Chapter 1

Programming Languages

Starting a course on programming languages and compilers it makes sense first to stipulate what programming languages are. In a nutshell, programming languages are languages for writing computer programs. While looking vacuous, this definition nevertheless discovers an important observation: since programming languages are *languages*, i.e. *sign systems*, to reason about programming languages we can apply the notions and terminology of *semiotics*, a branch of science dealing with sign systems.

Figure 1.1: Charles William Morris (1901-1979)

One of the founders of semiotics, Charles William Morris, has identified the following important notions:

- *Syntax* — relations between the signs of sign system themselves.
- *Semantics* — relations between a sign system and objects.
- *Pragmatics* — relations between a sign system and a person.

In a narrower context of programming languages syntax denotes the form of program representation, semantics — the "meaning" of programs, and pragmatics — the relation between programming language and developer. In our course we focus mainly on syntax and semantics, putting all pragmatics questions aside.

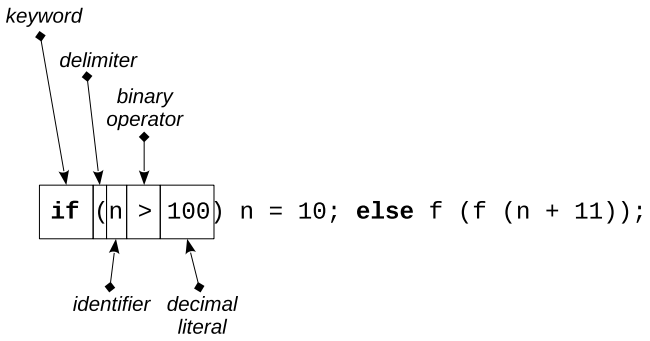
1.1 Syntax

Similarly to natural languages, the syntax of a programming language can be decomposed into a few levels (lexical structure, grammar, etc.) However, unlike natural languages, which have been evolving more or less spontaneously, the syntax for a programming language is intelligently designed taking into account a number of specific requirements; in particular, it is (as a rule) *unambiguous* and allows for efficient analysis.

To illustrate the concept of programming language syntax consider the following simple snippet in C:

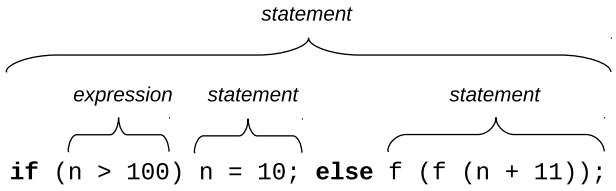
```
if (n > 100) n = 10; else f (f (n + 11));
```

From the *lexical* standpoint this fragment can be seen as a sequence of *tokens* (a keyword, a delimiter, an identifier, a binary operator, a decimal constant, etc.):



This sequence of tokens, in turn, is grouped into an hierarchy of syntactic constructs (in this case, expressions and operators):

Natural languages, as a rule, are *ambiguous*. Consider the following phrase: "Entering the doors with pets hold them." Hold what? The doors or the pets? In this case we encounter a *global ambiguity* which cannot be resolved even



taking into account the context. The only way to resolve it is to rephrase ("Entering the doors hold your pets" or "Hold your pets while entering the doors"). The presence of such unresolvable ambiguities in a programming language is unacceptable.

1.2 Semantics

The drastic difference between natural and programming languages manifests itself in a full bloom on the semantic level. Unlike natural languages, for programming languages there are ways to formally specify their semantics and acquire a *mathematically proven* results.

Why formal semantics matters? While in a common practice of using programming languages for application-level software we can rely on our vague, fuzzy understanding of the meanings of their constructs, when we develop system-level programming *tools*, in particular, compilers, this understanding turns out to be insufficient. Imagine, for example, that we know that in some programming language expressions consist of variables, constant and four arithmetic operators. Is this knowledge complete?

Let us have the following expressions:

$0 * (\mathtt{x} / 0)$
$1 + \mathtt{x} - \mathtt{x}$

What should be the results of their evaluation?

In the first sample, on one hand, the multiplication to zero always gives zero; on the other hand, the division by zero is undefined. So, would the result of the expression be zero or undefined?

In the second sample, we add a value of `x` to 1 and immediately subtract it. On one hand this would give us 1; on the other hand, the value of `x` can be undefined, or adding `x` to 1 might lead to an overflow. So it remains unclear if we can replace $1 + \mathtt{x} - \mathtt{x}$ with 1 while preserving the behaviour of the program in all cases.

Even if we cannot come up with definite oral answers to these questions we still can write programs using this language; after all, we always can see what happens in each case if we have a compiler. But what should we do if our task is to implement the *first* compiler for this language? What happens if we ask a dozen of people to implement a dozen of compilers independently? Would all

these compilers agree in their semantics, or, perhaps, we eventually will have a dozen of *different* compilers since their authors have *different* intuition?

One may argue that all these samples come from a very rare and unrealistic use cases. Indeed, how often a new compiler is created, let alone a dozen of those? And those snippets with "murky" semantics look like antipatterns: why on earth a sane developer would write "$0*(x/0)$" instead of "0" or "$1+x-x$" instead of "1"?

The answer, somewhat unexpected, is that actually all these cases are rather *general* then exceptional. Of course, implementing a completely new compiler from scratch is not an everyday task; however, at the same time programming languages and compilers are rarely developed "once and for all". As a rule, both languages and their compilers evolve through time: new constructs are being added to languages, and new features are being implemented in compilers. Carrying out all of these routine tasks require a strong semantic foundations. Besides this, compilers by no means are the only semantic-sensitive development instruments: there is an abundance of other important tools like IDEs, model checkers, static analyzers, debuggers, profilers, etc., all of which have to interpret the semantics of programs in a coherent way.

Then, not all programs are written directly by human beings. Actually, a fair share of them are generated by other tools, in particular, *preprocessors* or other *metaprogramming* tools. The results of metaprogramming as a rule look exactly like the samples discussed above: they contain a lot of vacuous use of constructs and programming language features, and it is *expected* from the underlying compiler to clean up this mess in a semantics-preserving manner.

Finally, for a compiler there is nothing "weird" in those code samples. The reason is simple: compilers do not have intuition. They just routinely convert program texts into executables no matter how weird they would look from a human being point of view. To illustrate this, consider the following short program in C:

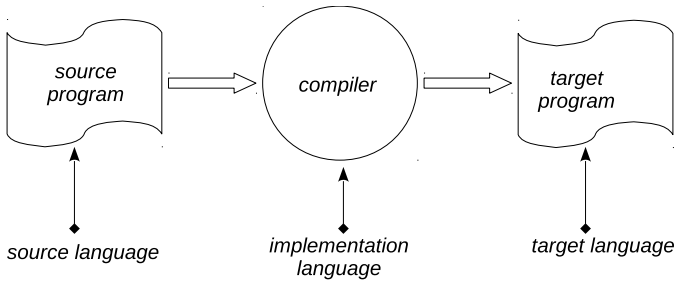
```
m(f,a,s)char*s;
{char c;return f&1?a!=*s++?m(f,a,s):s[11]:f&2?a!=*s++?
1+m(f,a,s):1:f&4?a--?putchar(*s),m(f,a,s):a:f&8?*s?
m(8,32,(c=m(1,*s++,"Arjan Kenter. \no$../.\""),
m(4,m(2,*s++,"POCnWAUvBVxRsoqatKJurgXYyDQbzhLwkNjdMT"
"GeIScHFmpliZEf"),&c),s)):65:(m(8,34,"rgeQjPruaOnDaP"
"eWrAaPnPrCnOrPaPnPjPrCaPrPnPrPaOrvaPndeOrAnOrPnOrPn"
"OaPnPjPaOrPnPrPnPrPtPnPrAaPnBrnnsrnnBaPeOrCnPrOnCaP"
"nOaPnPjPtPnAaPnPrPnPrCaPnBrAnxrAnVePrCnBjPrOnvrCnxr"
"AnxrAnsrOnvjPrOnUrOnornnsrnnorOtCnCjPrCtPnCrnnirWtP"
"nCjPrCaPnOtPrCnErAnOjPrOnvtPnnrCnNrnnRePjPrPtnrUnnr"
"ntPnbtPrAaPnCrnnOrPjPrRtPnCaPrWtCnKtPnOtPrBnCjPronC"
"aPrVtPnOtOnAtnrxaPnCjPrqnnaPrtaOrsaPnCtPjPratPnnaPr"
"AaPnAaPtPnnaPrvaPnnjPrKtPnWaOrWtOnnaPnWaPrCaPnntOjP"
"rrtOnWanrOtPnCaPnBtCjPrYtOnUaOrPnVjPrwtnnxjPrMnBjPr"
"TnUjP"),0);}
main(){return m(0,75,"mIWltouQJGsBniKYvTxODAfbUcFzSp"
"MwNCHEgrdLaPkyVRjXeqZh");}
```

If you have any doubts if this is a valid C program, compile and run it. C is never considered as particularly hard to understand or syntactically challenging (both arguable), yet the program presented is totally incomprehensible. Actually, it was specifically designed to be such: there is a whole competition in writing *obfuscated* code[1]. For us, however, the important observation is that this program no more obfuscated for a compiler then any other one. Our job as compiler implementors is to make them behave in such a way.

1.3 The Essence of Translation

Translation is a syntactic transformation of programs in one language into (equivalent) programs into another. The feasibility of fully automatic translation constitutes a drastic difference between programming languages and natural ones. Although natural language translation techniques and tools made a tremendous progress in a recent years the adequacy of the results remains the matter of discussion; and, anyway, this adequacy can not be mathematically proven. For compilers this problem is essentially solved.

A compiler itself is not a magical creature — it is a *program*, written, in turn, in some programming language. Thus, speaking about compilation, we actually deal with *three* languages:



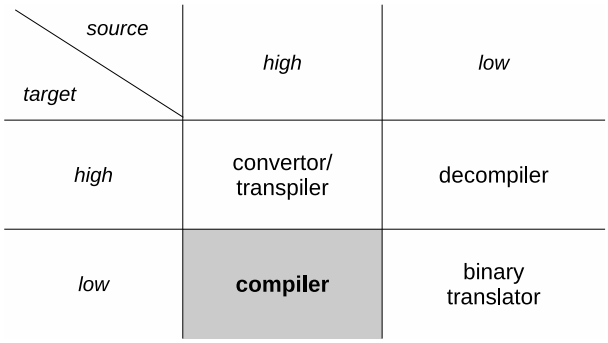
- The language in which the programs being compiled are written (*source language*).
- The language in which the programs being compiled are compiled (*target language*).
- The language in which the compiler from source to target language is written (*compiler implementation language*).

1.4 Translation Subspecies

As you probably know, there is a subdivision of languages into high- and low-level. This subdivision is not absolute: for example, HASKELL is consid-

[1] https://www.cise.ufl.edu/~manuel/obfuscate/obfuscate.html

ered by many as more high-level language than C, which, in turn, is of higher level then FORTRAN, etc. As a rule, high-level languages are equipped with type systems, while low-level are not (but SCHEME is untyped while there exist typed assemblers). This coarse-grained classification of languages leads to a corresponding coarse-grained classification of translators:



There is a somewhat common knowledge that among all these subspecies compilers are the most challenging to implement. While we generally agree, it's worth mentioning that the other kinds of translators are not pieces of cakes at all: for example, it is expected from a decompiler or convertor to properly utilize the abstractions of the target language, to produce a human-readable maintainable results, etc.

1.5 Environments, Runtime Support and Cross-Compilation

Programs are rarely work in isolation; as a rule they rely on a certain environment: operation system, system- and user-level libraries, etc. An important component of this environment is a *runtime support library*. This library contains an implementation of certain programming language constructs which are more beneficent to implement as a library then to built in a compiler itself, for example, memory managements and synchronization primitives, etc. The invocation of these primitives is generated by a compiler; as a rule, they can not be accessed from programs on the user level. In contrast, *standard library* contains an initial implementation of useful data structures and functions in terms of the source programming language, for example, input-output functions, standard data types, collection implementations, etc.

Since compiler is a program itself, it also requires a certain environment, called *compilation environment*. On the other hand, the environment for which compiler generates its output is called *target environment*. Usually these two coincide — for example, out $\mathcal{X}^a\mathcal{M}^a$ compiler works under Linux on x86 processor, and generates programs which work under Linux on x86.

However, in general case, compilation environment can be different from the

target one. In this case the compiler is called *cross-compiler*. For example, we could rewrite our $\mathcal{X}^a\mathcal{M}^a$ compiler into a cross-one, which, still running on x86, would generate code for ARM.

A typical scenario for cross-compilation is *bootstrapping* (see below) or the development of embedded systems when the target platform is not enough performant/well-equipped to support the execution of a compiler.

1.6 Bootstrapping

An interesting (and practically important) operation involving compilers is their *bootstrapping*. This term denotes the implementation of a compiler in its own source languages (i.e. when implementation language coincides with the source one). The term itself originates from an idiom describing a process of pulling oneself up by the hooks on the back of their own boots.

Bootstrapping

From Wikipedia, the free encyclopedia

For other uses, see Bootstrapping (disambiguation).

In general, **bootstrapping** usually refers to a self-starting process that is supposed to continue or grow without external input.

Contents [show]

Etymology [edit]

Tall boots may have a tab, loop or handle at the top known as a bootstrap, allowing one to use fingers or a boot hook tool to help pulling the boots on. The saying "to pull oneself up by one's bootstraps"[1] was already in use during the 19th century as an example of an impossible task. The idiom dates at least to 1834, when it appeared in the *Workingman's Advocate*: "It is conjectured that Mr. Murphee will now be enabled to hand himself over the Cumberland river or a barn yard fence by the straps of his boots."[2] In 1860 it appeared in a comment on philosophy of mind: "The attempt of the mind to analyze itself [is] an effort analogous to one who would lift himself by his own bootstraps."[3] Bootstrap as a metaphor, meaning to better oneself by one's

A pair of boots with one bootstrap visible

The bootstrapping of a compiler for a new language (for which no compiler yet exists) is comprised of its implementation in some other language and then rewriting it in this new language using just written compiler. For example, in such a way the $\mathcal{X}^a\mathcal{M}^a$ compiler was acquired (**not really yet**): initially it was implemented in OCaml and then reimplemented in $\mathcal{X}^a\mathcal{M}^a$ itself.

If some compiler for the language of interest already exists, but works on some other platform, then it is possible for implement a *cross-compiler* which works on that platform but generates code for the platform of interest. Then this compiler can be compiled by itself — this is the conventional way, for example, to port GCC compilers to other platforms.

Finally, if nothing useful exists beside the assembler for the platform of interest, then this assembler itself can be used to implement a small subset of the desirable language; then this subset can be used to implement a wider subset,

and so on. This is how modern compiler/language zoo was built historically.

Compiler bootstrapping is an ideologically important step: first, it assesses the expressiveness of the language; second, ir witnesses the maturity of the compiler, since for a new language its compiler, as a rule, is the first large and complex program.

1.7 Complete vs. Partial Correctness

Compiler (as any other translator) has to syntactically transform a program in one language into a program in another, preserving its semantics. In practice, however there are some subtleties.

Let us have a program **p**. We denote by

$$x \xrightarrow{\mathbf{p}} y$$

the fact that **p** on input x terminates with the output y (as we know, there may be two other outcomes: **p** crashes on input x or **p** loops forever on input x).

Let us have a compiler, let *source* be some source program, and let *target* be a target program after the compilation. We will say that a compiler is *completely correct* iff for arbitrary *source-target* pair and arbitrary input-output pairs x and y

$$x \xrightarrow{source} y \iff x \xrightarrow{target} y$$

In other words, the behaviour of the source and compiled programs is indistinguishable: on the same input both either terminate with the same results, or crash/loop.

Consider, however, the following simple program:

```
# include <stdio.h>

static int d = 0;

int main (int argc, char *argv[]) {
  int x;

  x = argc / d;

  return 255;
}
```

A brief analysis reveals that this program has to crash on each input. Indeed, the variable "`d`" has initial value zero and cannot be reassigned elsewhere (due to " **static** " storage class), and division by zero gives a runtime error. We could, alternatively, try to compile this program with "`gcc -O0`" command and see that it, indeed, crashes.

On the other hand, a more careful analysis shows that the value of the variable "`x`", during the computation of which the error occurs, actually is never used, so its computation can be completely omitted. And, indeed, if we compile this program with the command "`gcc -O3`", it terminates successfully with return code 255! This is because the option "`-O3`" turns on the optimizations, and one of those — *dead code elimination* — does exactly what we envisioned.

