A new approach to operational semantics by categories

William Steingartner

william.steingartner@tuke.sk

Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice, Slovakia

Semantics of programming languages

Spring 2022/2023

Theme 07 – Semantics in categories

Formal semantics

- provides unambiguous meaning of programs written in programming language;
- helps designers to prepare good and useful programming languages;
- serves for implementators to write correct compilers;
- encourages users/programmers how to use language constructions properly.

Semantic methods

- denotational semantics;
- operational semantics;
- natural semantics;
- axiomatic semantics;
- action semantics;
- game semantics.

Categories

- mathematical structures consisting of objects and morphisms between them;
- objects can be various mathematical structures, data structures, types;
- categories have become useful for modeling computations, processes, programs, program systems;
- are basic structures for coalgebraic behavioural models.

Categories in teaching

- quite simple mathematical structures;
- graphical representations useful for illustration of examples;
- understandable for our students.

Category theory

Original purpose of categories

in theyears 1942-1945 - in topology, especially algebraic topology, geometry (Samuel Eilenberg, Saunders MacLane)

Application of categories

- algebraic topology;
- geometry;
- physics; ...

In informatics

- expressing the models of computation;
- definition of semantics;
- definition of types and work with them;
- models of logic; ...

Definition of category

Category

- $Ob(\mathscr{C})$, objects of category $\mathscr{C}$: $A, B, \ldots$;
- $Morph(\mathscr{C})$, morphisms of category $\mathscr{C}$: $f : A \rightarrow B$;
- identical morphism for each object $\mathscr{C}$, $id_A : A \rightarrow A$;
- the composition of morphisms: for $f : A \rightarrow B$ and $g : B \rightarrow C$ is defined $g \circ f : A \rightarrow C$.

For each category are defined the following conditions:

1. the domain of composition of morphisms $f \circ g$ is the domain of f and codomain of composition $f \circ g$ is codomain of g;
2. composition of morphisms is *associative*, i.e. $h \circ (g \circ f) = (h \circ g) \circ f$;
3. the domain and codomain of identity id_A is an object A;
4. if $f : A \rightarrow B$ is a morphism, then it holds $f \circ id_A = id_B \circ f$.

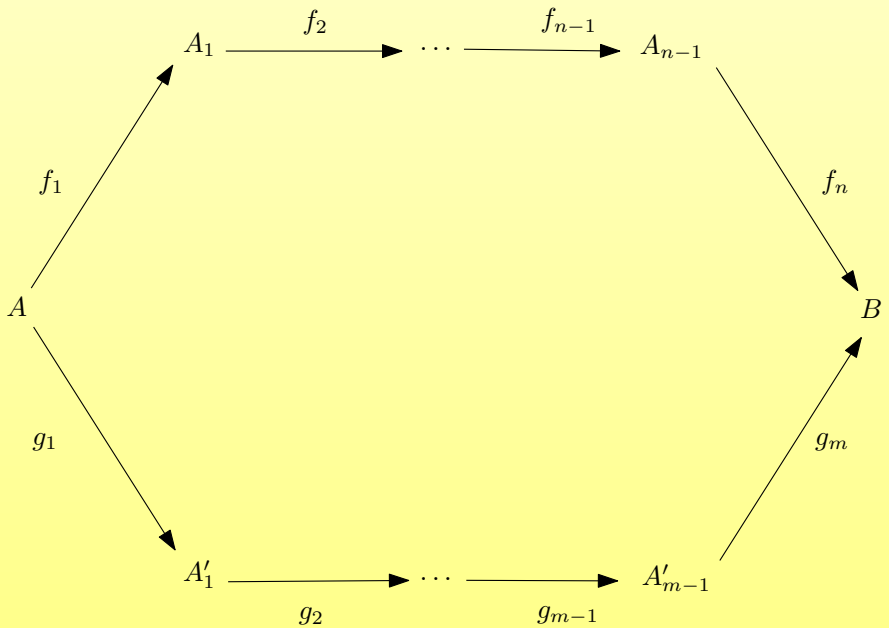
The commutative diagrams

The commutative diagram is a structure for graphical expressing of equalities in categories. Let A and B be the objects of category and let $\mathcal{D}$ be the diagram of category $\mathscr{C}$. The diagram $\mathcal{D}$ we call *commutative diagram*, if the all paths from A to B composed of morphisms are equal; it means that for the morphisms

$$f_i : A_{i-1} \to A_i \quad \text{and} \quad g_j : A'_{j-1} \to A'_j$$

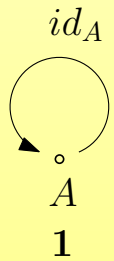
for $i = 1, \ldots, n$, $j = 1, \ldots, m$, where $A = A_0 = A'_0$ a $B = A_n = A'_m$ it holds

$$f_n \circ \ldots \circ f_2 \circ f_1 \quad = \quad g_m \circ \ldots \circ g_2 \circ g_1.$$

