Chapter 5

Semantics

In this chapter we start studying the semantics of programming languages. We saw how the syntax of a programming language defines which are its well-formed programs independently of the functions that they compute. The semantics gives a meaning to well-formed programs by interpreting their syntax symbols. Semantics provides us with insights on the dynamic evolution of programs as well as on static properties that hold independently of execution aspects.
We first motivate the need of a formal definition of semantics and we briefly survey the main features of operational, denotational and axiomatic semantics. We show that the syntactic apparatus of a language can be considerably simplified to deal with semantics, thus introducing abstract syntax. We then start studying some desirable properties of semantics. The first is compositionality that can be implemented via inductive definitions on the structure of the abstract syntax of languages. We then study substitutivity and full abstraction and we relate them together. Modularity of definitions is discussed as well. Finally, we briefly introduce the static properties described by semantic models.

5.1 Why semantics

Historically, the semantics of programming languages has been given in natural language simply by structuring the presentation for instance as in

the user manuals. Hence, the meaning of constructs is described ambiguosly. The only exact definitions are the ones provided by the implementations of the intermediate machines of the languages. As a consequence, different implementations define different programming languages.

A formal definition of the semantics of programming languages provides constructs with an exact and unambiguous meaning. Therefore, the implementor of the language has a precise specification and dialects of languages should arise no more. Furthermore, serious programmers can understand what their programs do exactly without executing them. Programmers can also benefit from formal descriptions such as set of axioms to verify and transform their programs. Finally, formal theories allow us to prove essential properties of programs like for instance security constraints.

Summing up a formal semantics is of help both for designing, implementing and using a programming language. In fact, the semantics should be a trade off between the language and the implementation of its intermediate machine. The foundamental aspects concerning the meaning of languages are described in this book at two levels: the formal semantics and the implementation of the intermediate machine. The semantics is independent of the particular implementation of a language, although it shuold highlight the potential of troubles and suggest some design alternatives. The implementation of the intermediate machine fixes some architectural aspects and must agree with the formal semantics. In other words, we can interpret the semantics as an abstract specification of the intermediate machine of the language. The programmer is interested in the structure of the intermediate machine as well, because it influences the efficiency of the programs. In particular, a programmer is intereted in knowing which part of the machine are hardware, firmware or software and in how high level constructs are translated into sequences of instructions of the host machine to have a rough estimation of the performance of programs.

5.2 Which semantics

There exist some techniques to provide programs with meaning. For instance, besides the operational semantics that we already mentioned there

are denotational and axiomatic semantics as well. We briefly sketch the peculiarities of these approaches.

- *Operational semantics*. The meaning of a construct of a language is given by the operations that it induces on the (abstract) machine. Therefore the operational semantics describes *how* the effect of the execution of a construct is obtained.

- *Denotational semantics*. The meaning of a construct of a language is given by a mathematical object. Therefore the denotational semantics only describes the *effect* of the execution of a construct.

- *Axiomatic semantics*. Peculiar properties of the execution of a construct are expressed via assertions. As a consequence, *some aspects* of the execution and of its effect can be ignored by axiomatic semantics. This semantics is thightly connected with mathematical logics.

We want to stress here that the different approaches at the formal definition of semantics are not mutual exclusive and they are not in competition. For instance the operational semantics is better suited than the other approaches for giving guidelines to implementors and to deal with concurrency and distributed issues of languages. The denotational semantics is instead more suitable to reason about programs and to study properties like detecting whether all variables have been initialised before their usage or whether all parts of programs are reachable. Action semantics is more concerned with readability still maintaining the formal rigour and peculiarities of denotational semantics. The axiomatic semantics provides us with a logical system that can be used to prove properties in a semi-automatic way. We shall see in the next sections that the operational semantics can be given in a logical style as well, thus recovering the peculiarities of the axiomatic approach.

