6. A second example: the Language SMALL

The language SMALL described in this chapter has two roles:

(i) To illustrate the main ideas of standard semantics.
(ii) To provide a 'kernel language' which we shall extend as we discuss the semantics of various constructs in later chapters.

6.1. Syntax of SMALL

6.1.1. Syntactic domains

The *primitive* syntactic domains of SMALL are:

Ide	The domain of identifiers **I**
Bas	The domain of basic constants **B**
Opr	The domain of binary operators **O**

We do not further specify these primitive domains. **Bas** could, for example, be {**0, 1, 2, ...**} and **Opr** could be {+, -, ×, /, <, >, ...}.

The *compound* syntactic domains of SMALL are:

Pro	The domain of programs **P**
Exp	The domain of expressions **E**
Com	The domain of commands **C**
Dec	The domain of declarations **D**

These domains are defined by the following syntactic clauses:

6.1.2. Syntactic clauses

P ::= **program C**

E ::= **B** | **true** | **false** | **read** | **I** | $E_1(E_2)$
| **if E then** E_1 **else** E_2 | E_1 **O** E_2

C ::= E_1 := E_2 | **output E** | $E_1(E_2)$ | **if E then** C_1 **else** C_2
| **while E do C** | **begin D;C end** | $C_1;C_2$

D ::= **const I** = **E** | **var I** = **E** | **proc** $I(I_1);C$ | **fun** $I(I_1);E$ | $D_1;D_2$

6. A second example: the Language SMALL

The language SMALL described in this chapter has two roles:

(i) To illustrate the main ideas of standard semantics.
(ii) To provide a 'kernel language' which we shall extend as we discuss the semantics of various constructs in later chapters.

6.1. Syntax of SMALL

6.1.1. Syntactic domains

The *primitive* syntactic domains of SMALL are:

Ide	The domain of identifiers **I**
Bas	The domain of basic constants **B**
Opr	The domain of binary operators **O**

We do not further specify these primitive domains. **Bas** could, for example, be {**0, 1, 2, ...**} and **Opr** could be {+, -, ×, /, <, >, ...}.

The *compound* syntactic domains of SMALL are:

Pro	The domain of programs **P**
Exp	The domain of expressions **E**
Com	The domain of commands **C**
Dec	The domain of declarations **D**

These domains are defined by the following syntactic clauses:

6.1.2. Syntactic clauses

P ::= program C

E ::= B | true | false | read | I | E_1(E_2)
| if E then E_1 else E_2 | E_1 O E_2

C ::= E_1 := E_2 | output E | E_1(E_2) | if E then C_1 else C_2
| while E do C | begin D;C end | C_1;C_2

D ::= const I = E | var I = E | proc I(I_1);C | fun I(I_1);E | D_1;D_2

We have procedure (and function) calls of the form **E_1(E_2)** rather than just **I(E)** to permit calls like **(if E_1 then I_1 else I_2)(E_2)** in which either the procedure (or function) denoted by **I_1** or the one denoted by **I_2** is applied to **E_2** depending on the value of **E_1**. Similarly we permit assignments of the form **(if E_1 then I_1 else I_2) := E_2** or even **I(E_1) := E_2** (in which the location assigned to is the result of the function call **I(E_1)**).

6.2. Semantics of SMALL

6.2.1. Semantic domains

The primitive semantic domains of SMALL are:

Num	The domain of numbers **n**.
Bool	The domain of bodeans **b**.
Loc	The domain of locations **ι**.
Bv	The domain of basic values **e**.

Basic values are the denotations of basic constants. If **Bas** = {**0, 1, 2, ...**} then **Bv** = {**0, 1, 2, ...**}—however for simplicity we shall not specify any particular choice of basic values or constants.