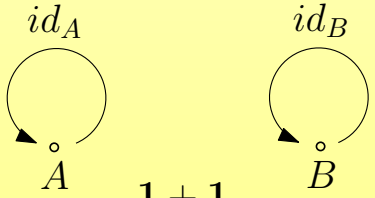


Simple categories

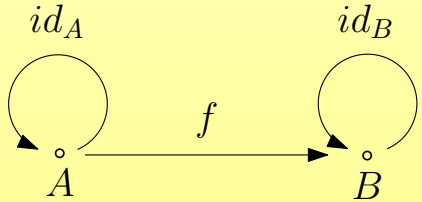
- *Empty category* $\emptyset$ - no objects, no morphisms.
- $\mathbf{1}$ category with only one object and one morphism - an identity on existing category object.
- $\mathbf{1}+\mathbf{1}$ category with two objects and two morphisms - identities on particular objects.
- $\mathbf{2}$ category with two objects, two identity morphisms and one morphism between category objects.



$\mathbf{1}$



$\mathbf{1}+\mathbf{1}$



$\mathbf{2}$

Category of sets

$\mathscr{S}et$

- objects are sets, $Ob(\mathscr{S}et) = \{A, B, \mathbb{N}, \ldots\}$;
- morphisms are functions, $f : A \to B$, $\ln : \mathbb{R}^+ \to \mathbb{R}$ etc.;
- identity is defined on each set, $id_A : A \to A$;
- composition of morphisms is a composition of functions: for $f : A \to B$ and $g : B \to C$ exists a new morphism $g \circ f : A \to C$:

$$(g \circ f)(x) = g(f(x))$$

for $x \in A$.

Functor

Functor

Functor $F : \mathscr{C} \to \mathscr{D}$ is defined as tuple of functions (F_0, F_1)

$$F_0 : \mathscr{C}_{obj} \to \mathscr{D}_{obj}$$
$$F_1 : \mathscr{C}_{morp} \to \mathscr{D}_{morp},$$

with the following conditions:

- if $f : A \to B$ is a morphism in $\mathscr{C}$, then $F_1(f) : F_0(A) \to F_0(B)$ v $\mathscr{D}$;
- for each object A in $\mathscr{C}$ holds $F_1(id_A) = id_{F_0(A)}$;
- if $f \circ g$ is a composition in $\mathscr{C}$, then the composition $F_1(f) \circ F_1(g)$ is defined in $\mathscr{D}$ and it holds $F_1(f \circ g) = F_1(f) \circ F_1(g)$.

$$F \circ f = F(f) \circ F$$

Categorical semantics

- denotational semantics uses category of types where objects are types and morphisms are functions;
- algebraic semantics uses institutions as complex structures based on categories of signatures;
- game semantics uses category of arenas.

Why categorical operational semantics

- provides illustrative view of dynamics of states;
- provides simply understandable mathematical model of programs;
- appropriate for informaticians writing compilers;
- serves for creating skills to work with formal methods.

Basic ideas of our approach

Construction of category of states

- we consider simple imperative language;
- our language has only two implicit types;
- we do not consider exception, jumps and recursion;
- so simplified model is understandable without losing exactness.

Language $\mathscr{J}ane$

- consists of traditional syntactic constructions of imperative languages;
- for defining formal syntax of $\mathscr{J}ane$ the following syntactic domains are introduced:

$n \in \mathbf{Num}$	- for digit strings;
$x \in \mathbf{Var}$	- for variable names;
$e \in \mathbf{Expr}$	- for arithmetic expressions;
$b \in \mathbf{Bexpr}$	- for boolean expressions;
$S \in \mathbf{Statm}$	- for statements;
$D \in \mathbf{Decl}$	- for sequences of variable declarations.