Thus, we can see that in practice target programs not always completely equivalent to the source ones: they can deliver some results on inputs, for which the source ones loop or crash. This property is called *partial correctness*, and can be formally specified as follows: for arbitrary input-output values x and y, and for arbitrary source and target programs *source* and *target*

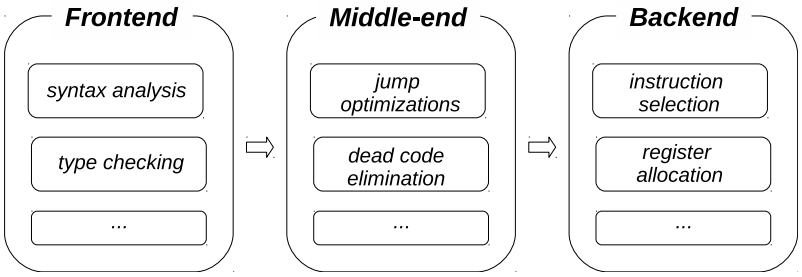
$$x \xrightarrow{source} y \Longrightarrow x \xrightarrow{target} y$$

This means, that when the source program terminates with a definite result, the target also does so with the same result; however, the target one can still deliver some output values for inputs on which the source program is not defined. In other words, compilers are allowed to *extend the domains* of programs they compile. The majority of compilers (and other tools as well) are partially correct.

1.8 Compiler Architecture

The majority of real compilers share the same architectural features. They consist of a number of *passes*, each of which takes as input and returns as a result a certain *intermediate representation*, *IR*, (not necessarily the same) of a program being compiled. The concrete set of these passes and concrete forms of intermediate representations vary from a compiler to a compiler, but there are still some common representation which can be found throughout various implementations: *abstract syntax tree*, *three-address code*, *stack machine code*, *static single assignment* (SSA), etc. Some of these forms of representation will be utilized in our compiler, some others will be left out unused.

All the diversity of passes can be subdivided in a few categories.



First, the *frontend* of a compiler collects a number of *analyzing* passes: syntax analysis, type inference/checking, name resolution, etc.

Then, the *backend* of a compiler contains code generation passes: instruction selection, register allocation, instruction scheduling, low-level optimizations.

Finally, there can be *middle-end*, which comprised of various machine-independent optimising passes such as jump optimizations, dead code elimination, loop invariant code motion, common subexpression elimination, etc.

Most compilers allow to peep on what's going on under the hood. For example, `GCC` accepts the option `-ftime-report`, which makes it output the time taken by each pass performed. An example of such an output is shown below:

```
Execution times (seconds)
 phase setup             :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall    1179 kB (25%) ggc
 phase parsing           :   0.03 (33%) usr   0.01 (100%) sys   0.04 (40%) wall    1197 kB (25%) ggc
 phase opt and generate  :   0.06 (67%) usr   0.00 ( 0%) sys   0.06 (60%) wall    2328 kB (49%) ggc
 callgraph optimization  :   0.01 (11%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall       0 kB ( 0%) ggc
 cfg cleanup             :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.01 (10%) wall       0 kB ( 0%) ggc
 trivially dead code     :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.01 (10%) wall       0 kB ( 0%) ggc
 df reaching defs        :   0.01 (11%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall       0 kB ( 0%) ggc
 alias analysis          :   0.01 (11%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall      55 kB ( 1%) ggc
 preprocessing           :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.01 (10%) wall     345 kB ( 7%) ggc
 lexical analysis        :   0.03 (33%) usr   0.01 (100%) sys   0.01 (10%) wall       0 kB ( 0%) ggc
 parser inl. func. body  :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.02 (20%) wall      79 kB ( 2%) ggc
 dominator optimization  :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.01 (10%) wall      45 kB ( 1%) ggc
 forward prop            :   0.01 (11%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall      13 kB ( 0%) ggc
 loop init               :   0.02 (22%) usr   0.00 ( 0%) sys   0.01 (10%) wall     193 kB ( 4%) ggc
 loop fini               :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.01 (10%) wall       0 kB ( 0%) ggc
 final                   :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.01 (10%) wall      48 kB ( 1%) ggc
 TOTAL                 :   0.09             0.01             0.10              4714 kB
```

1.9 Beyond Compilers

While compilers, indeed, transform source programs to machine code, the code they output, as a rule, cannot be directly run on hardware. The reason is that it still contains certain abstractions the hardware cannot deal with. In particular, compilers usually generate assembler code with *symbolic names*. This approach plays an important role in supporting *separate compilation* — a feature which allows to combine programs from a number of precompiled modules. The majority of application programs make use of various libraries; as a rule, the source code of these libraries is not compiled alongside with the application itself, but *linked* at post-compilation time, thus greatly reducing the time required for build. To make it possible, compilers produce not ready-to-run binary code, but so-called *object files* which, besides machine code, contain supplementary *metadata* to make linking possible. Thus, on the way to the real executable binary the output which compilers provide is transformed again a number of times.

In the most general case, the compiler generates a textual assembler program. Then the following tools can be used:

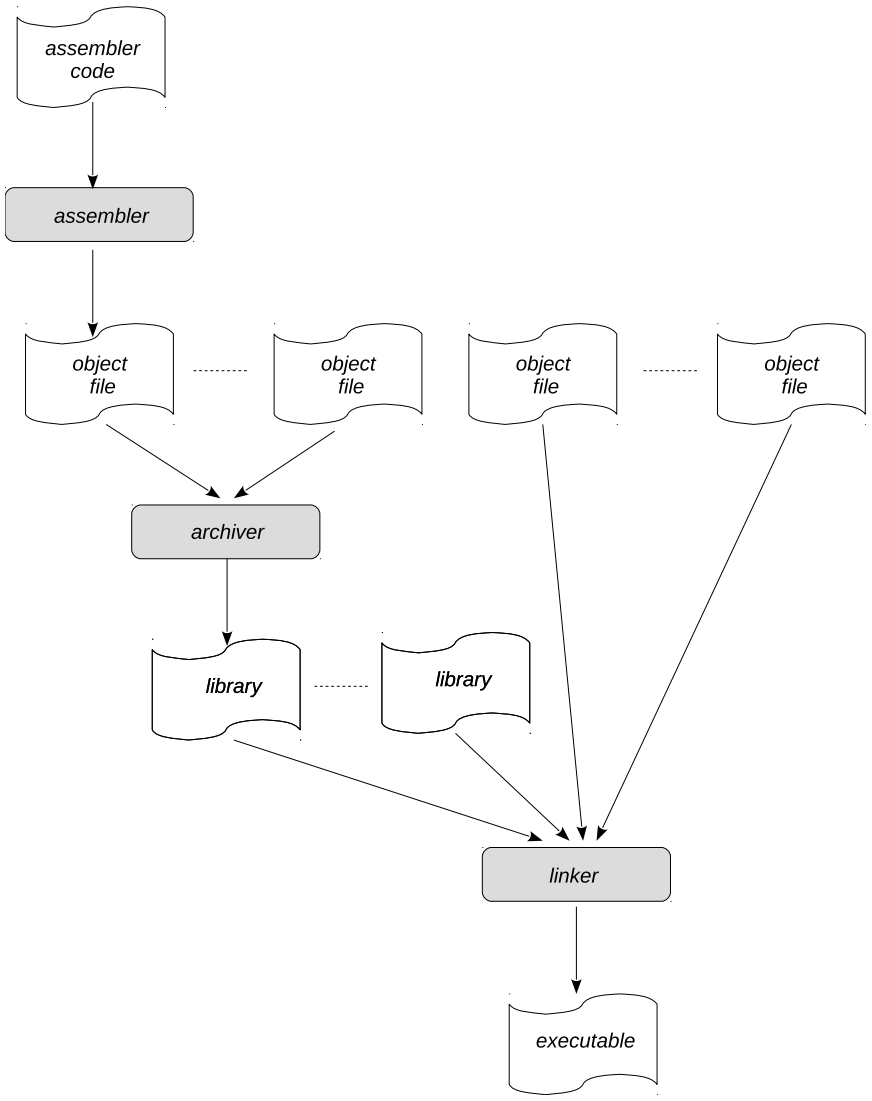


Figure 1.2: Toolchain

- *assembler* which reads assembler programs and outputs object files;
- *archiver/librarian* which combines multiple object files into one archive/library file;
- *linker* which combines multiple object and library files into one executable.

These tools together with a compiler form so-called *toolchain* (Fig. 1.2). When a new processor comes to market it is expected from the vendor to supply a conventional toolchain for the developers. In UNIX-like systems the conventional toolchain consists of separate programs **cpp** (C preprocessor), **cc** (C compiler), **as** (assembler), **ld** (linker) and **ar** (archiver). GCC compiler implements only two first of these components — **cpp** and **cc**, — and invokes others to complete the compilation process depending on what options were specified by users. The top-level program **gcc** itself is just a *driver* which controls the execution of other tools of the toolchain.

1.10 The $\chi^a\mathcal{M}^a$ Compiler

Now, when we discussed a little bit how compilers work, let's have a look at $\chi^a\mathcal{M}^a$ compiler, which will be our main tool throughout the course, and which we will be implementing.

The $\chi^a\mathcal{M}^a$ compiler is organized much simpler than the majority of industrial-tier compilers like `GCC`, which we already mentioned multiple times (see Fig. 1.3).

The source file with a $\chi^a\mathcal{M}^a$ program is parsed by a syntax analyser which converts it into an abstract syntax tree, or AST. An example of a program and its AST (actually, an *HTML-rendering* of its AST) is shown below:

```
printf ("Hello, world!\n")
```

- Scope
  - []
  - Call
    - Var
      - "printf"
    - list
      - String
        - "Hello, world!\\n"

AST is one of the most important program representations, and its use in compilers and other tools is ubiquitous.

The next component, which is rarely implemented in real-world compilers, is a source-level *interpreter*. We consider this object in details later, for now it is sufficient to know that interpreter is a component which directly runs a program in some representation — in our case, in the form of AST, — according to the semantics of the language. The presence of interpreter plays an important role from both educational and technological standpoints. First, the implementation of interpreter facilitates the internalization of formal semantics description method which we use, namely — big-step operational semantics. Next, it allows

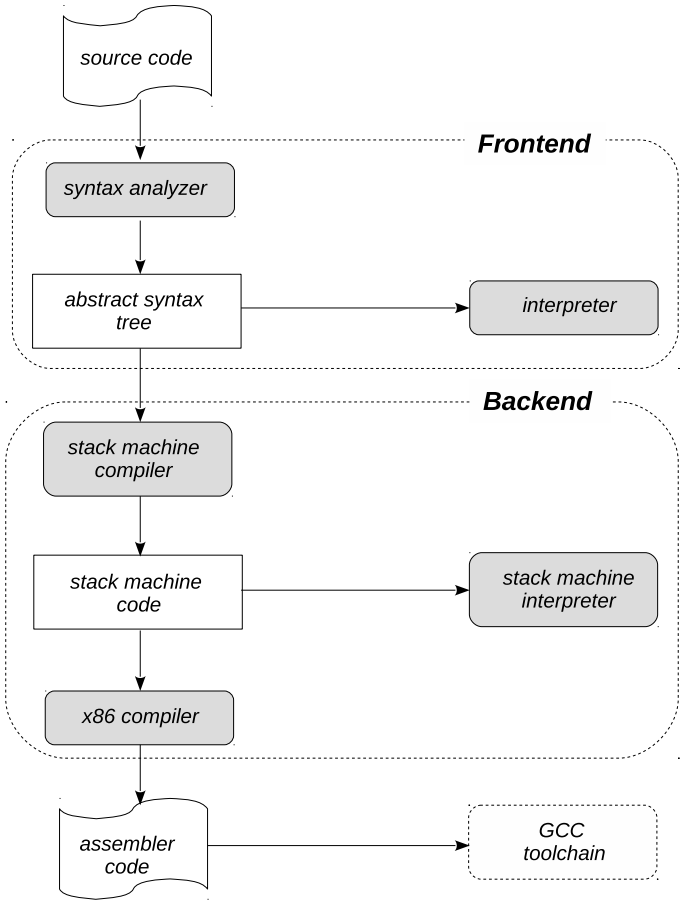


Figure 1.3: The Structure of $\lambda^a\mathcal{M}^a$ Compiler

to find and eliminate some errors at early stage. The implementation of interpreter is rather a simple task, and it is advantageous to be capable of running program at early stages of compiler implementation.

Then, there is a compiler of AST into *stack machine* code. This machine resembles a simplified model of actual hardware processor; thus, on one hand, to generate stack machine code a similar set of tasks has to be solved; on the other, these solutions are a bit simpler than for an actual hardware. For our example program the corresponding stack machine code looks like

```
LABEL ("main")
BEGIN ("main", 2, 0, [], [], [])
STRING ("Hello, world!\\n")
CALL ("Lprintf", 1, false)
END
```

Generated stack machine code can then be run on the *stack machine in-*

terpreter. Similarly to source-level interpreter case, the capability to run stack machine code makes it possible to develop and debug stack machine compiler in isolation.

Finally, the last component of the compiler is code generator for `x86` processor which transforms stack machine code into `x86` assembler program; this program is then passed to the GCC infrastructure to be finally transformed into an object module. An example of `x86` assembler listing for our example program is shown below:

```
          .globl     main
          .data
string_0:.string "Hello, world!\n"
main:
# BEGIN ("main", 2, 0, [], [], []) /
# STRING ("Hello, world!\\n") /
          movl       $string_0,      %ebx
          pushl      %ebx
          call       Bstring
          addl       $4,     %esp
          movl       %eax,   %ebx
# CALL ("Lprintf", 1, false) /
          pushl      %ebx
          call       Lprintf
          addl       $4,     %esp
          movl       %eax,   %ebx
# END /
          movl       %ebx,   %eax
Lmain_epilogue:
          movl       %ebp,   %esp
          popl       %ebp
          xorl       %eax,   %eax
          ret
```

Thus, from the architectural point of view, syntax analyser constitutes a frontend, while compilers for stack machine and `x86` — a backend.

Chapter 2

Semantics and Interpreters

In this section we set the foundations for formal semantics which will be used in the rest of the course. We also discuss the relation between programs and their representation in a concrete data domain, introduce the notion of interpreter and consider some sample languages, their semantics and techniques for interpreter implementations.

2.1 Languages and Semantics

We consider a programming language $\mathcal{L}$ as a (countable) set of programs

$$\mathcal{L} = \{\mathbf{p}_1, \mathbf{p}_2, \dots\}$$

To give a *semantics* for the language $\mathcal{L}$ is to specify two objects:

- a *semantic domain* $\mathcal{D}$;
- a *total* mapping $[\![\bullet]\!]_{\mathcal{L}} : \mathcal{L} \to \mathcal{D}$.

Thus, for a program

$$\mathbf{p} \in \mathcal{L}$$

its semantics is just a

$$[\![\mathbf{p}]\!]_{\mathcal{L}} \in \mathcal{D}$$

When the language is easily deducible from the context we will omit the subscript and write simply $[\![\bullet]\!]$.

By claiming the totality of $[\![\bullet]\!]$ we make sure that any program has some semantics. The nature of semantic domain $\mathcal{D}$ essentially defines the nature of $[\![\bullet]\!]_{\mathcal{L}}$; for example, we can set

$$\mathcal{D} = \{\text{☠}\}$$

and (the only choice)

$$[\![\mathbf{p}]\!] = \text{🙀}$$

for every $\mathbf{p} \in \mathcal{L}$. We admit that this particular semantics might not be very useful (this, however, depends on the nature of $\mathcal{L}$), but the important observations are that

- the choice of $\mathcal{D}$ has to be made;
- there might be (and, *as a rule*, are) multiple semantics for a given language.

These various semantics for a language may reflect its various properties (and we will see some of those in a little while); however, as a rule, one particular semantics is chosen as the "standard" one.

2.2 Data Domain

If we speak of a general-purpose programming language and interested in its execution semantics then, probably, we may consider choosing the semantic domain to be the set of all *partially-recursive* functions and $[\![p]\!]$ to be the function p evaluates. This choice, however, would be too high-level and abstract for our purposes.

To be more concrete, we first choose a *data domain* $\mathfrak{D}$ to be the set of all reasonable data values the programs can take as inputs and produce as outputs; then the semantic domain for executable semantics would be

$$\mathcal{D} : \mathfrak{D} \to \mathfrak{D}$$

Thus, executable semantics maps programs to data-processing functions.

In order to make further progress we stipulate the following properties of the data domain. First, we require its *closedness under product*:

$$\mathfrak{D} \times \mathfrak{D} \subset \mathfrak{D}$$

In other words, $\mathfrak{D}$ contains all pairs, triples, etc. of its values.