We exemplify in the next subsections the application of the kinds of semantics above to the simple program

```
z:=2; y:=z; y:=y+1; z:=y;
```

made up of four assignment statements.

5.2.1 Operational semantics

Since the main characteristic of the operational semantics concerns the description of *how* programs are executed, we can state informally that our sample program is executed by

- performing the statements separated by semicolons sequentially and in the order in which they are listed from left to right; and by
- determining for any statement the value of its right hand side (the one on the right of the symbol `:=`) that is assigned to the left hand side.

Let the state of the abstract machine that executes our sample program be a pair whose first component is the program to be run and the second component is a function that associates any variable in the program with its current value. Assume that initially any variable holds the value 0. The execution of the program can then be represented by the following sequence of transitions between states

$$\begin{array}{l}
\langle \texttt{z:=2; y:=z; y:=y+1; z:=y;}, [z = 0, y = 0]\rangle \rightarrow \\
\langle \texttt{y:=z; y:=y+1; z:=y;}, [z = 2, y = 0]\rangle \rightarrow \\
\langle \texttt{y:=y+1; z:=y;}, [z = 2, y = 2]\rangle \rightarrow \\
\langle \texttt{z:=y;}, [z = 2, y = 3]\rangle \rightarrow [z = 3, y = 3]
\end{array}$$

where we represent the function component of the states as a sequence of elements $x = v$ to mean that variable x has value v. In the first step we execute the leftmost assignment that changes the value of z from 0 to 2 (see the function component of the resulting state). The program which is left to execute is `y:=z; y:=y+1; z:=y;`. The second step assign to y the value of z, then we increment y. The last step assign to z the new value of y. In the final state we simply report the function part of the state because no other statement has to be executed.

The operational description of the execution of our sample program is indeed an *abstraction* of how the program is executed. No architecture detail like usage of registers or addressing techniques is explicitly given. Therefore operational semantics is architecture independent although dealing with execution of programs.

The kind of operational semantics that we used in the example above is sometimes called *small step semantics* because each step of a computation that produces partial changes on the abstract machine is visible. Another approach to define operational semantics is called *natural* (or *big step*) *semantics*. This kind of semantics describes in a single (big) step the overall computation. In our example we have a single big transition

$$\langle \texttt{z:=2; y:=z; y:=y+1; z:=y;}, [z = 0, y = 0]\rangle \Rightarrow [z = 3, y = 3]$$

Although big step semantics is still defined in terms of transitions between configurations is quite close to denotational semantics. In fact, the effect of the execution of a program on the final configuration is available. The intermediate steps are hidden in the big transition. A drawback of big step semantics is that it applies only to terminating programs. We will discuss small and big step semantics in a later chapter.

5.2.2 Denotational semantics

The main characteristics of the denotational semantics is the description of the *effect* of the execution of a program. We state below informally the effect of the execution of our sample program.

- The effect of the execution of a sequence of statements separated by semicolons is the function composition $\circ$ of the effects of the individual statements listed from left to right; and
- the effect of the execution of any statement is a function that given a state produces a new state in which the value of the right hand side of the assignement is equal to the one of the left hand side.

We define an *interpretation function* for any construct to associate with it a *denotation* (the mathematical object describing the effect of the execution of the construct). For our running example we need an interpretation function $\mathcal{E}$ for the assignment. It takes as arguments the assignment to be executed and the current values of variables and returns the new value of the variable modified by the assignment. For instance,

$$\mathcal{E}[\![\texttt{z:=2}]\!]([z = 0]) = [z = 2].$$

The meaning of the program is obtained by composing the functions $\mathcal{E}$ applied to all the statements.