Language $\mathscr{J}ane$ - Syntax

The elements $n \in \mathbf{Num}$ and $x \in \mathbf{Var}$ have no internal structure from semantic point of view.

The syntactic domain $\mathbf{Expr}$ consists of all well-formed arithmetic expressions created by the following production rule

$$e ::= n \mid x \mid e + e \mid e - e \mid e * e.$$

Boolean expression from $\mathbf{Bexpr}$ can be of the following structure:

$$b ::= \texttt{false} \mid \texttt{true} \mid e = e \mid e \leq e \mid \neg b \mid b \wedge b.$$

The variables used in programs have to be declared. We consider $D \in \mathbf{Decl}$ as a sequence of declarations:

$$D ::= \texttt{var}\ x; D \mid \varepsilon.$$

As the statements $S \in \mathbf{Statm}$ we consider five Dijkstra's statements together with block statement and input statement:

$$S ::= x := e | \texttt{skip} | S; S | \texttt{if}\ b\ \texttt{then}\ S\ \texttt{else}\ S | \texttt{while}\ b\ \texttt{do}\ S | \texttt{begin}\ D; S\ \texttt{end} | \texttt{input}\ x.$$

Specification of states

State

- can be considered as some abstraction of computer memory;
- change of state means change of value in memory;
- because of block structure of $\mathscr{J}ane$, we have to consider also a level of block nesting;
- every variable occurring in a program has to be allocated;

The signature Σ_{State} for states

$$\Sigma_{State} = \begin{array}{ll} \underline{types:} & State, Var, Value \\ \underline{opns:} & init :\rightarrow State \\ & alloc : var, State \rightarrow State \\ & get : Var, State \rightarrow Value \\ & del : State \rightarrow State \end{array}$$

Specification of states

Representation

The representation of the elements of the type $Value$ we consider the set of integer numbers:

$$\mathbf{Value} = \mathbb{Z}.$$

For undefined values we use the symbol $\perp$.

Type Var is represented by set $\mathbf{Var}$ of variable names.

Levels of declaration l are denoted by natural numbers:

$$l \in \mathbf{Level}, \quad \mathbf{Level} = \mathbb{N}.$$

Operational semantics

Operational model

- we construct operational model of $\mathscr{J}ane$ as the category $\mathscr{C}_{State}$ of states;
- we assign to states their representation;
- because of block structure of $\mathscr{J}ane$, we have to consider also a level of block nesting ($l \in \mathbf{Level}, \mathbf{Level} = \mathbb{N}$);
- representation of type $State$ has to express variable, its value with respect to actual nesting level;

State representation

Sequence

$$s : \mathbf{Var} \times \mathbf{Level} \rightharpoonup \mathbf{Value}$$

Every state s can be expressed as a sequence of ordered pairs $((x, l), v)$:

$$s = \langle ((x, 1), v_1), \ldots, ((z, l), v_n) \rangle$$

Table

variable	level	value
x	1	v_1
$\vdots$		
z	l	v_n

Representation of operations

The operation $[\![init]\!]$

$$[\![init]\!] = s_0 = \langle((\bot, 1), \bot)\rangle$$

creates the initial state of a program, with no declared variable.

variable	level	value
$\bot$	1	$\bot$

The operation $[\![alloc]\!]$

$$[\![alloc]\!](x, s) = s \diamond ((x, l), \bot),$$

sets actual nesting level to declared variable. Because of undefined value of declared variable, the operation $[\![alloc]\!]$ does not change the state.

variable	level	value
$\vdots$	$\vdots$	$\vdots$
x	l	$\bot$

Representation of operations

The operation $[\![get]\!]$ returns a value of a variable declared on the highest nesting level,

$$[\![get]\!](x, \langle \ldots, ((x, l_i), v_i), \ldots, ((x, l_k), v_k), \ldots \rangle) = v_k,$$

where $l_i < l_k$.