The next requirement follows from out intention to make *metaprogramming* possible. In short, the idea behind metaprogramming is to use *programs* as *data*. Indeed, all programming tools follow this approach: a compiler takes a program as input and returns (another) program as output, etc. This can be done only if programs can be *encoded* somehow in the data domain. We consider some concrete encodings, using $\mathcal{X}^a\mathcal{M}^a$ data domain later; for now we assume that for arbitrary programming language $\mathcal{L}$ the set $\mathfrak{D}$ contains the *representations* of all programs in $\mathcal{L}$:

$$\forall \mathcal{L} \,.\, \mathcal{L} \subset \mathfrak{D}$$

We agreed above to consider programming languages as sets of programs; we could, equivalently, consider them as sets of *representations* of programs in some universal data domain. Since the correspondence between programs and their representations is one-to-one this convention would not hinder any follow-up reasonings. From now on we will not distinguish programs (abstract objects) from their representations (concrete objects) in $\mathfrak{D}$. Note, there can be multiple representations within one data domain, but all of them are "equivalent" in the sense that each pair of them admits an unambiguous conversions in both directions.

2.3 Semantic Properties

Let us have a language $\mathcal{L}$ and its *two* semantics

$$\begin{array}{rcl} [\![\bullet]\!] & : & \mathcal{L} \to \mathcal{D} \\ [\![\bullet]\!]' & : & \mathcal{L} \to \mathcal{D} \end{array}$$

with the *same* semantic domain. We say that these semantics are *equivalent* if and only if for arbitrary program $\mathbf{p} \in \mathcal{L}$

$$[\![\mathbf{p}]\!] = [\![\mathbf{p}]\!]'$$

In other words, equivalent semantics assign to each program in the language the same element of semantic domain. A question might arise why would we need two equivalent semantics; wouldn't the single one be sufficient? The answer is that there are multiple ways of describing semantics, and these different ways have different properties which make them preferable in different settings. Sometimes it is desirable to reformulate the semantics in different terms. By proving the equivalence between the two we can justify that we still deal with the language with the same semantic properties.

Note, when the semantic domain is domain of functions $\mathfrak{D} \to \mathfrak{D}$, the equation above denotes the equality of functions: for arbitrary $x, y \in \mathfrak{D}$

$$[\![\mathbf{p}]\!]' x = y \Leftrightarrow [\![\mathbf{p}]\!] x = y$$

In other words, in both semantics $\mathbf{p}$ is defined on exactly the same inputs and for each of these inputs it provides the same outputs.

Another important property is *equivalence of programs* within the same semantics. We say that $\mathbf{p}_1$ is (semantically) equivalent to $\mathbf{p}_2$ (notation: $\mathbf{p}_1 \equiv \mathbf{p}_2$) if and only if

$$[\![\mathbf{p}_1]\!] = [\![\mathbf{p}_2]\!]$$

Thus, equivalent programs have the same semantics. The equivalence of *different* programs within the same semantics plays a crucial role in justifying the correctness of program transformations. Let us have some transformation of programs into programs:

$$f : \mathcal{L} \to \mathcal{L}$$

We say that f is *semantically correct* if and only if for all programs $\mathbf{p}$

$$[\![f\,(\mathbf{p})]\!] = [\![\mathbf{p}]\!]$$

Thus, equivalent transformations do not change the semantics of programs; they can, however, change other their properties.

When the semantic domain is domain of functions, the equation above, again, denotes the equality of functions; additionally, the following important notion can be introduced in this case. We say that f is *partially correct* if and only if for all programs $\mathbf{p}$ and all $x, y \in \mathfrak{D}$

$$[\![\mathbf{p}]\!]\, x = y \Rightarrow [\![f\,(\mathbf{p})]\!]\, x = y$$

The difference between correctness and partial correctness is that in the former case the programs are defined for exactly the same inputs, while in the latter the transformed program can be defined even if the original one in not. To some extent this is how optimizing transformations work, as we've seen in the previous chapter.

Finally, there can be transformations between *different* languages with the *same* semantic domain. Let us have two languages $\mathcal{L}$ and $\mathcal{M}$ and let their semantics be

$$\begin{array}{rcl} [\![\bullet]\!]_{\mathcal{L}} & : & \mathcal{L} \to \mathcal{D} \\ [\![\bullet]\!]_{\mathcal{M}} & : & \mathcal{M} \to \mathcal{D} \end{array}$$

We say that a transformation

$$f : \mathcal{L} \to \mathcal{M}$$

is *semantically correct* if and only if for all programs $\mathbf{p}$

$$[\![\mathbf{p}]\!]_{\mathcal{M}} = [\![f\,(\mathbf{p})]\!]_{\mathcal{L}}$$

And, again, when $\mathcal{D}$ is a domain of functions the equation above denotes the equality of functions, and the notion of partial correctness arises. One example of transformation between languages is compilation; as we already know, compilers as a rule are partially correct.

2.4 Interpreters, Compilers, Specializers

We already mentioned compilation as a syntactic transformation from one language to another; we also talked of compilers as programs which implement compilation. Here we consider them and some other useful transformations in the form of programs in more details.

Let $\mathcal{L}$ and $\mathcal{M}$ be two languages, and

$$\begin{array}{rcl} [\![\bullet]\!]_{\mathcal{L}} & : & \mathcal{L} \to \mathfrak{D} \to \mathfrak{D} \\ [\![\bullet]\!]_{\mathcal{M}} & : & \mathcal{M} \to \mathfrak{D} \to \mathfrak{D} \end{array}$$

— their semantics. An *interpreter* for language $\mathcal{L}$, written in language $\mathcal{M}$, is a program $\mathbf{int}^{\mathcal{L}}_{\mathcal{M}} \in \mathcal{M}$, such that for each program $\mathbf{p}_{\mathcal{L}} \in \mathcal{L}$ and each data value $x \in \mathfrak{D}$

$$[\![\mathbf{int}^{\mathcal{L}}_{\mathcal{M}}]\!]_{\mathcal{M}}\,(\mathbf{p}_{\mathcal{L}} \times x) = [\![\mathbf{p}_{\mathcal{L}}]\!]_{\mathcal{L}}\,(x) \qquad (\star)$$

To some extent an interpreter *implements* the semantics of a programming language: given a program and its input it provides exactly the same result this program calculates. Of course there can be many interpreters for a given pair of languages; any program $\mathbf{i}_{\mathcal{M}}$ satisfying equation $(\star)$, e.g. such than

$$\forall \mathbf{p}_{\mathcal{L}},\, x \in \mathfrak{D} \,.\, [\![\mathbf{i}_{\mathcal{M}}]\!]_{\mathcal{M}}\,(\mathbf{p}_{\mathcal{L}} \times x) = [\![\mathbf{p}_{\mathcal{L}}]\!]_{\mathcal{L}}\,(x)$$

is an interpreter.

A particular interesting kind of interpreter is *self*-interpreter $\mathbf{int}^{\mathcal{L}}_{\mathcal{L}}$, i.e. an interpreter which interprets the language of its own implementation. In the computability theory such interpreters are known under the name "*universal functions*", and it is proven than universal functions exist for all Turing-complete languages.

Another interesting program is, of course, a compiler. Given languages $\mathcal{L}$, $\mathcal{M}$, and $\mathcal{N}$, a compiler from $\mathcal{L}$ to $\mathcal{N}$, written in a language $\mathcal{M}$, is a program $\mathbf{comp}^{\mathcal{L}\to\mathcal{N}}_{\mathcal{M}} \in \mathcal{M}$ such that for all programs $\mathbf{p}_{\mathcal{L}} \in \mathcal{L}$ and all input data values $x \in \mathfrak{D}$ the following equation holds:

$$[\![[\![\mathbf{comp}^{\mathcal{L}\to\mathcal{N}}_{\mathcal{M}}]\!]_{\mathcal{M}}\,(\mathbf{p}_{\mathcal{L}})]\!]_{\mathcal{N}}\,(x) = [\![\mathbf{p}_{\mathcal{L}}]\!]_{\mathcal{L}}\,(x) \qquad (\star\star)$$

Indeed, a compiler takes a program representation $\mathbf{p}_{\mathcal{L}}$ as input and produces another program, $[\![\mathbf{comp}^{\mathcal{L}\to\mathcal{N}}_{\mathcal{M}}]\!]_{\mathcal{M}}\,(\mathbf{p}_{\mathcal{L}})$, this time in the language $\mathcal{N}$, which gives exactly the same result as $\mathbf{p}_{\mathcal{L}}$ for any input x. And, again, any program $\mathbf{c}_{\mathcal{M}}$ satisfying the equation $(\star\star)$ is a compiler.

Finally, there can be a program called *specializer* $\mathbf{spec}^{\mathcal{L}}_{\mathcal{M}}$, written in a language $\mathcal{M}$ for a language $\mathcal{L}$, such that for all programs $\mathbf{p}_{\mathcal{L}} \in \mathcal{L}$ and all data values $x, y \in \mathfrak{D}$

$$[\![[\![\mathbf{spec}^{\mathcal{L}}_{\mathcal{M}}]\!]_{\mathcal{M}}\,(\mathbf{p}_{\mathcal{L}} \times x)]\!]_{\mathcal{L}}\,(y) = [\![\mathbf{p}_{\mathcal{L}}]\!]_{\mathcal{L}}\,(x \times y) \qquad (\star\star\star)$$

Informally, a specializer takes as input a program $\mathbf{p}_{\mathcal{L}}$ and *one* of its inputs x and builds a program in the same language $\mathcal{L}$ which takes y — the remaining inputs of $\mathbf{p}_{\mathcal{L}}$, — and provides exactly the same result as $\mathbf{p}_{\mathcal{L}}$ for both x and y. The existence of specializers is, again, guaranteed by the computability theory (*Kleene s-m-n–theorem*).

2.5 Futamura Projections

We now study a few elegant theoretical constructs which connect together the notions of interpreters, compilers and specializers. To simplify the presentation we introduce the following denotation

$$p_{\mathcal{L}} = [\![\mathbf{p}_{\mathcal{L}}]\!]_{\mathcal{L}}$$

for a program $\mathbf{p}_{\mathcal{L}}$. Thus, while $\mathbf{p}_{\mathcal{L}}$ is a program in a language $\mathcal{L}$ (i.e. a syntactic object), $p_{\mathcal{L}}$ is its semantics (a function in the data domain).

Our first step is to apply a specializer $\mathbf{spec}^{\mathcal{L}}_{\mathcal{M}}$ to some interpreter $\mathbf{int}^{\mathcal{N}}_{\mathcal{L}}$ and some program $\mathbf{p}_{\mathcal{N}}$ it can interpret:

$$[\![\underline{spec^{\mathcal{L}}_{\mathcal{M}}\,(\mathbf{int}^{\mathcal{N}}_{\mathcal{L}} \times \mathbf{p}_{\mathcal{N}})}]\!]_{\mathcal{L}}\,(x) = int^{\mathcal{N}}_{\mathcal{L}}\,(\mathbf{p}_{\mathcal{N}} \times x) = [\![\underline{\mathbf{p}_{\mathcal{N}}}]\!]_{\mathcal{N}}\,(x) \qquad \text{(I)}$$

The first equality follows immediately from $(\star\star\star)$ while the second — immediately from $(\star)$. Let's now look at the underlined parts. The *right* one is, obviously, $\mathbf{p}_{\mathcal{N}}$, a program in the language $\mathcal{N}$. The *left* one is some program in the language $\mathcal{L}$. The equation itself states that the semantic of these two programs give the same value for every input x. In other words, these two programs are equivalent. This is the first Futamura projection:

> *The specialization of an interpreter for a program gives the representation of this program in the language of interpreter implementation.*

Next, let's specialize a specializer for an interpreter:

$$[\![[\![\underline{spec^{\mathcal{M}}_{\mathcal{K}}\,(\mathbf{spec}^{\mathcal{L}}_{\mathcal{M}} \times \mathbf{int}^{\mathcal{N}}_{\mathcal{L}})}]\!]_{\mathcal{K}}\,(\mathbf{p}_{\mathcal{N}})]\!]_{\mathcal{L}}\,(x) = [\![spec^{\mathcal{L}}_{\mathcal{M}}\,(\mathbf{int}^{\mathcal{N}}_{\mathcal{L}} \times \mathbf{p}_{\mathcal{N}})]\!]_{\mathcal{L}}\,(x) = [\![\mathbf{p}_{\mathcal{N}}]\!]_{\mathcal{N}}\,(x) \qquad \text{(II)}$$

The first equation, again, immediately follows from $(\star\star\star)$, while the second — from (I). If we look at the underlined part long enough, it becomes evident that it is a program in the language $\mathcal{M}$ which satisfies the equation $(\star\star)$, i.e. a compiler $\mathbf{comp}^{\mathcal{N}\to\mathcal{L}}_{\mathcal{M}}$. This is a second Futamura projection:

> *The specialization of a specializer for an interpreter gives a compiler from the interpreting language to the language of interpreter implementation.*

Finally, we can specialize a specializer to a specializer:

$$[\![spec^{\mathcal{K}}_{\mathcal{T}}\,(\mathbf{spec}^{\mathcal{M}}_{\mathcal{K}} \times \mathbf{spec}^{\mathcal{L}}_{\mathcal{M}})]\!]_{\mathcal{T}}\,(\mathbf{int}^{\mathcal{N}}_{\mathcal{L}}) = spec^{\mathcal{M}}_{\mathcal{K}}\,(\mathbf{spec}^{\mathcal{L}}_{\mathcal{M}} \times \mathbf{int}^{\mathcal{N}}_{\mathcal{L}}) \qquad \text{(III)}$$

The equation immediately follows from $(\star\star)$; its right part, according to (II), is $\mathbf{comp}^{\mathcal{N}\to\mathcal{L}}_{\mathcal{M}}$. This is the third Futamura projection:

> *The specialization of a specializer for a specializer gives a compiler generator which for an interpreter generates a compiler from the interpreting language to the language of interpreter implementation.*

Futamura projections are named after Y.Futamura, who described the first two of them in the beginning of 1970s. All three projections were independently discovered by V.Turchin and A.Ershov, who gave them their current name.

The beauty of Futamura projections is that they give a rather simple equations for rather complex tools like compilers and compiler generators. However, this immediately raises a question if one can indeed acquire these tools using such a high-level description.

Let's assume that we are going to use Futamura projections to implement a compiler from $\lambda^a\mathcal{M}^a$ to `x86`. Then we, first, need an interpreter $\mathbf{int}^{\lambda^a\mathcal{M}^a}_{\texttt{x86}}$ for $\lambda^a\mathcal{M}^a$ written in `x86` assembler. The task of implementing such an interpreter, while involving some low-level programming, does not look very challenging. Then, we need a specializer $\mathbf{spec}^{\texttt{x86}}_{\mathcal{L}}$ for `x86` assembler written in some language $\mathcal{L}$, not necessarily $\lambda^a\mathcal{M}^a$. We can choose any suitable language for this purpose. Having both at hands, we will be able to compile $\lambda^a\mathcal{M}^a$-programs to `x86` code using the first Futamura projection:

$$spec^{\texttt{x86}}_{\mathcal{L}}\,(\mathbf{int}^{\lambda^a\mathcal{M}^a}_{\texttt{x86}} \times \mathbf{p}_{\lambda^a\mathcal{M}^a}) = \mathbf{p}_{\texttt{x86}}$$

And here comes the hard part: the simplest possible specializer (for example, that guaranteed by the *s-m-n*–theorem) would produce a very poor machine code; it would, in fact, just link the interpreter with the program, which invalidates the very idea of compilation. In order to acquire a decent result, the specializer has to be non-trivial. The task of developing a non-trivial specializer even for the first Futamura projection is non-trivial as well; nevertheless there are frameworks where this task is solved. For example, `GraalVM` uses the first Futamura projection as a tool for language bootstrapping.

If we move higher in the Futamura projection hierarchy we would need at least one additional specializer $\mathbf{spec}^{\mathcal{L}}_{\mathcal{L}}$, this time for the language $\mathcal{L}$; it can be written in the $\mathcal{L}$ as well. This specializer has to be even more advanced than $\mathbf{spec}^{\texttt{x86}}_{\mathcal{L}}$ since we expect it to decently specialize more complicated program, than interpreters.

Finally, for the third Futamura projection we need even more advanced specializer since it has to be capable of decently specialize specializer for a specializer. Note, is the third Futamura projections the last two specializers need not necessarily be the same programs, but it is very appealing from both scientific and aesthetic standpoints to have the single, *self-applicable*, specializer.

Thus, using Futamura projections beyond the first one in practice is still a hard venture. In the middle of 1980s all three projections were implemented by the group led by N.Jones, but this was rather a proof-of-concept than a working industrial technology.