$$\mathcal{E}[\![\texttt{z:=2; y:=z; y:=y+1; z:=y;}]\!]([z=0,y=0]) =$$
$$(\mathcal{E}[\![\texttt{z:=y}]\!] \circ \mathcal{E}[\![\texttt{y:=y+1}]\!] \circ \mathcal{E}[\![\texttt{y:=z}]\!] \circ \mathcal{E}[\![\texttt{z:=2}]\!])([z=0,y=0]) =$$
$$(\mathcal{E}[\![\texttt{z:=y}]\!](\mathcal{E}[\![\texttt{y:=y+1}]\!](\mathcal{E}[\![\texttt{y:=z}]\!](\mathcal{E}[\![\texttt{z:=2}]\!])([z=0,y=0])))) =$$
$$(\mathcal{E}[\![\texttt{z:=y}]\!](\mathcal{E}[\![\texttt{y:=y+1}]\!](\mathcal{E}[\![\texttt{y:=z}]\!])([z=2,y=0]))) =$$
$$(\mathcal{E}[\![\texttt{z:=y}]\!](\mathcal{E}[\![\texttt{y:=y+1}]\!])([z=2,y=2])) =$$
$$\mathcal{E}[\![\texttt{z:=y}]\!]([z=2,y=3]) = [z=3,y=3]$$

Note that we describe the effect of our program simply by manipulating mathematical objects. Indeed the peculiarity of the denotational semantics is to abstract away from any execution concept. The difference with operational semantics would be more evident if we chose a program with more sophisticated constructs.

A disadvantage of denotational semantics is its readability. In fact, any definition is a composition of functions that tends to become long-wired and error-prone. A variant of denotational semantics is *action semantics* which tries to translate denotational formulae in english-based readable sentences.

Action semantics uses a standard family of operators to describe the standard features of programming languages. Standard structures called *facets* with operators to manipulate their elements are pre-defined for expressions, declarations and commands. Combinators to compose the operators of the facets are given as well.

The action semantics has two levels of abstraction. The upper level specifies a language in terms of *actions*: entities that can be performed on some input data and producing outcomes for other actions. This is the aspect more directly connected to abstract machines. The lower level specifies the actions.

For instance the meaning of the assignment

$$z \texttt{ := } 2$$

looks like

$$\begin{array}{l}\textbf{execute}[\![z := 2]\!] = \\ \quad (\textbf{find}\, z \,\texttt{and}\, \textbf{evaluate}[\![2]\!])\,\texttt{then}\,\textbf{update}\end{array}$$

where the words in boldface are operators and the ones in typewritten are combinators. The operators together with their arguments form the actions. Thus, **find** z is an action and **evaluate**$[\![2]\!]$ is another one. The two actions are combined by the combinator `and`. The operator **find** extracts the location of z and **evaluate** returns the value associated with the literal 2. The **update** operator generates the association of z with the new value.

The action semantics of our running example is

$$\begin{array}{l}\textbf{execute}\,[\![z := 2; y := z; y := y + 1; z := y]\!] = \\ \quad \textbf{execute}\,[\![z := 2]\!]\ \texttt{andthen} \\ \quad \textbf{execute}\,[\![y := z]\!]\ \texttt{andthen} \\ \quad \textbf{execute}\,[\![y := y + 1]\!]\ \texttt{andthen} \\ \quad \textbf{execute}\,[\![z := y]\!]\end{array}$$

where each **execute** is expanded as in the example above.

A distinguishing feature of action semantics is that the structure to hold values of identifiers such as stores are dealt with implicitly.

5.2.3 Axiomatic semantics

Axiomatic semantics is deefined by giving a set of *proof rules* defined according to the syntax of the language. This semantics asserts properties of programs rather than defining their meanings. The literature often report the sentence *partial correctness properties* for the properties studied by axiomatic semantics. A program is partially correct with respect to a precondition and a postcondition if whenever the initial state fulfill the precondition and the program terminates, then the final state is guaranteed to fulfill the postcondition. The word partial refers to the fact that the program may not terminate. In fact, the termination of the program under investigation must be proved separately. If we prove partial correctness and termination we have *total correctness*.

Hereafter we let P, Q, R range over preconditions and postconditions, i.e. assertions of a first order language with equality. We write $\{P\}c\{Q\}$ to mean that the precondition P holds, then the command c is executed and the postcondition Q holds in the state reached.