The operation $[\![del]\!]$ deallocates (forgets) all variables declared on the highest nesting level l_j:

$$[\![del]\!](s \diamond \langle ((x_i, l_j), v_k), \ldots, ((x_n, l_j), v_m) \rangle) = s.$$

variable	level	value
$\vdots$	$\vdots$	$\vdots$
x	l_i	v
x_i	l_j	v_k
$\vdots$	$\vdots$	$\vdots$
x_n	l_j	v_m

Arithmetic expressions

Arithmetic expressions

$$[\![e]\!] : \mathbf{State} \to \mathbf{Value}.$$

$$[\![n]\!]s = \mathbf{n}$$

$$[\![x]\!]s = [\![get]\!](x, s)$$

$$[\![e_1 + e_2]\!]s = [\![e_1]\!]s \oplus [\![e_2]\!]s$$

$$[\![e_1 - e_2]\!]s = [\![e_1]\!]s \ominus [\![e_2]\!]s$$

$$[\![e_1 * e_2]\!]s = [\![e_1]\!]s \otimes [\![e_2]\!]s$$

$$\mathbf{Value} = \mathbb{Z}$$

Boolean expressions

Boolean expressions

$$[\![b]\!] : \mathbf{State} \to \mathbf{Bool}$$

$$[\![\mathtt{true}]\!]s = \mathbf{true}$$

$$[\![\mathtt{false}]\!]s = \mathbf{false}$$

$$[\![e_1 = e_2]\!]s = \begin{cases} \mathbf{true} & \text{if } [\![e_1]\!]s = [\![e_2]\!]s \\ \mathbf{false} & \text{otherwise} \end{cases}$$

$$[\![e_1 \leq e_2]\!]s = \begin{cases} \mathbf{true} & \text{if } [\![e_1]\!]s \leq [\![e_2]\!]s \\ \mathbf{false} & \text{otherwise} \end{cases}$$

$$[\![\neg b]\!]s = \begin{cases} \mathbf{true} & \text{if } [\![\neg b]\!]s = \mathbf{false} \\ \mathbf{false} & \text{otherwise} \end{cases}$$

$$[\![b_1 \wedge b_2]\!]s = \begin{cases} \mathbf{true} & \text{if } [\![b_1]\!]s = [\![b_2]\!]s = \mathbf{true} \\ \mathbf{false} & \text{otherwise} \end{cases}$$

$$\mathbf{Bool} = \mathbb{B}$$

Declarations

Declarations

A declaration

$$\texttt{var}\ x$$

is represented as an endomorphism:

$$[\![\ ]\!]_D : s \to s$$

for a given state s and defined by

$$[\![\texttt{var}\ x]\!]s = [\![alloc]\!](x, s).$$

A sequence of declarations

$$[\![\texttt{var}\ x; D]\!]s = [\![D]\!] \circ [\![alloc(x, s)]\!].$$

A declaration creates new entry for declared variable with the actual level of nesting and undefined value

$$((x, l), \bot).$$

Statements

$$[\![S]\!] : s \to s'$$

$$[\![x := e]\!]s = \begin{cases} s' = s\left[\left((x,l),v\right) \mapsto \left((x,l),[\![e]\!]\right)s\right] & \text{for } \left((x,l),v\right) \in s; \\ \bot & \text{otherwise.} \end{cases}$$

$$[\![\texttt{skip}]\!] = id_s, \qquad [\![\texttt{skip}]\!]s = s$$

$$[\![S_1; S_2]\!] = [\![S_2]\!] \circ [\![S_1]\!], \qquad [\![S_1; S_2]\!]s = [\![S_2]\!]\left([\![S_1]\!]\right)s$$

$$[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]s = \begin{cases} [\![S_1]\!]s & \text{if } [\![b]\!]s = \textbf{true}; \\ [\![S_2]\!]s & \text{otherwise.} \end{cases}$$

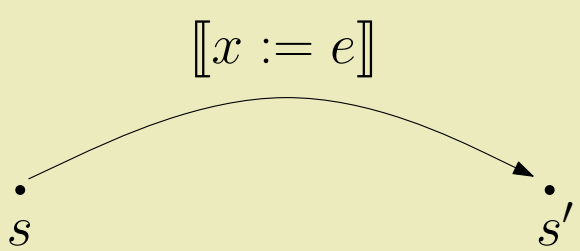
$$[\![\texttt{while } b \texttt{ do } S]\!]s = [\![\texttt{if } b \texttt{ then } (S; \texttt{while } b \texttt{ do } S) \texttt{ else skip}]\!]$$

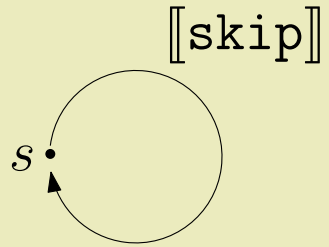
$$[\![\texttt{input } x]\!]s = \begin{cases} s' = s\left[\left((x,l),v\right) \mapsto \left((x,l),v'\right)\right] & \text{for } \left((x,l),v'\right) \in s; \\ \bot & \text{otherwise.} \end{cases}$$