The compound semantic domains are defined by the following domain equations:

Dv	= **Loc + Rv + Proc + Fun**	—denotable values **d**
Sv	= **File + Rv**	—storable values **v**
Ev	= **Dv**	—expressible values **e**
Rv	= **Bool + Bv**	—R-values **e**
File	= **Rv***	—files **i**
Env	= **Ide→[Dv + {unbound}]**	—environments **r**
Store	= **Loc→[Sv + {unused}]**	—stores **s**
Cc	= **Store→Ans**	—command continuations **c**
Ec	= **Ev→Cc**	—expression continuations **k**
Dc	= **Env→Cc**	—declaration continuations **u**
Proc	= **Cc→Ec**	—procedure values **p**
Fun	= **Ec→Ec**	—function values **f**
Ans	= {**error, stop**} + [**Rv × Ans**]	—final answers **a**

We assume **Loc** contains a location **input** which holds the input file. We

have not provided SMALL with any facilities for creating new files, this is discussed in 9.5. For the time being the only file around is the input.

6.2.2. Semantic functions

We assume as given:

B:Bas→Bv
O:Opr→[Rv × Rv]→Ec→Cc

For example if **Bas** = {**0, 1, 2, ...**}, **Opr** = {+, ×, -, /, ...} and **Bv** = **Num** then we might have **B[0]** = **0**, **O[+](1, 2)k** = **k 3**, **O[/](1, 0)k** = **err** etc.

The meaning of the non-basic constructs are given by:

P:Pro→File→Ans
R:Exp→Env→Ec→Cc
E:Exp→Env→Ec→Cc
C:Com→Env→Cc→Cc
D:Dec→Env→Dc→Cc

These are defined by the following semantic clauses.

6.2.3. Semantic clauses

6.2.3.1. Programs

(P) **P[program C] i = C[C]( ) (λs . stop)(i/input)**

The final answer produced when **program C** is run with input **i** is obtained by evaluating the command **C** with an empty environment (**( ) = λl . unbound**), a store in which the only used location is **input** which has contents **i** (**(i/input) = λι . ι = input→i, unused**) and a continuation which when sent a store stops with final answer **stop**.

6.2.3.2. Expressions

(R) **R[E] r k = E[E] r;deref;Rv?;k**

E is evaluated, its result dereferenced and then checked to make sure it is an R-value and then passed to the rest of the program.

(E1) **E[B] r k = k(B[B])**

B[B] is passed to the rest of the program.

(E2) **E[true] r k = k true , E[false] r k = k false**

The appropriate boolean value is passed to the rest of the program.

(E3) **E[read] r k s =**
null(s input)→error, k(hd(s input))(s[tl(s input)/input])

If the input file (i.e. **s input**) is empty then an error occurs, otherwise the first item on the input (**hd(s input)**) is sent to the rest of the program **k** together with a store in which the item first read has been removed from the input file (**s[tl(s input)/input]**).

(E4) **E[I] r k = (r I = unbound)→err, k(r I)**

If **I** is unbound an error occurs, otherwise the value denoted by **I** in the environment is sent to the rest of the program.

(E5) **E[E$_1$(E$_2$)] r k = E[E$_1$] r;Fun?λf . E[E$_2$] r;f;k**

E$_1$ is evaluated and its value checked to ensure it is a function **f**, **E$_2$** is then evaluated and its value passed to **f** and finally the result of **f** is passed to the rest of the program **k**.

(E6) **E[if E then E$_1$ else E$_2$] r k =**
R[E] r;Bool?;cond(E[E$_1$] r k,E[E$_2$] r k)

E is evaluated for its R-value which is then checked to ensure it is a boolean, then **E$_1$** or **E$_2$** are evaluated depending on whether **E**'s value was **true** or **false**.

(E7) **E[E$_1$OE$_2$] r k =**
R[E$_1$] rλe$_1$. R[E$_2$] rλe$_2$. O[O](e$_1$,e$_2$)k

E$_1$ is evaluated for its R-value **e$_1$**, then **E$_2$** is evaluated for its R-value **e$_2$** and finally the result of doing **O** to **e$_1$** and **e$_2$** are sent to the rest of the program **k**.

6.2.3.3. Commands

(C1) **C[E$_1$:= E$_2$] r c =**
E[E$_1$] r;Loc? λι . R[E$_2$] r;update ι;c

$\mathbf{E}_1$ is evaluated for its L-value ι, then $\mathbf{E}_2$ is evaluated for its R-value which is then stored in ι and the resulting store passed to the rest of the program **c**.

