integer, Boolean and procedural (macro-instruction) expressions.

2 Semantic domains for Simpl

N	Naturals {0,1,2,...}
B	Booleans {true, false}
P = (S→S)	procedures, state transformations
E = N+B+P	values of expressions, as stored or computed
S = (V→E)	states, values assigned to variables
V	variables, the same as listed in the abstract syntax.

A special “undefined” value ⊥ needs to be added to domain E for several purposes :

- uninitialised variables,
- division by zero or **pred** zero
- incompatible types of operator arguments may result in undefined values in E
- non-terminating **while** loops result in undefined states in S.

3 Semantic fonctions in Simpl

σ_{Expr} : Expr × S → E σ_{Expr} : Expr → (S → E)

σ_{Instr} : Instr x S → S σ_{Instr} : Instr → (S → S)

σ_{Prog} : Prog x N → N σ_{Prog} : Prog → (N → N)

Note

We deal with (equivalent) more or less Curried presentations

σ_{Expr} x = σ_{Expr} : Expr → (S → E)

case x [rule #] **of**

- x∈Nb [1] : valeur décimale de x, indép. de s;
- x∈V [2] : λs.s(x);
- **succ** x [3] : $\lambda s.\sigma_{Expr}\ x\ s\ |_N + 1$;
- **pred** x [4] : $\lambda s.cond((\sigma_{Expr}\ x\ s|_N)>0,\ (\sigma_{Expr}\ x\ s|_N)-1,\ \perp)$;
- Expr Op Expr[5] : operations on N and B are defined as usual.
  For example :
  $\sigma_{Expr}\ (x<y): \lambda s.(\sigma_{Expr}\ x\ s)|_N < (\sigma_{Expr}\ y\ s\ )|_N$;
- (**procedure** i)[6] : $\lambda s.\sigma_{Instr}\ i$;

end x ;

Note

(**procedure** i) is an unusual notion :

it is a state transformation, which abstracts with respect to the present state s.

σ_{Instr} i = $\qquad\qquad$ σ_{Instr} : Instr $\rightarrow$ (S $\rightarrow$ S)

case i **of**

(x:= y) : $\lambda s.s_x^n$ with n = σ_{Expr} y s;

(**call** y) : $\lambda s.(\sigma_{Expr}\ y\ s)\ |_P\ (s)$;

(x;y) : $\lambda s.\sigma_{Instr}\ y\ (\sigma_{Instr}\ x\ s)$

(**if** x **then** y **else** z) :
$\lambda s.cond(\sigma_{Expr}\ x\ s,\ \sigma_{Instr}\ y\ s,\ \sigma_{Instr}\ z\ s)$;

(**while** x **do** y) : $\lambda s.\lim_{i\to\infty} s_i$;
with s_0 = s, and
$s_{i+1} = cond\ (\sigma_{Expr}\ x\ s_i\ ,\ \sigma_{Instr}\ y\ s_i,\ s_i)$.

end i ;

Note

An equivalent definition for (**while** x **do** y) is the least **fixpoint** of the following transformation of p :

$$\lambda p.\lambda s.cond\ (\sigma_{Expr}\ x\ s,\ p(\sigma_{Instr}\ y\ s)\ ,\ s).$$

σ_{Prog} : Prog $\rightarrow$ (N $\rightarrow$ N)

σ_{Prog}(**read** x ; i ; **write** y) = $\lambda n.\sigma_{Expr}\ y\ (\sigma_{Instr}\ i\ s_x^n)$
with $x\in V$, $i\in$ Instr, $y\in$ Expr , $n\in N$
and $s(v) = \perp$ for every variable $v\in V$.

Note

- The notation $z|_D$ is used for $z|_D = z$ if $z\in D$ and $z|_D = \perp \neq z$ if $z\notin D$.
- The notation cond(a,b,c) is defined as cond(a,b,c) = b if a = true, c if a = false, and $\perp$ when $a\notin B$.
- The functions σ_{Expr} , σ_{Instr} et σ_{Progr}, are actually defined on the syntactic domains Expr, Instr et Progr respectively; they transform syntactic constructs from these domains into functions in (S$\rightarrow$E), (S$\rightarrow$S) and (N$\rightarrow$N) respectively.

Semantics of a block structured language

To make Simpl a practicable programming language several erxtensions are needed.

We first consider the familiar block structure (Algol). The idea is to allow an optional declaration before each instruction. The scope of a declaration is the instruction that it precedes.

Each declaration may introduce a new variable (local) or define a value name (constant or procedure).

1 Abstract syntax, extended with blocs (SimplB)

We need two new syntactic domains : Bloc and Decl. (we only indicate the differences with Simpl) :

Instr Instr (as in Simpl) | Bloc

Bloc Decl; Instr

Decl **var** V | **def** V=Expr

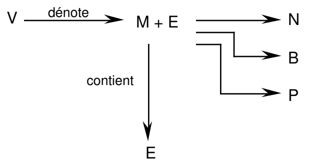
2 Semantic domains for SimplB

We need a new semantic domain M for memory locations or cells (each fit to store an integer, Boolean or procedural value, each designated by an identifier from V).

We also need the notion of an **environment**, i.e. the set of all definitions accessible from a bloc.

U = (V$\rightarrow$D)	environments (identifiers and their currently denoted value)
D = M + E	denotable objects : E = N+B+P, M = memory
S = (M$\rightarrow$E)	memory states (values contained in each cell)
P = (S$\rightarrow$S)	procedures.