Statements

$$[\![S]\!] : s \to s'$$

$$[\![x := e]\!]s = \begin{cases} s\left[\left((x,l),v\right) \mapsto \left((x,l),[\![e]\!]s\right)\right], & \text{for } \left((x,l),v\right) \in s \\ s_\perp, & \text{otherwise} \end{cases}$$





The notation

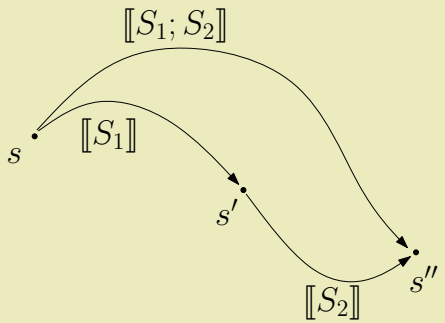
$$s' = s\left[\left((x,l),v\right) \mapsto \left((x,l),[\![e]\!]s\right)\right]$$

describes a new state s' that is an actualization of the state s in its entry for the declared variable x whose value is changed to $[\![e]\!]s$.

$$[\![\texttt{skip}]\!] = id_s, \qquad [\![\texttt{skip}]\!]s = s$$

Statements

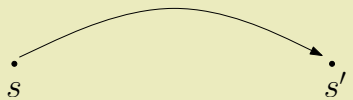
$$[\![S_1; S_2]\!] = [\![S_2]\!] \circ [\![S_1]\!], \qquad [\![S_1; S_2]\!]s = [\![S_2]\!]\,([\![S_1]\!])\,s$$



If the state s is undefined, i.e. $s = s_\perp$, then execution of any statement in this undefined state provides also undefined state:

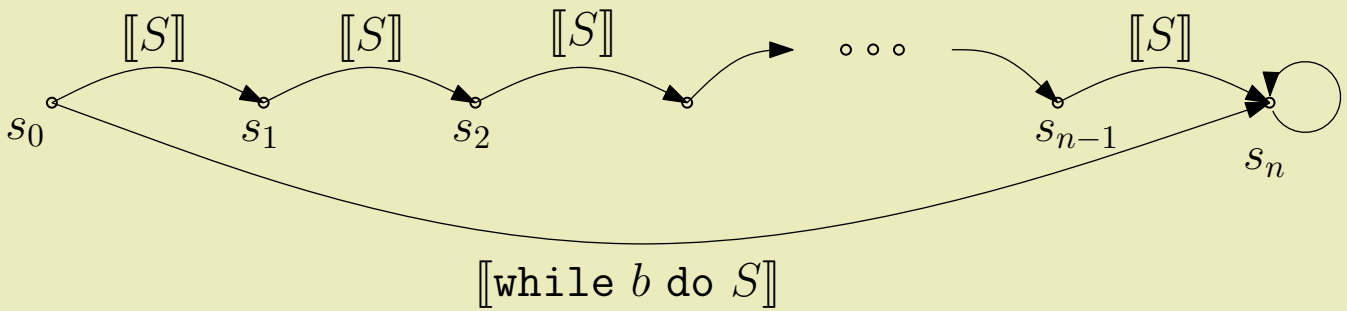
$$[\![S]\!]s_\perp = s_\perp.$$

$$[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]s = \begin{cases} [\![S_1]\!]s, & \text{if } [\![b]\!]s = \mathbf{true} \\ [\![S_2]\!]s, & \text{otherwise} \end{cases}$$



Statements

$$[\![\texttt{while}\ b\ \texttt{do}\ S]\!]s = [\![\texttt{if}\ b\ \texttt{then}\ (S;\texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ \texttt{skip}]\!]s$$



$$[\![\texttt{input}\ x]\!]s = \begin{cases} s\left[((x,l),v) \mapsto ((x,l),v')\right], & \text{for } ((x,l),v) \in s; \\ s_\perp & \text{otherwise.} \end{cases}$$

Block statement

$$\texttt{begin}\ D; S\ \texttt{end}$$

The following is a summary of the four steps used to execute of unnamed blocks:

- nesting level l is incremented. We represent this step by fictive entry in state table

  $$\left((\texttt{begin}, l+1)\,, \bot\right)$$

  i.e. endomorphism $s \to s$;
- local declarations are elaborated on nesting level $l+1$;
- the body S of block is executed;
- locally declared variables are forgotten at the end of block. We model this situation using operation $[\![del]\!]$.