Chapter 3

Straight Line Programs

In this chapter we introduce the language of *straight line* programs which can be considered as a smallest non-trivial subset of $\mathcal{X}^a\mathcal{M}^a$. In this subset all programs are executed sequentially statement by statement with no branching. Thus, any program either comes to an end or stops due to an error, but cannot loop forever. We use this simple language to showcase all the ingredients of our approach to language description: abstract and concrete syntax specification, denotational and operational semantics, etc. We also introduce some components of the compiler we will be working on: source-level interpreter, stack machine compiler and interpreter, and `x86` code generator.

3.1 Expressions

Syntactically, our language encorporates two *syntactic categories*: expressions and statements. We start from describing so-called *abstract syntax* for the expression category. We consider a countable set of *variables*

$$\mathscr{X} = \{x_1, x_2, \dots\}$$

and a set of *binary operators*

$$\otimes = \{+, -, \times, /, \%, <, \leq, >, \geq, =, \neq, \vee, \wedge\}$$

which contains all thirteen built-in $\mathcal{X}^a\mathcal{M}^a$ operators. Then, the category of expressions $\mathscr{E}$ can be defined by the following recursive scheme:

$$\begin{array}{rcl} \mathscr{E} & = & \mathscr{X} \\ & & \mathbb{N} \\ & & \mathscr{E} \otimes \mathscr{E} \end{array}$$

This scheme defines a countable set of *labeled ordered trees* of finite height: each node of such a tree is labeled, and for any node the order of its immediate subtrees is essential. The simplest trees of this form are just leaves labeled with

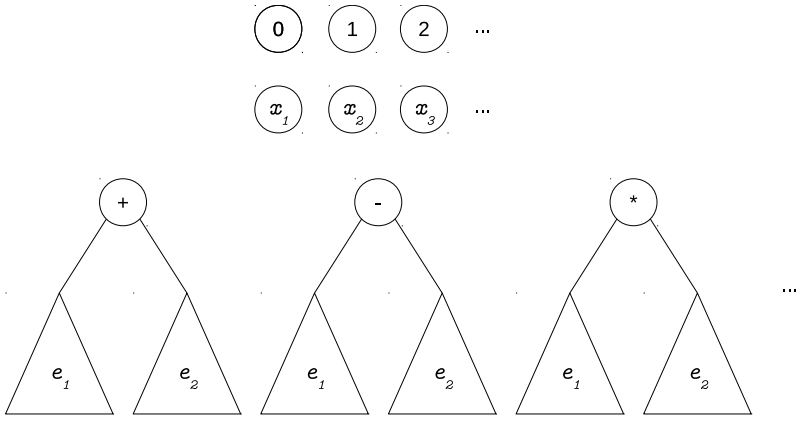


Figure 3.1: Abstract Syntax for Expressions

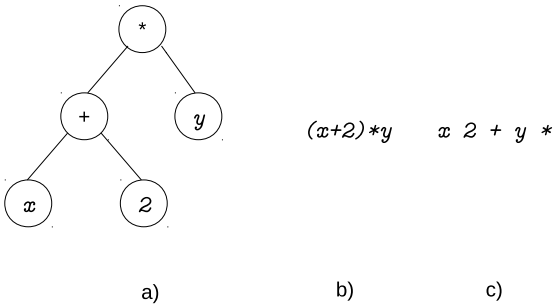


Figure 3.2: Various Concrete Syntaxes for Expression Language

either variables or natural numbers; we simply write $\mathscr{X}$ or $\mathbb{N}$ in the first two lines of definition of $\mathscr{E}$, but actually we mean tree nodes *labeled* by the symbols of these sets. As for the third line, it stipulates that for arbitrary two expressions e_1, $e_2 \in \mathscr{E}$ a tree with a root labeled with any symbol from $\otimes$ and immediate subtress e_1 and e_2 is also expression (see Fig. 3.1).

We call this definition *abstract* syntax because it describes nothing more than a subordination between elementary constructs. In order to represent expressions in some medium, however, we need *concrete* syntax; it is easy to anticipate that there can be multiple concrete syntaxes for given abstract one. In Fig. 3.2 we give some examples of those for expressions: the first (a) consists of graphical elements such as circles, lines, texts, etc. Another one (b) is the familiar *infix notation* which includes numbers, letters, binary operators and brackets. Yet another (but by no means the last one) is *reverse Polish notation* (c), in which binary operators are put *after* the operands they connect. In what follows we will stick with infix notation.

Now we need to define the semantic domain for the semantics of expressions. We already hinted that this domain should be shaped like a set of some data

$$\begin{array}{rcl} [\![z]\!]_{\mathscr{E}} & = & \sigma \mapsto z \\ [\![x]\!]_{\mathscr{E}} & = & \sigma \mapsto \sigma\, x \\ [\![e_1 \otimes e_2]\!]_{\mathscr{E}} & = & \sigma \mapsto [\![e_2]\!]_{\mathscr{E}}\, \sigma \oplus [\![e_2]\!]_{\mathscr{E}}\, \sigma \end{array}$$

Figure 3.3: Denotational Semantics of Expressions

processing functions $\mathfrak{D} \to \mathfrak{D}$; however, we need to be more specific.

As we deal with arithmetic expressions it is rather natural to expect that the results of their evaluation are interger values, i.e. $\mathbb{Z}$ (as we agreed earlier, we assume $\mathbb{Z} \subset \mathfrak{D}$); on the other hand, the value of an expression depends on the values of variables it contains. We can encode these values as a *state* — a function which maps variables to integer values:

$$St : \mathscr{X} \to \mathbb{Z}$$

There is nothing wrong with assuming $St \subset \mathfrak{D}$: as any expression can contain only a finite number of variables we are interested only in states with finite domains which can be encoded, for example, as finite lists of pairs. Thus, finally, we have the following "type" for the semantics of expressions:

$$[\![\bullet]\!]_{\mathscr{E}} : \mathscr{E} \to (St \to \mathbb{Z})$$

3.1.1 Denotational Semantics

There are multiple ways to give the semantics for a language formally. Here we use so-called *denotational* way in which it is immediately specified what object from the semantic domain corresponds to a given language construct. For this concrete language denotational semantics looks simple and natural; however, for more advanced languages more advanced mathematical apparatus would be required. It is also worth mentioning that, as a rule, denotational semantics gives us a very abstract, high-level view on the behavior of programs, which may or may not be desirable from a practical standpoint.

The denotational semantics for expressions is shown in Fig. 3.3. We give here three equations, one for each syntactic form. In the right-hand side of each equation we immediately give the object (a function from states to integers) which corresponds to the semantics of the expression in the left-hand side. The notation $\star \mapsto \bullet$ is used to denote a function from $\star$ to $\bullet$; we refrain from using the lambda notation since these functions are elements of the *meta-language* (the language we use to describe the semantics), not of the *object* one (the language which semantics is being described).

In the first equation, when the expression is a natural number z, its semantics is a constant function, which for any state σ return just this number z.

When the expression in question is a variable x, its semantics is a function which, given a state σ, returns the value this state assigns to this variable.

Finally, when the expression is a binary operator with two subexpressions e_1 and e_2, its semantics is a function which, given a state σ, first calcalates the values of subexpressions e_1 and e_2 in the same state, and then combines them using a certain arithmetic operator $\oplus$. The correspondence between $\otimes$ and $\oplus$ is described by the following table:

$\otimes$	$\oplus$ in $\lambda^a\mathcal{M}^a$
$+$	$+$
$-$	$-$
$\times$	$*$
$/$	$/$
$\%$	$\%$
$<$	$<$
$>$	$>$
$\leq$	$<=$
$\geq$	$>=$
$=$	$=$
$\neq$	$!=$
$\wedge$	$\&\&$
$\vee$	$!!$

Here we use built-in $\lambda^a\mathcal{M}^a$ binary operators to specify the semantics of $\oplus$; this approach is good enough for now since our primary objective is to implement a reference interpreter in $\lambda^a\mathcal{M}^a$. Later, when we will deal with `x86` codegenerator we will refine the understanding of these operators' semantics.

Note, while the symbols in the first and second columns look similar, they actually have different nature: the left ones are elements of syntax while the right ones — conventional denotations for familiar arithmetic operators. The last equation in Fig. 3.3, thus, is actually a generic one which denotes *thirteen* concrete equations in which $\otimes$ and $\oplus$ are substituted coherently according to the table given above.

We can make two important observations.

First, in given semantics there is a single rule for any "kind" of expression (variable, constant, binary operation), and for each rule its right part defines semantic function unambiguously. Thus, for each expression e and each state σ there is *at most* one integer number y such that

$$[\![e]\!]_{\mathscr{E}}\,\sigma = x$$

This to some extent justifies our desire for $[\![e]\!]_{\mathscr{E}}$ to be a function from states to integers. Indeed, the property we just established is *functionality*. On the other hand, in the domain of semantics the same property has another name: *determinism*. Thus, the semantics in question is deterministic, meaning that evaluating any expression in a given state delivers at most one value. Non-deterministic semantics, according to which there can be multiple such values, seemingly are not compatible with our framework of semantic functions; nev-

ertheless, such semantics exist, and there are ways to fix this incompatibility. Further we will deal only with deterministic semantics.

Another important property is *compositionality*: the semantics of a construct is expressed in the terms of the semantics of its proper subconstructs. Indeed, the first two equations are *axioms*, meaning, that no expressions containing semantic brackets "$[\![\bullet]\!]_{\mathscr{E}}$" occur in the right-hand side; the third equation is not an axiom, but semantic backets are applied only to proper subconstructs (e_1 and e_2) of the construct in question (e). Compositionality is a distinctive property of denotational semantics; using other semantic description styles may or may not result in compositional semantic specification.

When a semantic is compositional, a certain proof principle — *structural induction* — can be used to establish its properties. This technique is essentially a specific kind of mathematical induction applied to *finite trees* rather than to natural numbers. To prove by structural induction that some property holds for all trees one needs to prove, first, that this property holds for all leaves (*base of induction*); then, assuming that the property holds for all trees up to a certain height (*induction hypothesis*) one needs to prove that the property holds for all trees one level higher. We demonstrate the application of this principle by the following example.

3.1.2 Strictness

We are going to prove the *strictness* property of given semantics. It informally means that in order to calculate the value for the whole expression one needs to calculate the values for all its subexpressions. First, we define the following relation "$\preceq$" of one expression being a subexpression of another:

$$\begin{gathered} e' \preceq e' \otimes e \\ e' \preceq e \otimes e' \\ e \preceq e \\ e' \preceq e'' \wedge e'' \preceq e \Rightarrow e' \preceq e \end{gathered}$$

The first two lines define the *immediate* subexpression relation while the last two — its *reflexive-transitive* closure. For example, for the expression $(x+2)*y$ all its subexpression are $(x+2)*y$, $x+2$, y, x, and 2.

Lemma (Strictness)**.** *For all e, σ and x if*

$$[\![e]\!]_{\mathscr{E}}\,\sigma = x$$

then for all $e' \preceq e$ there exists x' such that

$$[\![e']\!]_{\mathscr{E}}\,\sigma = x'$$

Proof. For base case (variable and constant) the lemma holds vacuously since in both cases the only possible subexpressions are these expressions themselves.

Assume the lemma holds for e_1 and e_2; we need to prove it holds for $e_1 \otimes e_2$. By the definition of "$\preceq$" for any $e' \preceq e_1 \otimes e_2$ one of the following is true:

1. $e' = e_1$, or
2. $e' = e_2$, or
3. $e' \preceq e_1$, or
4. $e' \preceq e_1$.

By the condition of lemma we have $[\![e_1 \otimes e_2]\!]_{\mathscr{E}}\,\sigma = x$. By the definitiono of $[\![\bullet]\!]_{\mathscr{E}}$ we have $[\![e_1]\!]_{\mathscr{E}}\,\sigma \oplus [\![e_2]\!]_{\mathscr{E}}\,\sigma = x$. By the definition of $\oplus$ there exist x_1 and x_2 such that

$$\begin{array}{rcl} [\![e_1]\!]_{\mathscr{E}}\,\sigma & = & x_1 \\ [\![e_2]\!]_{\mathscr{E}}\,\sigma & = & x_2 \end{array}$$

If $e' = e_1$ or $e' = e_2$ then the lemma follows immediately. If $e' \preceq e_1$ (or $e' \preceq e_2$) the induction hypothesis can be applied as we just have proven that both e_1 and e_2 have some values being evaluated in the state σ. □

The strictness property, in particular, means that if variable x is undefined in some state σ, then any expression e, containing x, is also undefined in σ. Indeed, if x occurs in e, then, naturally, $x \preceq e$. If $\sigma\,x$ undefined but $[\![e]\!]_{\mathscr{E}}\,\sigma$ not, this would contradict the lemma we've just proven.

Now we can give precise answers to the questions asked in section 1.2. The first question was if the expression $0{*}(\mathtt{x}/0)$ evaluates to zero or undefined. Due to the strictness of our semantics it is undefined in any state. Indeed

$$\mathtt{x}/0 \preceq 0{*}(\mathtt{x}/0)$$

and $[\![\mathtt{x}/0]\!]_{\mathscr{E}}\,\sigma$ is undefined for any state σ since either $\sigma\,\mathtt{x}$ is undefined or $\sigma\,\mathtt{x}$ is defined but $\sigma\,\mathtt{x}$ / 0 is undefined due to the division by zero.

The second question was if $1{+}\mathtt{x}{-}\mathtt{x}$ is equivalent to 1. Again, by the strictness and the fact that $\mathtt{x} \preceq 1{+}\mathtt{x}{-}\mathtt{x}$ we immediately have that for the *empty state* Λ, undefined for every variable x, $[\![1]\!]_{\mathscr{E}}\,\Lambda = 1$ but $[\![1{+}\mathtt{x}{-}\mathtt{x}]\!]_{\mathscr{E}}\,\Lambda$ is undefined. Thus, these two expressions are not equivalent.

We stress that these answers are specific to the concrete semantics of expressions we described; for different semantics the answers can be different.

3.2 Statements

The other syntactic category of the straight line programs language is *statements*. Its abstract syntax is given by the following description:

$$\begin{array}{rcl} \mathscr{S} & = & \mathbf{skip} \\ & & \mathscr{X} := \mathscr{E} \\ & & \mathbf{read}\ \ (\mathscr{X}) \\ & & \mathbf{write}\ \ (\mathscr{E}) \\ & & \mathscr{S}\,;\mathscr{S} \end{array}$$

Here $\mathscr{E}$ and $\mathscr{X}$ stand for the sets of expressions and variables, as in the previous section. The first four lines of abstract syntax description define four *primitive* statements: empty, assignment, input and output respectively. The fifth one makes it possible to combine primitive statements info compositions. The order and subordination of composition counterparts strictly speaking is essential, thus

read (x); (y := x+4; **write** (y))

and

(**read** (x); y := x+4); **write** (y)

are different statements; we use here brackets as elements of concrete syntax to reflect the grouping of subtrees of abstract syntax tree. To reduce the use of brackets we assume that composition by default associates to the *right* (i.e. as in the former example).

3.2.1 Big-Step Operational Semantics

Our next step is to define the semantics for statements. First, as always, we need to specify the semantic domain. From the syntactic form of statements it should be clear that we are dealing with the language with *side effects*: there are read and write statements, which, presumably, communicate with outer world, and assignment, which, presumably, changes the values of variables. These two kinds of side effects play different roles: while read and write make the effect of a program execution *externally observable*, assignments define internal behavior. Thus, essentially different by their internal behavior programs can be indistinguishable while observed externally. We reflect this consideration by choosing the semantic domain to be the set of functions from input streams of integers to output streams of integers

$$\mathbb{Z}^* \to \mathbb{Z}^*$$

We call the pair of input-output streams *world* and define the set of all worlds to be

$$\mathscr{W} = \mathbb{Z}^* \times \mathbb{Z}^*$$

For simplicity, we define the following operations for worlds:

$$\begin{array}{rcl} \mathbf{read}\,\langle xi,\, o\rangle & = & \langle x,\, \langle i,\, o\rangle\rangle \\ \mathbf{write}\,x\,\langle i,\, o\rangle & = & \langle i,\, ox\rangle \\ \mathbf{out}\,\langle i,\, o\rangle & = & o \end{array}$$

The first one, "**read**", takes a world in which input stream xi contains at least one element x and returns a pair of elements: x and the residual word with the first element of input stream removed. The next one, "**write**", to some extent does the opposite: it takes some number x and a world and returns

$$c \xRightarrow[\mathscr{S}]{\textbf{skip}} c \qquad \text{[Skip]}$$

$$\langle \sigma, \omega \rangle \xRightarrow[\mathscr{S}]{\texttt{x := } e} \langle \sigma\,[x \leftarrow [\![e]\!]_{\mathscr{E}}\,\sigma], \omega \rangle \qquad \text{[Assign]}$$

$$\frac{\langle z, \omega' \rangle = \textbf{read}\,\omega}{\langle \sigma, \omega \rangle \xRightarrow[\mathscr{S}]{\textbf{read}\ \ (x)} \langle \sigma\,[x \leftarrow z], \omega' \rangle} \qquad \text{[Read]}$$

$$\langle \sigma, \omega \rangle \xRightarrow[\mathscr{S}]{\textbf{write}\ \ (e)} \langle \sigma, \textbf{write}\,([\![e]\!]_{\mathscr{E}}\,\sigma)\,\omega \rangle \qquad \text{[Write]}$$

$$\frac{c_1 \xRightarrow[\mathscr{S}]{s_1} c' \quad c' \xRightarrow[\mathscr{S}]{s_2} c_2}{c_1 \xRightarrow[\mathscr{S}]{s_1\,;\,s_2} c_2} \qquad \text{[Seq]}$$

Figure 3.4: Big-step operational semantics for statements

a world in which this number is appended to the end of the output stream. Finally, "**out**" just returns the output stream of a given world.