The rule

$$\frac{\{P\}c_0\{Q\},\ \{Q\}c_1\{R\}}{\{P\}c_0; c_1\{R\}}$$

states that the sequential composition of two commands is possible if the postcondition of the first command coincides with the precondition of the second one. Finally, an important rule is the one of *consequence*

$$\frac{P' \Rightarrow P, \{P\}c\{Q\}, Q \Rightarrow Q'}{\{P'\}c\{Q'\}}$$

It states that less informative formulae can be inferred from more informative ones.

In our running example we can take as precondition of the sequence of assignments $\{tt\}$ and as postcondition $\{z = y\}$. The proof of partial correctness must establish that if the program terminates, it ends up in a state where $z = y$. We can view the proof as a derivation tree in a logical system of axioms and inference rules.

$$\frac{\dfrac{\{z = 0\} \Rightarrow \{tt\}, \{tt\}\texttt{z:=2}\{z = 2\}, \{z = 2\} \Rightarrow \{z = 2\}}{\{z = 0\}\texttt{z:=2}\{z = 2\}}, \{z = 2\}\texttt{y:=z}\{z = 2, y = 2\}}{\{z = 0\}\texttt{z:=2;y:=z}\{z = 2, y = 2\}}$$

The postcondition in the conclusion can be written $\{z = 2, y+1 = 3\}$ that can be seen as the precondition to apply the assignment `y:=y+1` and to produce the postcondition $\{z = 2, y = 3\}$. By applying the consequence rule we have

$$\frac{\{y = 3, z = 2\} \Rightarrow \{y = 3\}, \{y = 3\}\texttt{z:=y}\{z = 3, y = 3\}, \{z = 3, y = 3\} \Rightarrow \{z = y\}}{\{y = 3, z = 2\}\texttt{z:=y}\{z = y\}}$$

For an evidence that the axiomatic semantics consider only some aspects of the execution of programs, we may list many programs that satisfy the

pre and postconditions above, but that beahave differently. An example is `y:= -1000;z:=y` or simply `z:=y`. Furthermore, a program can satisfy many different pre- and post-conditions. For instance our sample program satisfies the precondition $\{z = n\}$ and the postcondition $\{z = n + 3\}$ as well.

It is important for the formal system of axioms being *sound*, i.e. every formula derived in the formal system is true in the semantic interpretation considered. Another important feature is *completeness*. It says that every true formula in the interpretation considered is derivable in the formal system.

5.3 Abstract syntax

The *concrete syntax* of a language is a set of strings over an alphabet. It is usually specified through a *context free grammar* G in BNF notation according to theory developed in Chapters 2 and 4. The definitional mechanisms are *syntactic* in nature because they describe how programs are assembled from their parts. In fact, we start from basic components (the tokens or terminal symbols) and we follow some rules (productions or BNF definitions) to compose them.

Sometimes context free grammars contains nonterminals and productions which are not needed to derive the strings of a language. They are introduced to rule out ambiguities from grammars like T and F in the Example 2.59. This allows us to define deterministic algorithms (parsers) to decide whether a string belongs or not to a language. Furtheromre productions and BNF definitions take care of the morphological aspects of program constituents. For instance, grammars differ if we represent assignment as x `:=` e or $ASS(x, e)$ or $e \mapsto x$ or even $x = e$ as in C. However, to study the meaning of an assignment we are not concerned at all with its morphological aspect. We only need to recognize the occurrence of an assignment within a program and to isolate its main arguments (in the example above x and e).

The *abstract syntax* of a language was originally introduced by Mc Carthy as an interface between the concrete syntax and the semantic interpretation of the language. Abstract syntax is *analytic* rather than synthatic

because describes how to decompose a program into basic bricks instead of showing how to compose them. Furthermore, it abstracts from the peculiar notation used to represent program components. We pick one notation that is well-suited for semantic definitions and we define it as simply as possible.