The semantics:

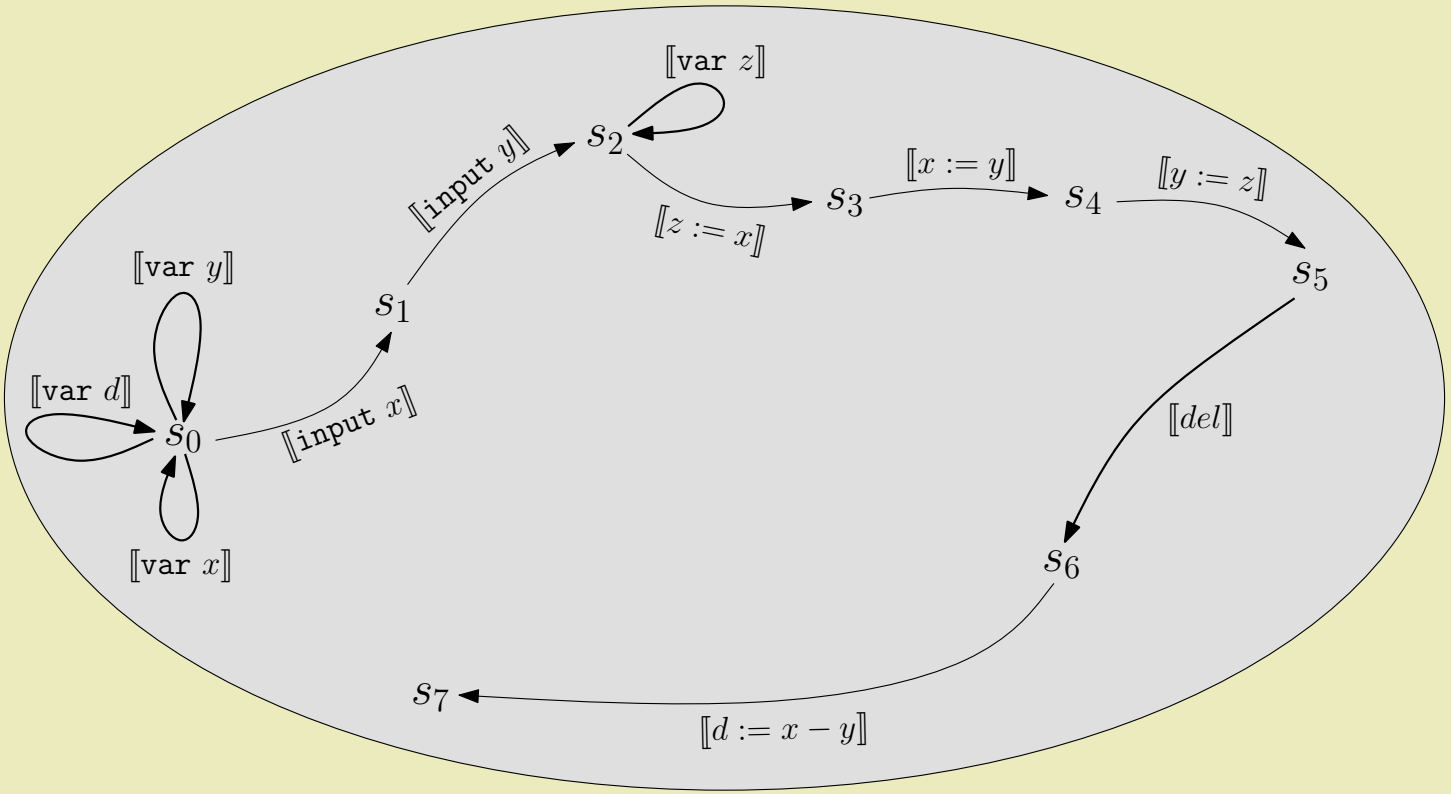
$$[\![\texttt{begin}\ D; S\ \texttt{end}]\!]s = [\![del]\!] \circ [\![S]\!] \circ [\![D]\!](s \diamond \langle((\texttt{begin}, l+1), \bot)\rangle)$$

Example 1

```
var x; var y; var d;
input x;
input y;
if (x ≤ y) then
        begin
            var z;
            z := x;
            x := y;
            y := z;
        end;
else
    skip;
d := x − y;
```

We assume that user inputs value **2** into variable x and value **7** into variable y.