To define the semantics we could use the denotational style as we did for expressions, and it would work just fine. However, as our language starts to evolve, the denotational style will be harder to adjust to meet our intentions; in addition studying yet another way for semantics' specification would make us more versatile.

The technique we are going to use is called *big-step operational semantics*. Unlike denotational case, where a semantic object is immediately given for each syntactic form, operational style involves the construction of intermediate *evaluation relation*, which we denote "$\Rightarrow_{\mathscr{S}}$". From the semantics of expressions we already know the notion of state and how to calculate the values of expressions in given states. Thus, each statement modifies an *enriched* state which consists of a regular state and a world. We call this enriched state *configuration* and define the set of all configurations to be

$$\mathscr{C} = St \times \mathscr{W}$$

Evaluation relation connects a statement and two configurations: *initial* and *final*:

$$\Rightarrow_{\mathscr{S}} \subseteq \mathscr{C} \times \mathscr{S} \times \mathscr{C}$$

We will use infix notation to denote the elements of evaluation relation:

instead of

$$\langle c_1,\, s,\, c_2\rangle \in \Rightarrow_{\mathscr{S}}$$

we will use the form

$$c_1 \xRightarrow[\mathscr{S}]{s} c_2$$

where $c_1,\, c_2 \in \mathscr{C}$ and $s \in \mathscr{S}$. The informal meaning of this notation is "the evaluation of a statement s in a configuration c_1 completes with the configuration c_2". As the statement s can have arbitrarily complex structure this semantic style is called "big-step" since the evaluation relation " $\Rightarrow_{\mathscr{S}}$ " immediately delivers us the final configuration c_2 as if the computations were performed in one big step, without observable subdivision to computations of smaller components of s.

The relation " $\Rightarrow_{\mathscr{S}}$ " is defined by the following deductive system (see Fig. 3.4). The system consists of five rules each of which has the following generic form

$$\frac{\mathit{premise} \ldots \mathit{premise}}{\mathit{conclusion}}$$

The informal meaning of a rule is that the conclusion holds under the condition that all the premises hold. As we use this system to define the relation " $\Rightarrow_{\mathscr{S}}$ " the conclusions of the rules always have the form

$$c_1 \xRightarrow[\mathscr{S}]{s} c_2$$

for some c_1, c_2 and s. Sometimes a rule does not contain premises, or there is no premise which contains " $\Rightarrow_{\mathscr{S}}$ "; such rules are called *axioms*. Finally, we mark each rule with a label on the right for reference; these labels are not a part of the deductive system and play role of comments.

We now give detailed comments for each rule to explain the whole idea of using a deductive system to specify semantics in whole, and big-step operational semantics in particular.

The first rule, Skip, defines the semantics of the **skip** statement. It is an axiom which tells us that the evaluation of **skip** statement does not change the configuration.

The next one, Assign, deals with assignments. It is also an axiom, which tells us that an assignment never changes a world (notice the second component of configuration, which is left unchanged). As for the state component, first, we evaluate the value of expression e in given state σ using the semantics for expressions $[\![\bullet]\!]_{\mathscr{E}}$, defined in the previous section. Then we substitute the value for variable x in the state with calculated value using the primitive $\bullet\,[\bullet \leftarrow \bullet]$ which has the following definition:

$$\sigma\,[x \leftarrow v]\,y = \begin{cases} \sigma\,y & , \quad y \neq x \\ v & , \quad y = x \end{cases}$$

Thus, for a state σ, variable x, and value v the state $\sigma\,[x \leftarrow v]$ assigns v to x and leaves other variables unchanged.

Two next rules, READ and WRITE, describe the semantics for **read** and **write** constructs. In both cases the definitions use corresponding primitives for worlds: in READ we extract the next value z (if any) from the input stream and return the modified state, in which the variable being read is associated with z, and remaining world. In WRITE we first calculate the value of the expression being written in current state and put it into the output stream of the word. Note, both rules are axioms as well: although READ has a premise, this premise does not contain "$\underset{\mathscr{S}}{\Rightarrow}$".

Finally, the last rule SEQ prescribes the semantics for the sequential composition. This time it is not an axiom, and it tells us that in order to evaluate the composition of two statements we first need to evaluate the first one, obtaining some intermediate configuration c', and then the second one, using this intermediate configuration as input. Note, the order of evaluation is defined not by the order of premises, but by their nature: regardless the order in which the premises are given it is impossible to calculate c_2 unless c' is calculated first.

With the relation "$\underset{\mathscr{S}}{\Rightarrow}$" defined we can abbreviate the "surface" semantics for the language of statements:

$$\frac{\langle \Lambda,\, \langle i,\, \epsilon\rangle\rangle \overset{s}{\underset{\mathscr{S}}{\Longrightarrow}} \langle \sigma,\, \omega\rangle}{[\![s]\!]_{\mathscr{S}}\, i = \mathbf{out}\,\omega} \qquad (\star)$$

This rule (which is *not* a part of big-step operational semantics) establishes a connection between evaluation relation "$\underset{\mathscr{S}}{\Rightarrow}$" and the semantics for statements $[\![\bullet]\!]_{\mathscr{S}}$. In order to calculate the output stream for given input sream i we first construct an initial configuration

$$\langle \Lambda,\, \langle i,\, \epsilon\rangle\rangle$$

(remember, Λ stands for the empty state), then calculate the final configuration $\langle \sigma,\, \omega\rangle$ using the evaluation relation, and then extract the output stream of the final world.

Similarly to the denotational case, we can formulate two properties of given semantics:

- *Determinism*: given arbitrary $c \in \mathscr{C}$ and $s \in \mathscr{S}$ there exists at most one $c' \in \mathscr{C}$ such that

$$c \overset{s}{\underset{\mathscr{S}}{\Longrightarrow}} c'$$

Indeed, we can see that for each kind of statement and for each initial configurations there is at most one applicable rule.

- *Compositionality*: for each non-axiom rule (this time, only SEQ) in all its premises only proper subconstructs of the conclusion construct are used (this time, s_1 and s_2 of $s_1\,;\,s_2$). Hence, the principle of structural induction can be used to prove the properties of the semantics.

We now show by example how big-step operational semantics works. Let us have the following program

$$\textbf{read}\ (\mathtt{x});\ \textbf{read}\ (\mathtt{y});\ \mathtt{z} := \mathtt{x} + \mathtt{y};\ \textbf{write}\ (\mathtt{z})$$

and let the input stream be $\langle 2,\,3\rangle$. First, according to $(\star)$, we construct an initial configuration and write down what we currently know about the evaluation relation:

$$\langle \Lambda,\, \langle \langle 2,\,3\rangle\,,\,\epsilon\rangle\rangle \xRightarrow[\mathscr{S}]{\textbf{read}\ (\mathtt{x});\ \textbf{read}\ (\mathtt{y});\ \mathtt{z} := \mathtt{x} + \mathtt{y};\ \textbf{write}\ (\mathtt{z})} \boxed{?}$$

We do not know yet what to put instead of $\boxed{?}$. To figure it out we need to seek for applicable rule. It turns out that the only one rule can be applied, namely SEQ (remember, ";" associates to the right). So, we can move one floor up by applying the rule and filling in the parts we already know so far:

$$\frac{\langle \Lambda,\, \langle \langle 2,\,3\rangle\,,\,\epsilon\rangle\rangle \xRightarrow[\mathscr{S}]{\textbf{read}\ (\mathtt{x})} \boxed{??} \qquad \boxed{??} \xRightarrow[\mathscr{S}]{\textbf{read}\ (\mathtt{y});\ \mathtt{z} := \mathtt{x} + \mathtt{y};\ \textbf{write}\ (\mathtt{z})} \boxed{?}}{\langle \Lambda,\, \langle \langle 2,\,3\rangle\,,\,\epsilon\rangle\rangle \xRightarrow[\mathscr{S}]{\textbf{read}\ (\mathtt{x});\ \textbf{read}\ (\mathtt{y});\ \mathtt{z} := \mathtt{x} + \mathtt{y};\ \textbf{write}\ (\mathtt{z})} \boxed{?}}$$

In addition to unknown configuration "$\boxed{?}$" we now have another one, "$\boxed{??}$". But, now we can apply rule READ to the first premise, which immediately lets us calculate what "$\boxed{??}$" is:

$$\frac{\dfrac{\langle 2,\,\langle\langle 3\rangle\,,\,\epsilon\rangle\rangle = \textbf{read}\,\langle\langle 2,\,3\rangle\,,\,\epsilon\rangle}{\langle \Lambda,\, \langle \langle 2,\,3\rangle\,,\,\epsilon\rangle\rangle \xRightarrow[\mathscr{S}]{\textbf{read}\ (\mathtt{x})} \langle [\mathtt{x}\mapsto 2],\, \langle\langle 3\rangle\,,\,\epsilon\rangle\rangle} \quad \ldots}{\langle \Lambda,\, \langle \langle 2,\,3\rangle\,,\,\epsilon\rangle\rangle \xRightarrow[\mathscr{S}]{\textbf{read}\ (\mathtt{x});\ \textbf{read}\ (\mathtt{y});\ \mathtt{z} := \mathtt{x} + \mathtt{y};\ \textbf{write}\ (\mathtt{z})} \boxed{?}}$$

Now we can proceed with the second premise:

$$\langle [\mathtt{x}\mapsto 2],\, \langle\langle 3\rangle\,,\,\epsilon\rangle\rangle \xRightarrow[\mathscr{S}]{\textbf{read}\ (\mathtt{y});\ \mathtt{z} := \mathtt{x} + \mathtt{y};\ \textbf{write}\ (\mathtt{z})} \boxed{?}$$

Again, we can move one floor up by using the rule SEQ (and only it):

$$\frac{\langle [\mathtt{x} \mapsto 2],\ \langle \langle 3 \rangle,\ \epsilon \rangle \rangle \xRightarrow[\mathscr{S}]{\textbf{read}\ (\mathtt{y})}\ \boxed{???}\quad \boxed{???} \xRightarrow[\mathscr{S}]{\mathtt{z}\ :=\ \mathtt{x}\ +\ \mathtt{y};\ \textbf{write}\ (\mathtt{z})}\ \boxed{?}}{\langle [\mathtt{x} \mapsto 2],\ \langle \langle 3 \rangle,\ \epsilon \rangle \rangle \xRightarrow[\mathscr{S}]{\textbf{read}\ (\mathtt{y});\ \mathtt{z}\ :=\ \mathtt{x}\ +\ \mathtt{y};\ \textbf{write}\ (\mathtt{z})}\ \boxed{?}}$$

And, again, we can apply the rule READ to the first premise, which gives us the value for "$\boxed{???}$":

$$\frac{\dfrac{\langle 3,\ \langle \epsilon,\ \epsilon \rangle \rangle = \textbf{read}\ \langle \langle 3 \rangle,\ \epsilon \rangle}{\langle [\mathtt{x} \mapsto 2],\ \langle \langle 3 \rangle,\ \epsilon \rangle \rangle \xRightarrow[\mathscr{S}]{\textbf{read}\ (\mathtt{y})}\ \langle [\mathtt{x} \mapsto 2,\ \mathtt{y} \mapsto 3],\ \langle \epsilon,\ \epsilon \rangle \rangle}\quad \ldots}{\langle [\mathtt{x} \mapsto 2],\ \langle \langle 3 \rangle,\ \epsilon \rangle \rangle \xRightarrow[\mathscr{S}]{\textbf{read}\ (\mathtt{y});\ \mathtt{z}\ :=\ \mathtt{x}\ +\ \mathtt{y};\ \textbf{write}\ (\mathtt{z})}\ \boxed{?}}$$

Moving to the second premise and applyng the rule SEQ gives us

$$\frac{\langle [\mathtt{x} \mapsto 2,\ \mathtt{y} \mapsto 3],\ \langle \epsilon,\ \epsilon \rangle \rangle \xRightarrow[\mathscr{S}]{\mathtt{z}\ :=\ \mathtt{x}\ +\ \mathtt{y}}\ \boxed{????}\quad \boxed{????} \xRightarrow[\mathscr{S}]{\textbf{write}\ (\mathtt{z})}\ \boxed{?}}{\langle [\mathtt{x} \mapsto 2,\ \mathtt{y} \mapsto 3],\ \langle \epsilon,\ \epsilon \rangle \rangle \xRightarrow[\mathscr{S}]{\mathtt{z}\ :=\ \mathtt{x}\ +\ \mathtt{y};\ \textbf{write}\ (\mathtt{z})}\ \boxed{?}}$$

which makes it possible to apply the rule ASSIGN (we left the reader to confirm that $[\![\mathtt{x}{+}\mathtt{y}]\!]_{\mathscr{E}}\,[\mathtt{x} \mapsto 2,\ \mathtt{y} \mapsto 3] = 5$):

$$\frac{\langle [\mathtt{x} \mapsto 2,\ \mathtt{y} \mapsto 3],\ \langle \epsilon,\ \epsilon \rangle \rangle \xRightarrow[\mathscr{S}]{\mathtt{z}\ :=\ \mathtt{x}\ +\ \mathtt{y}}\ \langle [\mathtt{x} \mapsto 2,\ \mathtt{y} \mapsto 3,\ \mathtt{z} \mapsto 5],\ \langle \epsilon,\ \epsilon \rangle \rangle\quad \ldots}{\langle [\mathtt{x} \mapsto 2,\ \mathtt{y} \mapsto 3],\ \langle \epsilon,\ \epsilon \rangle \rangle \xRightarrow[\mathscr{S}]{\mathtt{z}\ :=\ \mathtt{x}\ +\ \mathtt{y};\ \textbf{write}\ (\mathtt{z})}\ \boxed{?}}$$

This gives us the value for "$\boxed{????}$", which allows us to finally calculate "$\boxed{?}$" using the axiom WRITE:

$$\langle [\mathtt{x} \mapsto 2,\ \mathtt{y} \mapsto 3,\ \mathtt{z} \mapsto 5],\ \langle \epsilon,\ \epsilon \rangle \rangle \xRightarrow[\mathscr{S}]{\textbf{write}\ (\mathtt{z})}\ \langle [\mathtt{x} \mapsto 2,\ \mathtt{y} \mapsto 3,\ \mathtt{z} \mapsto 5],\ \langle \epsilon,\ \langle 5 \rangle \rangle \rangle$$

Thus, taking into account ($\star$) yet again, we have

$$[\![\textbf{read}\ (\mathtt{x});\ \textbf{read}\ (\mathtt{y});\ \mathtt{z}\ :=\ \mathtt{x}\ +\ \mathtt{y};\ \textbf{write}\ (\mathtt{z})]\!]_{\mathscr{S}}\ \langle 2, 3 \rangle = 5$$

We can see that big-step operational semantics allows us to perform program evaluation for concrete input: we just form an initial configuration for

this input and then systematically apply rules of the semantics until (and if) we come to the result of the evaluation. The application of an axiom completes in one step (we can immediately evaluate the result configuration); the application of the rule for sequential composition amounts to splitting the composite statement into two, moving one floor up in the semantics's rule, and repeat. In the process a certain structure, called *derivation tree*, is maintained implicitly. The nodes of a derivation tree are *instances* of the semantics' rules, the edges connect premises with conclusions. The derivation tree for the example we just considered is depictured in Fig. 3.5 (the program's statements are omitted due to space considerations). Additionally to the derivation tree itself a configuration calculation flow is shown explicitly by dashed arrows. We can see, that for axioms these arrows come directly from left- to right-hand part of the rules, while the rule for composition threads configurations from right-hard part of one rule to the left-hand part of another.