Abstract syntax is obtained by erasing from the concrete one irrelevant information such as precedence among operators, hierarchies of derivations and so on. In fact, when dealing with semantic issues, we are not interested in the way in which strings are derived, but only in their meaning. Derivation trees (also called *concrete syntax trees*) can be thus simplified by collapsing sequences of nonterminals along a path. The set of trees obtained in this way constitute the *abstract syntax trees* and the grammar (possibly ambiguous) that originates them defines the *abstract syntax*.

Definition 5.1 (abstract syntax tree)**.** *An* abstract syntax tree *is a tree in which each node represents an operator and the children of the node represent the operands.*

EXAMPLE 5.2 The concrete syntax tree of the expression $id + id * id$ is depicted in Fig. 5.1(a). The nodes labelled by nonterminals provide us with no insight on the meaning of the operation.
The abstract syntax tree (Fig. 5.1(b)) rules out superflous details to collect the essential information involved in the description of operations: the operators and their arguments. (The abstract syntax of the arithmetic expressions is defined by the grammar in the Example 2.18.) $\Diamond$

What we do with abstract syntax is to consider the deep structure of program components. As a consequence, we associate syntactic categories with sets of strings with *independent* semantic meaning. From an applicative point of view, we may assume that the abstract syntax of a language is the outcome of a parser.

5.4 The scenario

In the previous section we reported some examples of semantic definitions, and we introduced the notion of abstract syntax as the starting point for the

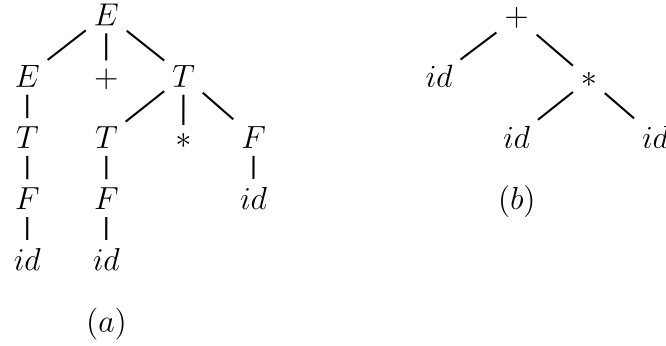

Figure 5.1: Concrete (a) and abstract syntax tree (b) of $id + id * id$.

study of semantics. We here formalize the notion of semantics and then we show how the examples above fit our framework.

The idea of a semantics has several components. There is an (*object-*) *language* $\mathcal{L}$ to the terms of which we have to give meaning. We assume hereafter that $\mathcal{L}$ is given through an abstract syntax. The meaning of a term of $\mathcal{L}$ is a term of a *meta-language* $\mathcal{M}$. The meta-language is usually more abstract than the object-language and it highlights the aspects of interest like the way in which a term is executed or simply the effect of its execution. Note that the meta-language can be flowcharts, lattices or other mathematical structures, and not necessarily a set of strings. A *semantic function* $\mathcal{E}$ maps terms of $\mathcal{L}$ to their meaning in $\mathcal{M}$. Finally, there is an *equivalence relation* $\equiv$ on meta-language terms establishing when they are equal. We would like to have a semantic function $\mathcal{E}$ such that

$$\forall l, l' \in \mathcal{L} \,.\, \mathcal{E}[\![l]\!] = \mathcal{E}[\![l']\!] \;\Rightarrow\; \mathcal{E}[\![l]\!] \equiv \mathcal{E}[\![l']\!].$$

The intuition is that we can exchange (sub-) programs that have the same meaning without changing the behaviour of a system.