Categorical representation of program



States during program execution

s_0		
x	1	$\perp$
y	1	$\perp$
d	1	$\perp$

$a)$

s_1		
x	1	**2**
y	1	$\perp$
d	1	$\perp$

$b)$

s_2		
x	1	**2**
y	1	**7**
d	1	$\perp$
begin	2	$\perp$
z	2	$\perp$

$c)$

s_3		
x	1	**2**
y	1	**7**
d	1	$\perp$
begin	2	$\perp$
z	2	**2**

$d)$

s_4		
x	1	**7**
y	1	**7**
d	1	$\perp$
begin	2	$\perp$
z	2	**2**

$e)$

s_5		
x	1	**7**
y	1	**2**
d	1	$\perp$
begin	2	$\perp$
z	2	**2**

$f)$

s_6		
x	1	**7**
y	1	**2**
d	1	$\perp$

$g)$

s_7		
x	1	**7**
y	1	**2**
d	1	**5**

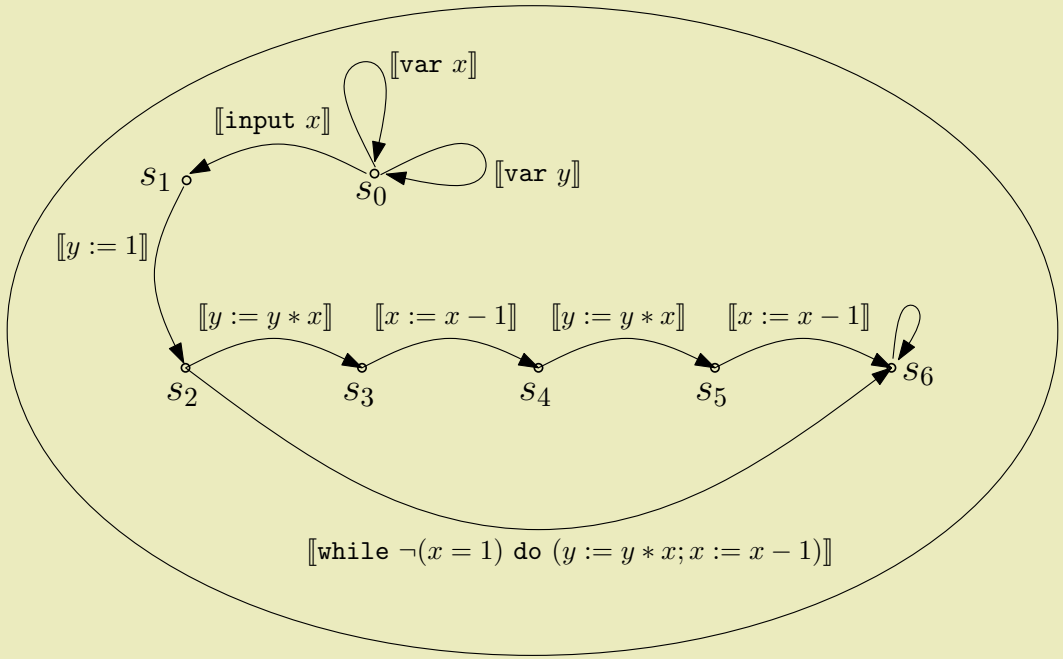
$h)$

Example 2

$$\texttt{var } x; \texttt{var } y;$$
$$\texttt{input } x;$$
$$y := 1;$$
$$\texttt{while } \neg(x = 1) \texttt{ do } (y := y * x; x := x - 1)$$

Assume user input $s\ x = \mathbf{3}$.

Categorical representation of program



States during program execution

s_0		
x	1	$\perp$
y	1	$\perp$

$a)$

s_1		
x	1	**3**
y	1	$\perp$

$b)$

s_2		
x	1	**3**
y	1	**1**

$c)$

s_3		
x	1	**3**
y	1	**3**

$d)$

s_4		
x	1	**2**
y	1	**3**

$e)$

s_5		
x	1	**2**
y	1	**6**

$f)$

s_6		
x	1	**1**
y	1	**6**

$g)$