(C2) $\mathbf{C[output\ E]\,r\,c} =$
$\mathbf{R[E]\,r\lambda e\,s\,.\,(e,\,c\,s)}$

E is evaluated for its R-value **e** and new store **s**, then **e** is put onto the front of the answer produced when the rest of the program **c** is passed **s**.

(C3) $\mathbf{C[E_1(E_2)]\,r\,c} =$
$\mathbf{E[E_1]\,r;Proc?\lambda p\,.\,E[E_2]\,r;p;c}$

$\mathbf{E}_1$ is evaluated and its value checked to ensure it is a procedure **p**, $\mathbf{E}_2$ is then evaluated and its value passed to **p** and finally the store resulting from **p** is passed to the rest of the program **c**.

(C4) $\mathbf{C[if\ E\ then\ C_1\ else\ C_2]\,r\,c} =$
$\mathbf{R[E]\,r;Bool?;cond(C[C_1]\,r\,c,C[C_2]\,r\,c)}$

E is evaluated for its R-value which is then checked to ensure it is a boolean, then $\mathbf{C}_1$ or $\mathbf{C}_2$ are evaluated depending on whether E's value was **true** or **false**.

(C5) $\mathbf{C[while\ E\ do\ C]\,r\,c} =$
$\mathbf{R[E]\,r;Bool?;cond(C[C]\,r;C[while\ E\ do\ C]\,r\,c\,,\,c)}$

E is evaluated for its R-value which is then checked to ensure it is a boolean, then if **E**'s value is **true** **C** is evaluated and the resulting store passed to the beginning of **while E do C** again. If **E**'s value is **false** then the rest of the program is immediately done.

(C6) $\mathbf{C[begin\ D;C\ end]\,r\,c = D[D]\,r\lambda r'\,.\,C[C]\,r[r']\,c}$

D is evaluated to get a little environment $\mathbf{r}'$, then **C** is evaluated in **r** updated with $\mathbf{r}'$ and then the rest of the program **c** is done.

(C7) $\mathbf{C[C_1;C_2]\,r\,c = C[C_1]\,r;C[C_2]\,r;c}$

$\mathbf{C}_1$ is done, then $\mathbf{C}_2$ is done and then the rest of the program **c** is done.

6.2.3.4. Declarations

(D1) $\mathbf{D[const\ I = E]\,r\,u = R[E]\,r\lambda e\,.\,u(e/I)}$

E is evaluated for its R-value **e** and then the little environment **e/I** is passed to the rest of the program **u**.

(D2) $\mathbf{D[var\ I = E]\,r\,u = R[E]\,r;ref\,\lambda\iota\,.\,u(\iota/I)}$

E is evaluated for its R-value which is then stored in a new location ι and the little environment ι/I passed to the rest of the program **u**.

(D3) $\mathbf{D[proc\ I(I_1);C]\,r\,u = u((\lambda c\,e\,.\,C[C]\,r[e/I_1]\,c)/I)}$

A little environment in which the procedure value $\lambda c\,e\,.\,C[C]\,r[e/I_1]\,c$ is bound to **I** is passed to the rest of the program **u**. When this procedure value is called the body **C** is evaluated in an environment got from the declaration time environment **r** by binding the actual parameter **e** to the formal parameter $\mathbf{I}_1$.

(D4) $\mathbf{D[fun\ I(I_1);E]\,r\,u = u((\lambda k\,e\,.\,E[E]\,r[e/I_1]\,k)/I)}$

A little environment in which the function value $\lambda k\,e\,.\,E[E]\,r[e/I_1]\,k$ is bound to **I** is passed to the rest of the program **u**.