Definition 5.3 (semantics)**.** *A* semantics *is a quadruple*

$$\langle \mathcal{L}, \mathcal{M}, \mathcal{E}_{i \in I}, \equiv \rangle,$$

where

- *the* object-language $\mathcal{L}$ *is a free algebra possibly many-sorted (see App. A.5);* [1]

- *the* meta-language $\mathcal{M}$ *is an algebra;*

- $\mathcal{E}_{i\in I} : \mathcal{L} \to \mathcal{M}$ *is an indexed family of* semantic functions*; and*

- $\equiv\, \subseteq \mathcal{M}\times\mathcal{M}$ *is an* equivalence relation *on which we require* $\forall\, m, m' \in \mathcal{M}\,.\, m = m' \;\Rightarrow\; m \equiv m'$.

The following examples shows how the three kinds of semantics introduced in the previous section can be casted into our definition.

EXAMPLE 5.4 (operational semantics) Consider the operational semantics of the sample program in Subsect. 5.2.1. The object-language $\mathcal{L}$ is the language whose meaning we are defining. The set of the states that a program can pass through is $S = \{\langle l, \rho : Var(l) \to Val\rangle \mid l \in \mathcal{L}\}$ where l is the program to be executed and the function ρ assigns values (from the set Val) to the variables in l (denoted by $Var(l)$). Then, the meta-language is defined as $\mathcal{M} = \langle S, \to\rangle$ where $\to\, \subseteq S \times S$ is the transition relation of Subsect. 5.2.1. Note that $\mathcal{M}$ is made up of graphs representing the dynamic evolution of programs sometimes called *transition graphs*. We have a single semantic function $\mathcal{E} : \mathcal{L} \to \mathcal{M}$. As far as big step semantics is concerned, we only need to change the definition of $\to$ into $\Rightarrow$.

Since we defined no equivalence relation on the terms of $\mathcal{M}$, we can let $\equiv$ be for instance the syntactical identity or an isomorphism of graphs preserving the ρ component of nodes. $\Diamond$

EXAMPLE 5.5 (denotational semantics) Consider the denotational semantics of the sample program in Subsect. 5.2.2. The object-language $\mathcal{L}$ is the language whose meaning we are defining. The meta-language is a function $\rho : Var(l) \to Val$ that assigns values to variables. The semantic function is the $\mathcal{E}$ defined in Subsect. 5.2.2 and the equivalence can be taken again to be syntactic equality.

Note that we here only consider the assignments of values to variables because we are only interested in the effect of the execution of the program. In the

[1] Note that the structure of an abstract grammar originates a signature. Therefore, we can say that the abstract syntax of a language is a term algebra. Since a term algebra is completely defined by its signature, we can define the semantics of the language by interpreting the symbols of the signature into a Σ-algebra.

Example 5.4 we define instead the sequence of transitions $\rightarrow$ that leads to the final assignment of values to variables because we were interested in how terms are executed. As a consequence, we can simplify the structure of $\mathcal{M}$ by only taking the second component of the meta-language used for the operational semantics. This is the effect of denotational semantics being more abstract than operational definitions. $\Diamond$

EXAMPLE 5.6 (action semantics) Since action semantics is a different formalism to express denotational formulae, we have the same object-language $\mathcal{L}$, meta-language $\mathcal{M}$ and equivalence relation of Example 5.5. The definition of the family of semantic functions is given in terms of actions and their combinators. $\Diamond$

EXAMPLE 5.7 (axiomatic semantics) Consider the axiomatic semantics of the sample program in Subsect. 5.2.3. The object-language $\mathcal{L}$ is still the language whose meaning we are defining. The meta-language $\mathcal{M}$ is a set of formulae made up of programs of $\mathcal{L}$ enriched by pre- and post-conditions. The semantic function $\mathcal{E}$ maps programs into proofs in the formal system of axioms. Finally, $\equiv$ groups together the proofs that satisfy the same pre- and postcondition (see Subsect. 5.2.3). $\Diamond$

We end this section with a remark. Note that the equivalence relation $\equiv\subseteq \mathcal{M} \times \mathcal{M}$ induces an equivalence relation on the strings of $\mathcal{L}$ defined by $l \equiv l' \;\Leftrightarrow\; [\![l]\!] \equiv [\![l']\!]$.