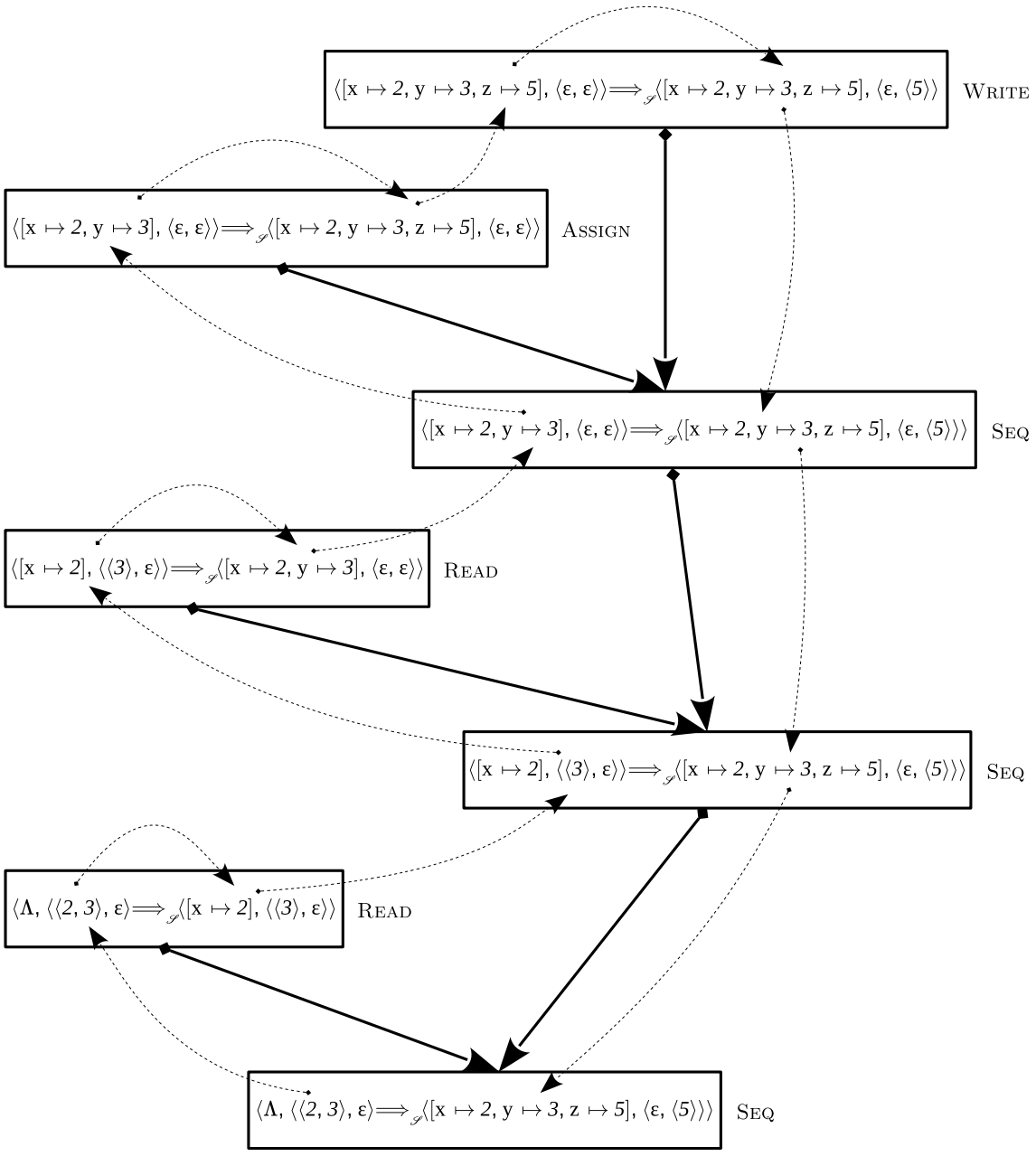


Figure 3.5: Derivation tree example

Besides evaluation big-step operational semantics can also be used to *prove* the properties of the programs. For example, we can prove, that for arbitrary $n, m \in \mathbb{N}$

$$[\![\textbf{read}\ (\mathtt{x});\ \textbf{read}\ (\mathtt{y});\ \mathtt{z} := \mathtt{x} + \mathtt{y};\ \textbf{write}\ (\mathtt{z})]\!]_{\mathscr{S}}\ \langle n, m\rangle = n + m$$

Indeed, we just need to repeat the construction of the derivation tree as we did in the evluation case, but this time instead of concrete numbers use their abstract denotations n and m; all steps could be performed as before until we arrive at the assignment rule. This time we would need to prove an additional lemma

$$[\![\mathtt{x} + \mathtt{y}]\!]_{\mathscr{E}}\,[\mathtt{x} \mapsto n, \mathtt{y} \mapsto m] = n + m$$

which, of course, can be easily done by unfolding corresponding rule for $[\![\bullet]\!]_{\mathscr{E}}$.

And, besides proving the properties of *concrete programs*, big-step operational semantics can be used to prove the properties of the semantics of the *language* as whole. We consider some examples in the next section.

3.2.2 Properties of the Semantics

Now we formulate some simple properties of the semantics for straight-line programs; as the language is rather simple, these properties and their proofs might look obvious. We nevertheless do this as a set of warming-up exersices to demonstrate relevant techniques before dealing with more advanced languages with less trivial properties.

Associativity of Composition

First we consider rather an expected property of sequential composition to be *associative*.

Lemma (Associativity of composition)**.** *For arbitrary s_1, s_2, s_3 and arbitrary c_1, c_2*

$$c_1 \xRightarrow[\mathscr{S}]{s_1;\ (s_2; s_3)} c_2 \iff c_1 \xRightarrow[\mathscr{S}]{(s_1; s_2); s_3} c_2$$

In other words, the grouping of statements inside compositions is not essential, only their order.

Proof. Proving this claim amounts to proving it in both directions; we only do it from left to right since the opposite can be done similarly. So, we assume

$$c_1 \xRightarrow[\mathscr{S}]{s_1;\ (s_2; s_3)} c_2 \qquad (\star)$$

From this it is immedialety follows that

$$\frac{c_1 \xRightarrow[\mathscr{S}]{s_1} c' \quad c' \xRightarrow[\mathscr{S}]{s_2; s_3} c_2}{c_1 \xRightarrow[\mathscr{S}]{s_1;\ (s_2; s_3)} c_2}$$

because there is no way to complete the derivation tree for ($\star$) other than by applying the rules of the semantics (this time, Seq). Similarly, we can apply the same rule for the second premise, which gives us

$$\frac{c_1 \xRightarrow[\mathscr{S}]{s_1} c' \qquad \dfrac{c' \xRightarrow[\mathscr{S}]{s_2} c'' \quad c'' \xRightarrow[\mathscr{S}]{s_3} c_2}{c' \xRightarrow[\mathscr{S}]{s_2; s_3} c_2}}{c_1 \xRightarrow[\mathscr{S}]{s_1;\ (s_2; s_3)} c_2}$$

Thus, we may conclude, that

$$\begin{gathered} c_1 \xRightarrow[\mathscr{S}]{s_1} c' \\ c' \xRightarrow[\mathscr{S}]{s_2} c'' \\ c'' \xRightarrow[\mathscr{S}]{s_3} c_2 \end{gathered}$$

for some configurations c' and c''. Now we turn our attention to the following evaluation relation:

$$c_1 \xRightarrow[\mathscr{S}]{(s_1; s_2); s_3} \boxed{?}$$

We have to prove that $\boxed{?} = c_2$; again, applying rule Seq twice we arrive at (a partial) derivation tree

$$\frac{\dfrac{c_1 \xRightarrow[\mathscr{S}]{s_1} \boxed{???} \quad \boxed{???} \xRightarrow[\mathscr{S}]{s_2} \boxed{??}}{c_1 \xRightarrow[\mathscr{S}]{s_1; s_2} \boxed{??}} \qquad \boxed{??} \xRightarrow[\mathscr{S}]{s_3} \boxed{?}}{c_1 \xRightarrow[\mathscr{S}]{(s_1; s_2); s_3} \boxed{?}}$$

By the determinism of the semantics we can conclude that $\boxed{???} = c'$, from which (again, by determinism) immediately follows that $\boxed{??} = c''$ and, finally, that $\boxed{?} = c_2$. □

Note, we did not use anything besides the rule Seq, thus the associativity of composition does not depend on other rules and will hold as long as the rule Seq is kept in its current form.

From what we've just proven it is immediately follows that for arbitrary s_1, s_2, s_3

$$[\![s_1;\ (s_2; s_3)]\!]_{\mathscr{S}} = [\![(s_1; s_2);\ s_3]\!]_{\mathscr{S}}$$

In other words, regrouping the components of composition is an equivalent transformation.

Finite Read and Write

The next property we are going to establish is that every statement can read only a finite number of input stream elements; this number depends only on the statement itself, and for all initial configurations with an input stream shorter than that number the evaluation never comes to a final configuration. Formally, we prove the following lemma:

Lemma (Finite read)**.** *For any statement s there is a natural number n such that for every configuration $c = \langle \sigma, \langle i, o \rangle \rangle$*

1. *if $|i| < n$ then there is no configuration c' such that $c \xRightarrow[\mathscr{S}]{s} c'$;*
2. *if $|i| \geq n$ and there is some configuration $c' = \langle \sigma', \langle i', o' \rangle \rangle$ such that $c \xRightarrow[\mathscr{S}]{s} c'$, then $|i'| = |i| - n$.*

Proof. The proof goes by structural induction. For the base case we need to consider four simple statements and four corresponding semantics rules. By inspecting the rules Skip, Assign, and Write we can conclude that neither of them touches input stream. Thus, if an evaluation according to these rules succesfully completes, then input streams of initial and final configurations are equal. This means that for these three statements $n = 0$.

On the contrary, the rule Read takes one element from input stream and puts it into a state. In the final configuration input stream will be one element shorter, and this configuration can not be constructed if the initial one had empty input stream. Thus, in this case $n = 1$.

To complete the proof, we need to perform an induction step by inspecting the rule Seq. Let us have a statement $s_1\,; s_2$; by induction hypothesis we may conclude, that there are numbers n_1 and n_2 such the claim is true for s_1 and s_2, respectively. Let first assume, that for some configuration $c = \langle \sigma, \langle i, o \rangle \rangle$ there is another one $c' = \langle \sigma', \langle i', o' \rangle \rangle$ such that

$$c \xRightarrow[\mathscr{S}]{s_1\,; s_2} c'$$

By unfolding the rule Seq we have

$$\frac{c \xRightarrow[\mathscr{S}]{s_1} \langle \sigma'', \langle i'', o'' \rangle \rangle \quad \langle \sigma'', \langle i'', o'' \rangle \rangle \xRightarrow[\mathscr{S}]{s_2} c'}{c \xRightarrow[\mathscr{S}]{s_1\,; s_2} c'}$$

From induction hypothesis for s_1 we know, that $|i''| = |i| - n_1$; from induction hypothesis for s_2 we know, that $|i'| = |i''| - n_2$; thus, overall $|i'| = |i| - n_1 - n_2$, or $|i'| = |i| - (n_1 + n_2)$ and $n = n_1 + n_2$.

Let us now assume that $|i| < n_1 + n_2$. If $|i| < n_1$, then, by the induction hypothesis for s_1, there is no configuration c'' such that

$$\langle \sigma, \langle i, o \rangle \rangle \xRightarrow[\mathscr{S}]{s_1} c''$$

If $n_1 \leq |i| < n_1 + n_2$ and there is a configuration $c'' = \langle \sigma'', \langle i'', o'' \rangle \rangle$ such that

$$\langle \sigma, \langle i, o \rangle \rangle \xRightarrow[\mathscr{S}]{s_1} c''$$

then $|i''| < n_2$ by induction hypothesis for s_1; by induction hypothesis for s_2 there is no configuration c' such that

$$c'' \xRightarrow[\mathscr{S}]{s_2} c'$$

which completes the proof. □

A similar property can be established for output streams.

Lemma (Finite write)**.** *Let s be a statement. Then there is a natural number n such that for each pairs of configurations $c = \langle \sigma, \langle i, o \rangle \rangle$ and $c' = \langle \sigma', \langle i', o' \rangle \rangle$ if*

$$c \xRightarrow[\mathscr{S}]{s} c'$$

then $|o'| = |o| + n$.

Proof. Like in the previous lemma, the principle of structural induction gives us the blueprint for the proof. In the base case, we consider four simple statements and corresponding semantics rules. For all cases, except for WRITE, the output stream is left untouched, thus $n = 0$. For WRITE, if the statement managed to succeed, the output stream in the final configuration is one element longer, than in the initial one, thus $n = 1$.

In the induction step case, we have a statement $s_1 ; s_2$ and we assume that lemma holds for s_1 and s_2; this gives us the numbers n_1 and n_2. Next we assume that

$$c \xRightarrow[\mathscr{S}]{s_1 ; s_2} c'$$

and unfold the rule SEQ:

$$\frac{c \xRightarrow[\mathscr{S}]{s_1} \langle \sigma'', \langle i'', o'' \rangle \rangle \quad \langle \sigma'', \langle i'', o'' \rangle \rangle \xRightarrow[\mathscr{S}]{s_2} c'}{c \xRightarrow[\mathscr{S}]{s_1 ; s_2} c'}$$

By induction hypothesis for s_1 we know that $|o''| = |o| + n_1$; by induction hypothesis for s_2 — that $|o'| = |o''| + n_2$. Thus, $|o'| = |o| + (n_1 + n_2)$, from which we conclude $n = n_1 + n_2$. □

We can see the similarity between these two proofs; to tell the truth this is the case for all claims that can be proven by structural induction. Thus, almost every time when we encounter such a claim we will skip the proof.

From these two lemmas we can derive the following corollaries:

1. For arbitrary $s \in \mathscr{S}, i, i' \in \mathbb{Z}^*$ if both $[\![s]\!]_{\mathscr{S}}\, i$ and $[\![s]\!]_{\mathscr{S}}\, i'$ defined, then

$$|[\![s]\!]_{\mathscr{S}}\, i| = |[\![s]\!]_{\mathscr{S}}\, i'|$$

   In other words, the length of the result depends only on the statement, not input data.

2. Let $[\![s]\!]_{\mathscr{S}}\, i$ is defined for some $s \in \mathscr{S}$ and $i \in \mathbb{Z}^*$. Then for arbitrary $i' \in \mathbb{Z}^*$

$$[\![s]\!]_{\mathscr{S}}\, ii' = [\![s]\!]_{\mathscr{S}}\, i$$

   In other words, if a program has returned some result for some input, it will also return the same result if we arbitrarily extend this input. The proof uses the determinism property of the semantics.

Variable Renaming

The last example of a semantic property we are going to consider is *equivalence up to variable renaming*.

By observing the semantics of expressions and statements we may notice that to some extent all variables are indistinguishable: no rule requires any *specific* variable to be used, any variable can be potentially used anywhere. Thus, if we, for example, have an assignment

$\mathtt{a} := E$

we can rename "a" in, say, "b", and nothing changes as long as we simultaneously rename all following occurrencies of "a" to "b" as well. We have, however, to be careful to avoid *coalescing* different variables into one; for example, if "b" already used in the same program, the result of renaming may or may not be correct depending on the properties of this concrete program. Thus, we should only consider *one-to-one* variable renamings.

More formally, we call a function

$$\theta : \mathscr{X} \to \mathscr{X}$$

a *renaming* iff the following two conditions hold:

1. θ is total (defined for each $x \in \mathscr{X}$);

2. for each $y \in \mathscr{X}$ there is a unique $x \in \mathscr{X}$ such that $\theta(x) = y$.

This is in fact a mathematical definition of a *bijective* mapping from $\mathscr{X}$ to $\mathscr{X}$. It is immediately follows from the definition that the inverse mapping θ^{-1} exists, which is also a bijection. Moreover, both compositions $\theta\theta^{-1}$ and $\theta^{-1}\theta$ are identity functions.