(D5) $\mathbf{D[D_1;D_2]\,r\,u = D[D_1]\,r\lambda r_1\,.\,D[D_2]\,r[r_1]\lambda r_2\,.\,u(r_1[r_2])}$

$\mathbf{D}_1$ is evaluated in **r** to get a little environment $\mathbf{r}_1$, then $\mathbf{D}_2$ is evaluated in $\mathbf{r}[\mathbf{r}_1]$ (i.e. **r** updated with the bindings in $\mathbf{r}_1$) to get another little environment $\mathbf{r}_2$ and then finally $\mathbf{r}_1[\mathbf{r}_2]$ (the bindings from $\mathbf{D}_1$ updated with those of $\mathbf{D}_2$) are passed to the rest of the program **u**. Notice that $\mathbf{D}_2$ is evaluated in an environment which contains the effects of $\mathbf{D}_1$ and that if $\mathbf{D}_1$ and $\mathbf{D}_2$ contain declarations of the same identifier then the result of $\mathbf{D}_1;\mathbf{D}_2$ is the result of $\mathbf{D}_2$. For example **const x = 1;const x = x+1** would bind **x** to **2**.

6.3. A worked example

To illustrate how the semantics of SMALL works we show how to use it to 'evaluate' **program begin var x = read;output x end** with respect to an initial input file **i** such that **null i = false** and **hd i = 1** (for example **i = 1.2.3...**). We show as expected that

$\mathbf{P[program\ begin\ var\ x = read;output\ x\ end]\,i = (1,\,stop)}$
i.e. $\mathbf{C[begin\ var\ x = read;output\ x\ end]()(\lambda s\,.\,stop)(i/input) = (1,\,stop)}$

We do the calculation in several stages, first let $\mathbf{r} = ()$, $\mathbf{s}_1 = \mathbf{(i/input)}$, $\mathbf{s}_2 = \mathbf{s}_1[\mathbf{tl\,i/input}] = \mathbf{(tl\,i/input)}$, assume $\mathbf{\iota}_2 = \mathbf{new\,s}_2$ and finally let $\mathbf{s}_3 = \mathbf{s}_2[1/\iota_2]$. Then:

(i) $\mathbf{E[read]\,r\,k\,s}_1$

$= \mathbf{null(s}_1\,\mathbf{input)\rightarrow error,\ k(hd(s}_1\,\mathbf{input))(s}_1[\mathbf{tl(s}_1\,\mathbf{input})/\mathbf{input}])$ (E3)

$= \mathbf{null\,i\rightarrow error,\ k\,1\,(s}_1[\mathbf{tl\,i/input}])$ $(\mathbf{s}_1\,\mathbf{input} = \mathbf{i})$

$= \mathbf{k\,1\,s}_2$ $(\mathbf{null\,i} = \mathbf{false})$

(ii) $\mathbf{D[var\,x = read]\,r\,u\,s}_1$

$= \mathbf{(R[read]\,r;ref}\,\lambda\mathbf{\iota\,.\,u(\iota/x))\,s}_1$ (D2)

$= \mathbf{(E[read]\,r;deref;Rv?;ref}\,\lambda\mathbf{\iota\,.\,u(\iota/x))\,s}_1$ (R)

$= \mathbf{E[read]\,r(deref(Rv?(ref}\,\lambda\mathbf{\iota\,.\,u(\iota/x))))\,s}_1$ (definition of ;)

$= \mathbf{deref(Rv?(ref}\,\lambda\mathbf{\iota\,.\,u(\iota/x)))\,1\,s}_2$ (by (i) above)

$= \mathbf{isLoc\,1\rightarrow\ldots,Rv?(ref}\,\lambda\mathbf{\iota\,.\,u(\iota/x))\,1\,s}_2$ (definition of **deref**) $(\mathbf{isLoc\,1} = \mathbf{false})$

$= \mathbf{Rv?(ref}\,\lambda\mathbf{\iota\,.\,u(\iota/x))\,1\,s}_2$ (definition of **Rv?**)

$= \mathbf{isRv\,1\rightarrow[ref}\,\lambda\mathbf{\iota\,.\,u(\iota/x))\,1\,s}_2\mathbf{,\ldots}$ $(\mathbf{isRv\,1} = \mathbf{true})$

$= \mathbf{ref}\,(\lambda\mathbf{\iota\,.\,u(\iota/x))\,1\,s}_2$

$= \mathbf{new\,s}_2 = \mathbf{error\rightarrow error,}$

$\mathbf{update\,(new\,s}_2\mathbf{)((}\lambda\mathbf{\iota\,.\,u(\iota/x))(new\,s}_2\mathbf{))\,1\,s}_2$ (definition of **ref**)