Semantic model (Algol 68)



- A constant or procedure identifier **denotes** a value in E (of type N , B or P)
- A variable identifier **denotes** a cell of M, which in turn **contains** a value in E

An identifier (we still use domain V for identifiers) denotes :
either a cell in M (it is then called a **variable**),or a value of type E = N+B+P.

4 Semantic functions of SimplB

The meaning of a statement in SimplB will now depend on the meaning of each visible identifier in the bloc. Un additional argument is thus required for the semantic functions σ: the environment u∈ U

σ_{Instr} : Instr x U x S → S	σ_{Instr} : Instr x U → (S → S)
σ_{Decl}: Decl x U x S → U	σ_{Decl}: Decl x U → (S → U)
σ_{Expr}: Expr x U x S → E	σ_{Expr}: Expr x U → (S → E)

σ_{Instr} : Instr x U →(S → S)

σ_{Instr} i u =
  case i u **of**
    i u   : as in Simpl
               if i∈Instr but i∉ Bloc

    (d;i) u   : $\lambda s.\sigma_{Inst}$ i (σ_{Decl} d u s) s;
               d∈Decl
  end i u ;

σ_{Decl}: Decl x U →(S → U)

σ_{Decl} i u =
  case i u **of**
    (**var** x) u : $\lambda s.u^m_x$;
            with m∈M, **new** i.e. m∉u(V)
    (**def** x=e) u : $\lambda s.u_x^{\sigma_{Expr}\ e\ u\ s}$;
  end i u ;

Note

Declarations update (only) the environment
Instructions change (only) the state
Variable declarations require a memory management mechanism for M

σ_{Expr}: Expr x U $\rightarrow$(S $\rightarrow$E)

```
σExpr x u =
   case x u of
      x[2] u     : λs.s(u(x)|M);
                   for x ∈ V

    (proc i)[6] u:λs.σInstr  i u |P;
                   for I ∈ instruction.
      …
   end x u ;
```

We do not enumerate all cases of Expr, as they behave like in Simpl

A block-structured language with jumps (SimplBJ)

1 Syntactic and semantic domains of SimplBJ

Syntax is simple : Labels may be inserted ahead of any Instruction, and goto statements are added

Instr = (L **;** Instr) | Instr (as in SimplB) | (**goto** L)

Semantic domains are more elaborate

C =	$(S \rightarrow N)$	instruction continuations
C' =	$(E \rightarrow N)$	expression continuations
C" =	$(U{\times}S \rightarrow N)$	declaration continuations
P =	$(C{\times}S \rightarrow N)$	procedures, P = $(C \rightarrow C)$

The meaning of a label is the instruction that follows it (considered in its context).
Continuations capture "what has to be done from this point to the end of the program"

2 Semantic functions of SimplBJ

σ_{Instr}: Instr $\times$ U $\times$ C $\times$ S $\rightarrow$N

σ_{Expr}: Expr x U x C' $\times$ S $\rightarrow$N

σ_{Decl}: Decl x U x C" x S $\rightarrow$N

σ_{Prog}: Prog x N $\rightarrow$N

σ_{Instr} : Instr $\times$U $\times$C$\rightarrow$C

σ_{Expr} : Expr x U x C' $\rightarrow$C

σ_{Decl} : Decl x U x C" $\rightarrow$C

Note

σ_{Prog} (**read** x ; i ; **write** e) will be obtained by :
- declaring a variable x in the environment of i ;
- initialising the state s by assigning x the value read by the program
- using a trivial continuation for the (**write** e) statement

σ_{Instr} : Instr $\times$U $\times$C$\rightarrow$(S $\rightarrow$N)

```
σInstr i u c =
   case i u c of
      (i;i') u c: (σInstr i u) ∘ (σInstr i'u) c;

      (if e then i else i') u c:
               σExpr e u k; where k∈C' such that :
               k(b) = Cond(b, σInstr i u c , σInstr i' u c)

      (while e do i) u c :
               σExpr e u k;  with  k (recursively defined)
               k(b) = Cond(b, σInstr i u (σExpr e u k), c)

      (call e) u c    :

               σExpr e u k; with k∈C' such that:
               k(e') = e'|P (c)(s)
```

{continuation of continuations} σ_{Instr} : Instr $\times$ U $\times$ C $\rightarrow$ (S $\rightarrow$ N)

(l:i) u c : c'; where c' is defined (recursively) by
c' = σ_{Instr} $u^{c'}_{l}$ c

(d;i) u c : σ_{Decl} d u q; with q$\in$C" such as:
q(u') = σ_{Instr} i u'

(**goto** l)u c : u(l); the continuation c is lost !

end i u c;

σ_{Expr}: Expr x U x C' $\rightarrow$ (S $\rightarrow$ N)

σ_{Expr} x u k =
case x u k **of**
x [2]u k : λs.k(s(u(x))); with x$\in$V
(**procedure** i) [6] u k : k(σ_{Instr} i u);
...
end x u k;

σ_{Decl}: Decl x U x C" x S $\rightarrow$ N

σ_{Decl} d u q =
case d u q **of**
(**var** x) u q : q u^{m}_{x}; with q$\in$C", m$\in$M **new** !

(**def** x=e) u q : σ_{Expr}e u k;
with q$\in$C" and k$\in$C' such that
k(e') = q($u^{e'}_{x}$)(s)

end x u k;

Note

M requires a memory management mechanism (such as a heap or a stack)