5.5 Compositionality

Any non trivial language allows one to write infinite programs (see Chapter 2). An immediate consequence is that the definition of a semantic function (independently of the kind of semantics chosen) cannot rely on particular programs. Semantic definitions only must consider the elementary and basic components of programs. Then, a mechanism to assemble together the meanings of these components to get the meaning of the whole program must be supplied. We express this idea by the following principle.

Principle 5.8 (compositionality)**.** *The meaning of any sentence is function of the meaning of its immediate constituents.*

The principle of compositionality is essential to define the behaviour or the meaning of a system that has potential infinite elements. It is not only typical of semantic definitions, but of many fields of computer science, mathematics and logics. Indeed this principle dates back to the work of Frege in the last decade of 1800. As a remark, some authors refer to compositional definitions as *syntax-directed* definitions.

The problem of finitely describing infinite objects was already faced in Chapter 2 where we used grammars as a generative description of languages. We can now rely on grammars to identify the basic constructs of a language and on the grammar productions to devise composition of elementary meanings.

EXAMPLE 5.9 (binary numerals) A binary numeral is described by the following BNF-like grammar

$$B ::= B0 \,|\, B1 \,|\, 0 \,|\, 1$$

that coincides with the object-language $\mathcal{L}$. The meaning of a binary numeral is its corresponding number in decimal notation. Hence the meta-language for this example is the set of natural numbers $I\!N$. We apply a simple algorithms that works independently of the binary chosen, and we codify it in the definition of the semantic function $\mathcal{E} : B \rightarrow I\!N$. In particular, we have

$$\mathcal{E}[\![0]\!] = 0,\;\; \mathcal{E}[\![1]\!] = 1,\;\; \mathcal{E}[\![B0]\!] = 2 \times \mathcal{E}[\![B]\!],\;\; \mathcal{E}[\![B1]\!] = 2 \times \mathcal{E}[\![B]\!] + 1.$$

To satisfy compositionality, we define the meaning of a composite binary numeral like $B0$ or $B1$ in terms of the meaning of the numerals B. Note that the operation of concatenation of literals within brackets $[\![\;]\!]$ (e.g., $B0$) is defined on $\mathcal{L}$, while $\times$ and $+$ are the usual multiplication and summation on $I\!N$ and therefore are operations of our meta-language.

For the sake of completeness, to characterize completely the semantics of binary nuemerals we must define an equivalence relation on meta-language terms. We let here $\equiv$ be the equality $=$ on $I\!N$. $\diamondsuit$

The denotational semantics as introduced in Subsect. 5.2.2 immediately satisfies the principle of compositionality because the meaning of programs is obtained by function composition. The axiomatic approach satisfies the principle above as well. In fact, the assertions on a program

are derived by the ones on its components (the premises of the inference rules).

The operational semantics has been considered for a long time to lack compositionality. Plotkin in the eighties solved the problem by defining the operational semantics in logical style by following the definition of the syntax of the language (see Subsect. **??**) and relying on the syntactic decomposition of complex sentences into simpler ones. The pattern of the decomposition being suggested by the grammar that defines the language.

We end this subsection with a remark. Note that there are constructs of programming languages to which it is not easy to give meaning compositionally.

EXAMPLE 5.10 Consider for instance a function call. Its evaluation depends on the body in the definition (declaration) of the function, but such a definition does not occur in the statement of the call.