$= \mathbf{update}\,\iota_2\,\mathbf{(u(}\iota_2\mathbf{/x))\,1\,s}_2$ $(\mathbf{new\,s}_2 = \iota_2)$

$= \mathbf{isSv\,1\rightarrow u(}\iota_2\mathbf{/x)(s}_2[1/\iota_2]\mathbf{),\ error}$ (definition of **update**)

$= \mathbf{u(}\iota_2\mathbf{/x)(s}_2[1/\iota_2]\mathbf{)}$ $(\mathbf{isSv\,1} = \mathbf{true})$

$= \mathbf{u(}\iota_2\mathbf{/x)s}_3$ $(\mathbf{s}_3 = \mathbf{s}_2[1/\iota_2])$

(iii) $\mathbf{C[begin\,var\,x = read;output\,x\,end]\,r\,c\,s}_1$

$= \mathbf{D[var\,x = read]\,r(}\lambda\mathbf{r'\,.\,C[output\,x]\,r[r']\,c)\,s}_1$ (C6)

$= (\lambda\mathbf{r'\,.\,C[output\,x]\,r[r']\,c)\,(\iota}_2\mathbf{/x)\,s}_3$ (by (ii) above)

$= \mathbf{C[output\,x]\,r[}\iota_2\mathbf{/x]\,c\,s}_3$

$= \mathbf{R[x]\,r[}\iota_2\mathbf{/x]\,(}\lambda\mathbf{e\,s\,.\,(e,c\,s))\,s}_3$ (C2)

$= \mathbf{E[x]\,r[}\iota_2\mathbf{/x]\,(deref(Rv?}\,\lambda\mathbf{e\,s\,.\,(e,c\,s)))\,s}_3$ (R)

$= \mathbf{deref(Rv?}\,\lambda\mathbf{e\,s\,.\,(e,c\,s))}\,\iota_2\,\mathbf{s}_3$ ((E4), $\mathbf{r[}\iota_2\mathbf{/x]x} = \iota_2$)

$= \mathbf{isLoc\,\iota\rightarrow cont(Rv?}\,\lambda\mathbf{e\,s\,.\,(e,c\,s))}\,\iota_2\,\mathbf{s}_3\mathbf{,\ldots}$ (definition of **deref**)

$= \mathbf{cont(Rv?}\,\lambda\mathbf{e\,s\,.\,(e,c\,s))}\,\iota_2\,\mathbf{s}_3$

$= \mathbf{(Rv?}\,\lambda\mathbf{e\,s\,.\,(e,c\,s))\,1\,s}_3$ (definition of **cont**, $\mathbf{isLoc}\,\iota_2 = \mathbf{true}$ and $\mathbf{s}_3\,\iota_2 = \mathbf{1}$)

$= (\lambda\mathbf{e\,s\,.\,(e,c\,s))\,1\,s}_3$ (definition of **Rv?**, $\mathbf{isRv\,1} = \mathbf{true}$)

$= \mathbf{(1,c\,s}_3\mathbf{)}$

(iv) $\mathbf{P[program\,begin\,var\,x = read;output\,x\,end]\,i}$

$= \mathbf{C[begin\,var\,x = read;output\,x\,end]()(}\lambda\mathbf{s\,.\,stop)(i/input)}$ (P)

$= \mathbf{(1,(}\lambda\mathbf{s\,.\,stop)s}_3\mathbf{)}$ (by (iii) above)

$= \mathbf{(1,stop)}$

Calculations like this, though very tedious, are purely mechanical. The Semantics Implementation System (SIS) of Peter Mosses [Mosses] is a computer system which, by automating such calculations, runs programs directly from a denotational semantics. Although the resulting 'implementation' is *very* inefficient it is nevertheless useful for 'debugging semantics' and as an aid to language designers. An implementation package (on DEC tapes) is currently available for PDP 10's from Peter Mosses at Aarhus University, Denmark.