We now define what is a result of application of a renaming θ to an expression e (notation: $e\,\theta$):

$$\begin{array}{rcll} x\,\theta & = & \theta(x) & , \quad x \in \mathscr{X} \\ n\,\theta & = & n & , \quad n \in \mathbb{N} \\ (l \oplus r)\,\theta & = & (l\,\theta) \oplus (r\,\theta) & \end{array}$$

In a nutshell, $e\,\theta$ is an expression in which all variables are substituted to their images according to θ. Similarly, we can extend the application of renaming to statements:

$$\begin{array}{rcl} \textbf{skip}\ \theta & = & \textbf{skip} \\ (x{:=}e)\,\theta & = & (\theta(x)){:=}(e\,\theta) \\ (\textbf{read}\ \ (x))\,\theta & = & \textbf{read}\ \ (\theta(x)) \\ (\textbf{write}\ \ (e))\,\theta & = & \textbf{write}\ \ (e\,\theta) \\ (s_1\,;s_2)\,\theta & = & (s_1\,\theta)\,;(s_2\,\theta) \end{array}$$

Lemma. *Let s be a statement and θ be a renaming. Then*

$$s \equiv s\,\theta$$

Proof. First, we recollect that

$$s \equiv s\,\theta$$

means precisely

$$[\![s]\!]_{\mathscr{S}} = [\![s\,\theta]\!]_{\mathscr{S}}$$

Since $[\![\bullet]\!]_{\mathscr{S}}$ is defined in terms of evaluation relation " $\Rightarrow_{\mathscr{S}}$ " we need to prove a similar claim for " $\Rightarrow_{\mathscr{S}}$ ". This claim on the first glance may look like the follows: for all c, $c' \in \mathscr{C}$

$$c \overset{s}{\Longrightarrow}_{\mathscr{S}} c' \Longleftrightarrow c \overset{s\,\theta}{\Longrightarrow}_{\mathscr{S}} c'$$

Alas, this does not hold in general case: let us have a statement **write** (x) and let $\theta(x) = y$. Then for a configuration $c = \langle [x \mapsto 3], \langle i, o \rangle \rangle$ there is no c' such that

$$c \overset{\textbf{write}\ \ (y)}{\Longrightarrow}_{\mathscr{S}} c'$$

since y is undefined in $[x \mapsto 3]$. The problem here is that renamed statements only behave similarly to the original ones in *renamed* states.

The fixed claim looks like the follows: let $c = \langle \sigma, w \rangle$ and $c' = \langle \sigma', w' \rangle$ then

$$c \xRightarrow[\mathscr{S}]{s} c' \Longleftrightarrow \langle \sigma \circ \theta^{-1}, w \rangle \xRightarrow[\mathscr{S}]{s\,\theta} \langle \sigma' \circ \theta^{-1}, w' \rangle \qquad (\clubsuit)$$

Informally speaking here we "repair" states for renamed statements by first applying the *inverse* renaming θ^{-1} which exists by bijectivity of θ. To prove this we need additional claim.

Claim ($\spadesuit$)**.** *Let e be an expression, σ be a state, and n be an integer number. Then*

$$[\![e]\!]_{\mathscr{E}}\,\sigma = n \Longleftrightarrow [\![e\,\theta]\!]_{\mathscr{E}}\,(\sigma \circ \theta^{-1}) = n$$

Proof. One may expect that the proof has to be performed in two directions. However, by the bijectivity of θ

$$\begin{array}{rcl} \theta & = & (\theta^{-1})^{-1} \\ (e\,\theta)\,\theta^{-1} & = & e \\ \sigma & = & (\sigma \circ \theta^{-1}) \circ \theta \end{array}$$

Thus, if we prove the claim from left to right, then (by arbitrariness of e, σ, and θ) we can assume

$$\begin{array}{rcl} e & = & e\,\theta \\ \sigma & = & \sigma \circ \theta^{-1} \end{array}$$

and the right-to-left direction will follow from left-to-right immediately.

The proof proceeds by structural induction.

The base case is a constant or a variable. Since the semantics of constants does not depend on a state the claim for constant case follows vacuously. If e is a variable x, then:

$$\begin{array}{rcl} [\![x\,\theta]\!]_{\mathscr{E}}\,(\sigma \circ \theta^{-1}) & = & \text{(by the definition of } [\![\bullet]\!]_{\mathscr{E}}) \\ (\sigma \circ \theta^{-1})\,(x\,\theta) & = & \text{(by the definition of renaming application)} \\ (\sigma \circ \theta^{-1})\,(\theta\,(x)) & = & \text{(by the definition of function composition)} \\ ((\sigma \circ \theta^{-1}) \circ \theta)\,(x) & = & \text{(by the associativity of function composition)} \\ (\sigma \circ (\theta^{-1} \circ \theta))\,(x) & = & \text{(by the definition of function inversion)} \\ \sigma\,(x) & = & \text{(by the condition of the claim)} \\ n & & \end{array}$$

The induction step assumes that the claim holds for some expressions l and r; we need to prove that it also holds for $l \oplus r$:

$$
\begin{array}{rcl}
[\![(l \oplus r)\,\theta]\!]\,(\sigma \circ \theta^{-1}) & = & \text{(by the definition of renaming application)} \\
[\![(l\,\theta) \oplus (r\,\theta)]\!]\,(\sigma \circ \theta^{-1}) & = & \text{(by the definition of } [\![\bullet]\!]_{\mathscr{E}}\text{)} \\
[\![l\,\theta]\!]\,(\sigma \circ \theta^{-1}) \otimes [\![r\,\theta]\!]\,(\sigma \circ \theta^{-1}) & = & \text{(by inductive hypotheses)} \\
[\![l]\!]\,\sigma \otimes [\![r]\!]\,\sigma & = & \text{(by the condition of the claim)} \\
n & &
\end{array}
$$

This completes the proof. □

The proof of (♣) goes by structural induction as well. Similarly to the claim (♠) we only need to prove it in one direction (let it be left-to-right).

In the base case we have to consider four types of simple statements:

1. As **skip** does not change a configuration the lemma holds vacuously.

2. Let us have an assignment $x{:=}e$. First, by the definition of renaming application

$$(x{:=}e)\,\theta = (\theta\,(x)){:=}(e\,\theta)$$

Then, by the definition of "$\Rightarrow_{\mathscr{S}}$":

$$\langle \sigma \circ \theta^{-1},\, w\rangle \xrightarrow{(\theta\,(x)):=(e\,\theta)}_{\mathscr{S}} \langle (\sigma \circ \theta^{-1})[\theta\,(x) \leftarrow [\![e\,\theta]\!]_{\mathscr{E}}\,(\sigma \circ \theta^{-1})],\, w\rangle$$

By (♠) we know that

$$[\![e\,\theta]\!]_{\mathscr{E}}\,(\sigma \circ \theta^{-1}) = [\![e]\!]_{\mathscr{E}}\,\sigma$$

By the definition of substitution in a state

$$(\sigma \circ \theta^{-1})[\theta\,(x) \leftarrow [\![e]\!]_{\mathscr{E}}\,\sigma]\,y = \begin{cases} [\![e]\!]_{\mathscr{E}}\,\sigma & , \quad y = \theta\,(x) \\ (\sigma \circ \theta^{-1})\,y & , \quad y \neq \theta\,(x) \end{cases}$$

By the properties of functional inversion and composition and, again, using the definition of substitution in a state we have

$$\begin{cases} [\![e]\!]_{\mathscr{E}}\,\sigma & , \quad y = \theta\,(x) \\ (\sigma \circ \theta^{-1})\,y & , \quad y \neq \theta\,(x) \end{cases} = \begin{cases} [\![e]\!]_{\mathscr{E}}\,\sigma & , \quad \theta^{-1}\,(y) = x \\ (\sigma\,(\theta^{-1}\,(y)) & , \quad \theta^{-1}\,(y) \neq x \end{cases} = (\sigma[x \leftarrow [\![e]\!]_{\mathscr{E}}])\,(\theta^{-1}\,(y))$$

Thus, $\sigma' = \sigma[x \leftarrow [\![e]\!]_{\mathscr{E}}] \circ \theta^{-1}$ which completes the proof for the assignment case.

3. Two remaining cases can be proven similarly using the observation that worlds are invariant under renaming.

Finally, to prove the induction step we assume that (♣) holds for statements s_1 and s_2 and we need to prove

$$\langle \sigma \circ \theta^{-1},\, w \rangle \xrightarrow[\mathscr{S}]{s_1\,;\,s_2} \langle \sigma' \circ \theta^{-1},\, w' \rangle$$

From the condition of the lemma we know

$$\frac{\langle \sigma,\, w \rangle \xrightarrow[\mathscr{S}]{s_1} \langle \sigma'',\, w'' \rangle \quad \langle \sigma'',\, w'' \rangle \xrightarrow[\mathscr{S}]{s_2} \langle \sigma',\, w' \rangle}{\langle \sigma,\, w \rangle \xrightarrow[\mathscr{S}]{s_1\,;\,s_2} \langle \sigma',\, w' \rangle}$$

for some σ'' and w''. Applying the induction hypotheses to the premises and then applying rule Seq we acquire the following derivation

$$\frac{\langle \sigma \circ \theta^{-1},\, w \rangle \xrightarrow[\mathscr{S}]{s_1} \langle \sigma'' \circ \theta^{-1},\, w'' \rangle \quad \langle \sigma'' \circ \theta^{-1},\, w'' \rangle \xrightarrow[\mathscr{S}]{s_2} \langle \sigma' \circ \theta^{-1},\, w' \rangle}{\langle \sigma \circ \theta^{-1},\, w \rangle \xrightarrow[\mathscr{S}]{s_1\,;\,s_2} \langle \sigma' \circ \theta^{-1},\, w' \rangle}$$

which proves the induction step.
Now, assume

$$\langle \Lambda,\, \langle i,\, \epsilon \rangle \rangle \xrightarrow[\mathscr{S}]{s} \langle \sigma,\, w \rangle$$

By (♣) we have

$$\langle \Lambda \circ \theta^{-1},\, \langle i,\, \epsilon \rangle \rangle \xrightarrow[\mathscr{S}]{s\,\theta} \langle \sigma \circ \theta^{-1},\, w \rangle$$

But $\Lambda \circ \theta^{-1} = \Lambda$, which completes the proof since now we have

$$[\![s]\!]_{\mathscr{E}}\, i = [\![s\,\theta]\!]_{\mathscr{E}}\, i$$

for arbitrary i by the definition of $[\![\bullet]\!]_{\mathscr{E}}$.

□

Chapter 4

Language Embedding

As we stated before, the first component of our compiler is source-level interpreter. In this chapter we develop such an interpreter, so far for straight-line programs language, and demonstrate, that semantic description formalisms used in the previous section can be seen as a specification of the interpreter at only slightly higher level of abstraction than actual runnable implementation.

The notion of interpreter, in turn, is tightly connected with the concept of *deep embedding* of one language into another. The symmetric concept of *shallow embedding* also exists; while we do not rely on shallow embedding in our development for the sake of completeness we still address it in the last section of this chapter.

4.1 Deep Embedding

Remember, dealing with interpreters involves two languages:

- a *target* language (the language being interpreted), and
- an *implementation* language (the language the interpreter is written in).

Thus, we represent target programs as data structures in implementation language (representation layer) and then evaluate these programs using interpreter (interpretation layer). In other words, we *embed* target language in the implementation one. This kind of embedding is called *deep embedding*. In our case, the case of straight-line programs languages, when we have two syntactic categories — expressions and statements — deep embedding amounts to developing two representations and two interpreters (for expressions and statements respectively).

4.1.1 Representation Layer

The first step of deep embedding is defining the representation of programs. Previously we stipulated that any program in any language has a representation

in the data domain $\mathfrak{D}$; that was rather a *abstract* theoretical requirement. For deep embedding, however, we need to devise a *concrete* representation for the *programs* in one language as *data structures* of another. The syntax of the straight-line programs language so far consists of the following elements:

$$\begin{array}{rcl} \mathscr{X} & : & \text{variables} \\ \otimes & : & \text{binary operators} \\ \mathbb{N} & : & \text{natural numbers} \\ \mathscr{E} & : & \text{expressions} \\ \mathscr{S} & : & \text{statements} \end{array}$$

We need to devise a representation of each of them in the implementation lenguage, i.e. $\mathcal{X}^a\mathcal{M}^a$. It can be done in various ways; the simplest one is as follows:

- variables and binary operators are represented by strings (thus, "x" becomes "`x`" and "$*$" becomes "$*$");
- natural numbers are represented by themselves.

The most natural way to represent abstract syntax trees ($\mathscr{E}$ and $\mathscr{S}$) in $\mathcal{X}^a\mathcal{M}^a$ is by using S-expressions. We can specify the "shape" of S-expressions to represent the categories of expressions and statements in our language as follows (see Fig. 4.1).

$$\begin{array}{rcll} expr & = & \texttt{Var} & (string) \\ & & \texttt{Const} & (int) \\ & & \texttt{Binop} & (string, expr, expr) \\ stmt & = & \texttt{Skip} & \\ & & \texttt{Assn} & (string, expr) \\ & & \texttt{Read} & (string) \\ & & \texttt{Write} & (expr) \\ & & \texttt{Seq} & (stmt, stmt) \end{array}$$

Figure 4.1: Representation of expressions and statements

One may notice that this specification almost literally repeats the AST definitions for corresponding syntactic categories. This is not a coincidence — S-expressions are known to represent such data structures particularly well. This specification can be understood as a constraint which describes a certain invariant all representations of ASTs should comply with. In fact we can even reify this constraint into the following two verification functions:

```
fun exprWF (e) {
  case e of
    Var   (#str)      → true
  | Const (#val)      → true
```

```
    | Binop (#str, l, r)  → exprWF (l) && exprWF (r)
    | _                  → false
    esac
  }

  fun stmtWF (s) {
    case s of
      Skip            → true
    | Assn  (#str, e) → exprWF (e)
    | Read  (#str)    → true
    | Write (e)       → exprWF (e)
    | Seq   (l, r)    → stmtWF (l) && stmtWF (r)
    | _               → false
    esac
  }
```

These functions being appied to an arbitrary $\mathcal{X}^a\mathcal{M}^a$ value return **true** if and only if this value is a *well-formed* AST for expression or statement respectively.

One may notice that our representation does not impose the strongest possible constraints: for example, it does not prohibit to use invalid binary operators like "???" or variables like "..." . While this is generally true we have to stress that the strength of representation constraints is not an objective in itself.

Constraints of this kind are otherwise known as *types*. For languages with *static typing* their compilers provide a compile-time guarantee that type constraints are never violated; for *dynamically-typed* languages type constraints are checked at runtime; finally, for *untyped* languages (including $\mathcal{X}^a\mathcal{M}^a$) neither compiler nor runtime support provide any essential ensurance of this kind. All three approaches have their strong and weak points, and we are not going to lean to either side. We only stress that in the untyped case we still can work with well-structured data, and further we will use the descriptions like those in Fig. 4.1 to specify what kind of values we are dealing with.

With all details of program representation defined we can encode straightline programs as $\mathcal{X}^a\mathcal{M}^a$ data structures; an example of a program and its corresponding representation is given in Fig. 4.2. This in principle already gives us everithing we need; however we may do the embedding a little more convenient to deal with. For this we first *redefine* all standard $\mathcal{X}^a\mathcal{M}^a$ binary operators to work with *syntax trees* instead of numbers:

```
infixl +  at +  (l, r) {Binop ("+",  opnd (l), opnd (r))}
infixl -  at -  (l, r) {Binop ("-",  opnd (l), opnd (r))}
infixl *  at *  (l, r) {Binop ("*",  opnd (l), opnd (r))}
infixl /  at /  (l, r) {Binop ("/",  opnd (l), opnd (r))}
infixl %  at %  (l, r) {Binop ("%",  opnd (l), opnd (r))}
infix  == at == (l, r) {Binop ("==", opnd (l), opnd (r))}
infix  != at != (l, r) {Binop ("!=", opnd (l), opnd (r))}
infix  <  at <  (l, r) {Binop ("<",  opnd (l), opnd (r))}
infix  <= at <= (l, r) {Binop ("<=", opnd (l), opnd (r))}
```

```
read (x);
read (y);
z := x + y;
write (z)
```

```
Seq (
  Read ("x"),
  Seq  (
    Read ("y"),
    Seq  (
      Assn  ("z", Binop ("+", Var ("x"), Var ("y"))),
      Write (Var ("z"))
))))
```

Figure 4.2: An example program and its representation

```
infix  >  at >  (l, r) {Binop (">",  opnd (l), opnd (r))}
infix  >= at >= (l, r) {Binop (">=", opnd (l), opnd (r))}
infixl && at && (l, r) {Binop ("&&", opnd (l), opnd (r))}
infixl !! at !! (l, r) {Binop ("!!", opnd (l), opnd (r))}

fun opnd (x) {
  case x of
    #str  → Var   (x)
  | #val  → Const (x)
  | _     → x
  esac
}
```

Now, any standard binary operator instead of actually performing designated arithmetic operation will construct corresponding expression AST. We assume that the arguments of these redefined operators can be either natural numbers (designating themselves), strings (designating variables) or other expression ASTs. A supplementary function `opnd` is used to consrtruct a well-formed AST for the operands of the first two forms.

Of course as soon as these definitions are introduced the "old" standard operators become inaccessible in the same scope; however this limitation can be easily worked around, which we leave to the reader as an exercise.