Another example is given by the **while** loop. Its meaning can be defined relying on an **if** statement by rewriting

while `cond` **do** `stmt`

as

if `cond` **then** `stmt;` **while** `cond` **do** `stmt`

Here compositionality is violated because the **if** statement does not occur in the **while** definition. ♢

5.6 Substitutivity and full abstraction

A suitable definition of the equivalence relation within a semantics induces an equivalence on $\mathcal{L}$ such that two sentences are equivalent $l \equiv l'$ if they can be exchanged one another within any program without affecting its meaning. Intuitively, this means that there is no way to distinguish the behaviour of the equivalent sentences. This property is usually known as substitutivity principle. Consider again Example 5.9. It holds the equivalence $0B \equiv B$ because $[\![B0]\!] \equiv [\![B]\!]$. To formalize this principle we need the notion of program context.

Definition 5.11 (context). *A program* context $\mathcal{C}[\bullet]$ *is a program with a hole. The definition easily generalizes to many holes contexts* $\mathcal{C}[\bullet, \ldots, \bullet]$*. Hereafter we avoid the word program when unambiguos.*

We now report some examples of contexts.

EXAMPLE 5.12 (contexts) Consider the program fragment

$\texttt{if}\ cond\ \texttt{then}$
$\quad \texttt{while}\ cond1\ \texttt{do}\ stmt$

The context of the first condition is

$\mathcal{C}_{cond}[\bullet] = \texttt{if} \bullet \texttt{then}$
$\quad \texttt{while}\ cond1\ \texttt{do}\ stmt$

The two holes context of the first condition and the statement is

$\mathcal{C}_{cond,stmt}[\bullet, \bullet] = \texttt{if} \bullet \texttt{then}$
$\quad \texttt{while}\ cond1\ \texttt{do} \bullet$

$\Diamond$

Eventually we can state the principle of substitutivity.

Principle 5.13 (substitutivity). *Let* l *and* l' *be two sentences that can occur within program contexts* $\mathcal{C}[\bullet]$*. Then,* $l \equiv l' \Rightarrow \forall \mathcal{C}[\bullet] \,.\, \mathcal{C}[l] \equiv \mathcal{C}[l']$.

Substitutivity is the theoretical basis for optimizing code with respect to performance, reliability, readability, security and so on. For instance optimizing compilers that drop from the object code variables which are never used or that transform loops that are entered only once into sequence of statements, apply a semantic equivalence of sentences of the language.

A good semantic definition should assign the same meaning to all the sentences that are equivalent in the sense above. This property is called full abstraction and it is formalized below.

Principle 5.14 (full abstraction). *Let* l *and* l' *be two sentences that can occur within program contexts* $\mathcal{C}[\bullet]$*. Then,* $\forall \mathcal{C}[\bullet] \,.\, \mathcal{C}[l] \equiv \mathcal{C}[l'] \Rightarrow l \equiv l'$.

Full abstraction is the converse of substitutivity. It allows one to state that any optimization that can be performed is considered by the equivalence relation $\equiv$ on sentences. We remark that full abstraction is extremely difficult to implement for many languages.

5.7 Modularity

The definition of the semantics of complex languages and their design is usually an incremental task. One starts considering the basic constructs and then enlarge the set of operators adding new definitions. To make this process easy it must not be necessary to modify the definitions of the already considered constructs when adding a new one. Thus, we state the following principle.

Principle 5.15 (modularity)**.** *The semantic definitions of a language must not be changed when adding new constructs to the language.*

Although modularity may seem a notational matter, it has a profound practical application for the separete compilation of the modules of programs.

5.8 Static semantics

In the previous section we exemplified the application of semantic definitions to describe the execution of programs. The properties involving the evolution of programs are called dynamic and the semantic techniques used to study them are classified as *dynamic semantics*.

There are properties of programs that can be verified without running (intended also as semantic simulation of execution) them. These properties are static, the semantic techniques used to verify them are known as *static semantics* and the verification is usually called *static analysis*. The main example of static analysis is maybe *type theory*.

5.8.1 Type theory

The type theory is a formal language to organize the values manipulated by programs. The central notion is the one of type. The main role of type theory is to rule out from the set of programs syntactically correct the ones that will surely generate an execution error when running. Note that this does not mean that well-typed programs cannot occur in execution errors.