Second, we provide similar AST-constructing primitives for statements:

```
fun read (x) {
  Read (x)
}

fun write (e) {
  Write (opnd (e))
}

infix ::= before := (x, e) {
```

```
  Assn (x, opnd (e))
}

infixr >> before ::= (s1, s2) {
  Seq (s1, s2)
}
```

Since neither ";" nor ":=" can be redefined in $\chi^a\mathcal{M}^a$, we devised a slightly difference syntax for assignment ("::=") and sequential composition (">>"). Now in the scope of these definitions we can write straight-line programs in just a little different concrete syntax than that for the actual $\chi^a\mathcal{M}^a$: we need to use "::=" instead of ":=", ">>" instead of ";" and take variables' names in quotes. Thus, the following $\chi^a\mathcal{M}^a$ expression

```
read ("x") >>
read ("y") >>
"z" ::= "x" + "y" >>
write ("z")
```

will be evaluated to exactly the same representation as in Fig. 4.2.

One may argue that in this concrete case we carefully staged all the things in advance in order to make deep embedding looking so nice. While, indeed, the very idea of $\chi^a\mathcal{M}^a$ development originates from the desire to have a language which would allow to easily demonstrate all relevant concepts and techniques in their direct form, the features we used here (S-expressions and user-defined binary operators) are by no means rare or exotic. They can be found in one or another form in many real-world production languages, and if some of them are missing the replacement can easily be found. On the contrary, modern languages are often equipped with specific features and tools (syntax extension mechanisms, preprocessors, hygienic macro systems, etc.) which $\chi^a\mathcal{M}^a$ does not have and which facilitate the development of *domain-specific languages* (DSLs), a valuable technique many software projects can benefit from.

4.1.2 Interpretation Layer

Similarly to the representation layer for the interpretation one we need to provide implementations for all components of lanaguage semantics and all relevant objects and operations. In our case these components are:

St	–	states
$\mathscr{W}$	–	worlds
$\mathscr{C}$	–	configurations
$[\![\bullet]\!]_{\mathscr{E}}$	–	semantics for expressions
$\Rightarrow$	–	big-step evaluation relation
$[\![\bullet]\!]^{\mathscr{S}}_{\mathscr{S}}$	–	semantics for statements

Luckily, corresponding $\chi^a\mathcal{M}^a$ implmentations are rather straightforward.

We will represent states as regular $\lambda^a\mathcal{M}^a$ functions; we also need implementations for empty state Λ and substitution operation $\bullet\,[\bullet \leftarrow \bullet]$:

```
fun emptyState (x) {
  failure ("undefined variable ""%s""\n", x)
}

infix ← before : (st, [x, v]) {
  fun (y) {
    if x = y then v else st (y) fi
  }
}
```

Note, `emptyState` kindly says us it is undefined on each variable instead of crushing. There were no such requirement in semantic description but from a software engineering standpoint it is always better to explicitly indicate a problem instead of going to hell in a solemn silence. For substitution we provide an infix operator; since there are no ternary operators in $\lambda^a\mathcal{M}^a$ we specify a variable-value pair as its second operand; thus instead of $s\,[x \leftarrow v]$ we will write $s \leftarrow [x, v]$.

We will represent worlds as pairs of input/output streams, and streams as lists of integers. The implementation for world primitives is straighforward as well:

```
fun createWorld (input) {
  [input, {}]
}

fun writeWorld (n, [input, output]) {
  [input, n:output]
}

fun readWorld ([n:input, output]) {
  [n, [input, output]]
}

fun getOutput ([_, output]) {
  reverse (output)
}
```

Function `createWorld` makes a fresh world from initial input stream; functions `readWorld`, `writeWorld`, and `getOutput` are implementations of world primitives **read**, **write**, and **out** respectively. Note, in the implementation we hold output stream in the *reverse* order (since it is slightly easier to add elements to the beginning of a list than to the end), and reverse it before extracting.

Now we may proceed with interpreter implementation for expressions. It worth mentioning that denotational semantics approach, which we used for expressions, conventionally rarely used to implement interpreters as it provides a

```
fun evalExpr (expr) {
  case expr of
    Var   (x)         → fun (st) {st (x)}
  | Const (n)         → fun (st) {n}
  | Binop (op, l, r)  → fun (st) {
                          evalOp (op)(
                            evalExpr (l)(st),
                            evalExpr (r)(st))
                        }
  esac
}
```

Figure 4.3: Expression interpreter, top-level function

too abstract, high-level view on the semantics. In a number of cases such an interpreter can not be implemented at all due to fundamental undecidability of what denotational semantics specifies (for example, there can be no refulationally complete interpreter for PROLOG while its denotational semantics, the least Herbrand model, can easily be described). However, in our case of simple arithmetic expressions denotational semantics is so simple that constructing a coherent interpreter is trivial.

The top-level function for expression interpreter, `evalExpr`, is shown in Fig. 4.3. Recall, in denotational semantics we specified a function from states to integer numbers for each possible form of expression, and we discriminated between these forms using different (and mutually-exclusive) equations. In $\lambda^a\mathcal{M}^a$ these equations are encoded almost literally using pattern-matching in the form of case expression. The branches of this case expression correspond to the right-hand sides of the semantic equations:

- If the expression being evaluated is a variable, then we return an integer value the current state `st` associates with this variable; remember, we represent states by functions, so it is sufficient to simply make a call to `st`.

- For a constant expression, we return integer value `n` this expression designates.

- For binary operator we, first, evaluate the values of its left (`l`) and right (`r`) operands using recursive calls to `evalExpr`. These recursive calls precisely correspond to denotational brackets in the right-hand side of relevant equation. Then, we combine these values in a way binary operator sign `op` prescribes. For this we use a supplementary function `evalOp`, which encodes the syntactic-to-semantic correspondence shown in the table on the page 26.

```
val evalOp =
  let ops = {["+"  , infix + ],
             ["-"  , infix - ],
             ["*"  , infix * ],
             ["/"  , infix / ],
             ["%"  , infix % ],
             ["==" , infix ==],
             ["!=" , infix !=],
             ["<"  , infix < ],
             ["<=" , infix <=],
             [">"  , infix > ],
             [">=" , infix >=],
             ["&&" , infix &&],
             ["!!" , infix !!]}
  in
  fun (op) {
    case assoc (ops, op) of
      Some (f) → f
    | _        → failure ("undefined binary operator ""%s""", op)
    esac
  };
```

Figure 4.4: Binary operators interpreter

The implementation of `evalOp` is shown in Fig. 4.4. We first construct a table `ops` which encodes a correspondence between syntactic representations of binary operators and their semantic counterparts[1]; then we construct a function which takes a syntactic representation of a binary operator as a string, searches the table and returns corresponding function (or signals an error if no such function is found).

We now can switch to the implementation of big-step interpreter for statements. Obviously, the main work is to encode the big-step evaluation relation " $\underset{\mathscr{S}}{\Rightarrow}$ ". We implement it as a function `evalStmt` (Fig. 4.5) which takes initial configuration and a statement and returns final configuration (if any). Again, in the specification of big-step operational semantics different rules handle different forms of statements. In implementation, the same work is done by pattern-matching, and different branches of case expression correspond to different rules of the semantics:

- For skip statement, we just return current configuration.
- For assignment, we first evaluate its right-hand side expression `e` in current state, modify current state using binary operator " $\leftarrow$ " and return

[1] Of course this should be done in a *different* scope than that in which we redefined all standard binary operators in order to implement deep embedding.

modified configuration.

- For read and write statements we use corresponding primitives for worlds and return modified configurations.
- Finally, for sequential composition we calculate the composition of evaluator applications.

Finally, the top-level interpreter just calls for `evalStmt`, constructing initial configuration, and returning the output stream from the final one:

```
fun eval (stmt, input) {
  getOutput $ evalStmt ([emptyState, createWorld (input)], stmt)
}
```

And, again, this function literally encodes the rule for $[\![\bullet]\!]_{\mathscr{S}}$ on page 32.

Now with this interpreter we can run straight-line programs. We take as an example a summation program already considered to illustrate how deep embedding works. We can evaluate the result of this program for the input stream "{2, 3}":

```
printf ("%s\n", evalStmt (read ("x") >>
                           read ("y") >>
                           "z" ::= "x" + "y" >>
                           write ("z"),
                          {2, 3}
                 )
)
```

This will print "{5}", of course.

In future we will refrain from giving such detailed comments on constructing interpreters from operational semantics description. We did it here for the first time to demonstrate that semantic rules can be directly encoded in a programmatical implementation. Later we will leave the development of interpreters to the reader, only giving a high-level implementation hints.

4.1.3 Metacircularity and Towers of Interpreters

The interpreter we presented in the previous section implements a subset of $\lambda^a\mathcal{M}^a$ in $\lambda^a\mathcal{M}^a$. When an interpreter of a language is implemented in this language itself it is called a *self*-interpreter, or *meta-circular* interpreter. Metacircularity is another way to specify the semantics of a language: indeed, an implementation of an interpreter delivers the same amount of knowledge (if not more) as operational semantics, and the language itself serves as a metalanguage. On the other hand, the whole idea of expressing the semantics of a language in terms of the semantics of the same language looks vacuous at the first glance: indeed, if we already understand the semantics there is no need to specify it in the form of an interpreter, and if not then how can we implement an interpreter in a language which semantics we do not fully understand? The answer is that we

```
fun evalStmt (c@[s, w], stmt) {
  case stmt of
    Skip           → c
  | Assn  (x, e)   → [s ← [x, evalExpr (e)(s)], w]
  | Read  (x)      → let [n,w] = readWorld (w) in
                       [s ← [x, n], w]
  | Write (e)      → [s, writeWorld (evalExpr (e)(s), w)]
  | Seq   (s1, s2) → evalStmt (evalStmt (c, s1), s2)
  esac
}
```

Figure 4.5: Big-step semantics interpreter

can use some small subset of the language which semantics we understand well enough to implement the interpreter of the full language. Thus the semantics of complicated constructs becomes encoded in terms of simpler ones.

Another interesting concept which arises naturally is *tower of interpreters*. Let us have a self-interpreter, say an interpreter of $\lambda^a\mathcal{M}^a$ implemented in $\lambda^a\mathcal{M}^a$. This interpreter can run arbitrary $\lambda^a\mathcal{M}^a$ programs, including *itself*. This construct can be generalized: we can have an interpreter for a language L_2 written in a language L_1, and interpreter for L_3 written in L_2, etc. Thus we can run L_3-program on L_3-interpreter running as L_2-program on L_2-interpreter running as L_1-program on L_1-interpreter, etc. We may wonder why someone would wish to use such an exotic appliance. The answer is that, first, like in the metacircular case, these languages can have different expressive power and different complexity. We may take some very simple language and implement an interpreter for more complex one, etc. Thus we can eventually come up with a very advanced language, but in a number of steps with moderately increasing complexity which may be desirable from an engineering standpoint. What is more important is that the properties of the semantics and runtime environment of interpreter implementation language to some extend are inherited for the language being interpreted. For example, let's assume that we have a profiler for some language L_1. If we implement an interpreter for L_2 in L_1 we will acquire a profiler for L_2 for free — indeed, we will only need to map the profiling points in the interpreter for L_2 to the points of a program being interpreted.

Tower of interpreters is a nice theoretical construct, an interesting research subject and at the same time a convenient engineering tool.

4.2 Shallow Embedding

Shallow embedding is an alternative way to implement one language "on top" of another. In contrast to deep embedding, when we first construct a representation of a program being embedded and then run it on an interpreter,

in shallow case we represent the constructs of embedded language directly in an executable form. We demonstrate this technique by repeating the job of implementing an embedding of a straight-line programs languages, but now in a shallow form. As we dealing with the same language and the same semantics many ingredients of deep embedding — states, configurations, etc. — will be reused in the shallow case, so we can immediately start from implementing embedding of expressions.

Remember, in the previous case we devised a representation of expressions' ASTs using S-expressions; this step, however, is disallowed in shallow embedding. What we do instead is represent the constructs of an expressions directly as their semantics. In out case the semantics of an expression is a function from states to integers, which means that under shallow embedding each expression is represented by such a function.

Let us have two expressions `l` and `r`; how can we build a shallow representation for, say, `l + r`? Both `l` and `r` are functions, and we have to construct a function which, given a state, returns the sum of `l (st)` and `r (st)`. This idea gives us the following hypothetical implementation of binary addition encoding:

```
infixl + at + (l, r) {
  fun (st) {
    l (st) + r (st)
  }
}
```

There are, however, two technical difficulties. First, the base cases for expressions are natural numbers and variables. It would be great to represent natural numbers as regular integer constants and variables as strings, but we need to represent them as functions. We can deal with this problem in a similar way as in the deep embedding case: in every context we expect an expression we wrap it in a function which tests for these corner cases and "lifts" integer constants and strings into functional domain:

```
fun opnd (e) {
  case e of
    #str → fun (st) {st (e)}
  | #val → fun (_)  {e}
  | _    → e
  esac
}
```

Note, we did exactly the same in the deep embedding case, and for the same reason; the only difference is that now we lift primitive expressions into different domain.

The second problem is that, actually, "+" inside the definition of "+" means itself, so instead of adding two numbers our function is going to loop forever. In order to overcome this problem we have, first, to provide a synonym to the build-in integer addition and, second, to place the redefinition of addition in a nested scope; we have to do this for every binary operator we redefine:

```
val add = infix +,
    sub = infix -,
    mul = infix *,
    ...

(
 infixl + at + (l, r) {
   fun (st) {
     add (opnd (l) (st), opnd (r) (st))
   }
 }

 infixl - at - (l, r) {
   fun (st) {
     sub (opnd (l) (st), opnd (r) (st))
   }
 }

 infixl * at * (l, r) {
   fun (st) {
     mul (opnd (l) (st), opnd (r) (st))
   }
 }
 ...
)
```

This completes the shallow embedding of expressions. Now when we write an expression built from binary operators, strings and constants its evaluation returns a representation of this expression in the form of a function from states to integers. For example, an expression

"y"*("x"+1)

is evaluated into the function

```
fun (st) {
  mul (fun (st) {st ("y")} (st),
       fun (st) {
         add (fun (st) {st ("x")} (st),
              fun (st) {1}        (st)
         )
       } (st)
  )
}
```

which can be simplified (by unfolding relevant definitions) into

```
fun (st) {st ("y") * (st ("x") + 1)}
```

Now we do not need an interpreter to evaluate this expression; we can just call it passing a relevant state as an argument.

Let's repeat this job for statements. Again, all we need is to represent constructs by their semantics, this time a function from configurations to configurations:

```
fun sk1p (conf) {conf}

fun read (x) {
  fun ([s, w]) {
    let [n,w] = readWorld (w) in
    [s ← [x, n], w]
  }
}

fun write (e) {
  fun ([s, w]) {
    [s, writeWorld (w, opnd (e) (s))]
  }
}

infix ::= before := (x, e) {
  fun ([s, w]) {
    [s ← [x, opnd (e)(s)], w]
  }
}

infixr >> before ::= (s1, s2) {
  fun (conf) {
    s2 (s1 (conf))
  }
}
```

Note, we could not encode **skip** statement with eponymous function since "skip" is a reserved word in $\mathcal{X}^a\mathcal{M}^a$, so we used its contrived version "sk1p". Otherwise we preserved the same surface syntax as with deep embedding.

One may notice that this implementation resembles the implementation of an interpreter in deep embedding case. This is, indeed, the case. In fact in shallow embedding we "fuse" representation and interpretation layers together, thus representation inherits the implementation of interpreter.

Thus, shallow embedding makes it possible to reduce interpretation overhead (but not eliminate is completely: we still express the semantics in the terms of states and configurations with precisely the same encoding as in the deep embedding case). On the other hand, shallow embedding makes it harder (but not completely impossible) to implement *transformations* of embedded programs. Deep and shallow embeddings are two sides of the same coin, and in different scenarios both of them can reveal their merits.

Chapter 5

Abstract Stack Machine

The notion of *abstract machine* is a convenient concept in the field of programming languages, compliers and tools. As the name suggests, abstract machine is a hypothetical computational device which fills the gap between a high-level language and an actual hardware. It, therefore, introduces an additional intermediate abstraction level which facilitates the decomposition of many compiler implementation and program analysis tasks. This decomposition makes it possible to separate certain implementation subtasks, solve them and assess the correctness of solutions independently.

An important question is how an abstract machine can be specified and built. Do we need a blueprint, what appliances and equipment should we use for its manufacturing? Fortunately, it turns out that we already have all what we need in our toolbox. For us an abstract machine is just a language, and to specify an abstract machine we need to specify the syntax and semantics of this language. Implementing an abstract machine amounts to implementing its interpreter. In this section we give the syntax and semantics of a $\lambda^a\mathcal{M}^a$-specific abstract stack machine, describe the compiler from straight-line programs language into this abstract machine and prove its correctness. But, first, we briefly survey the variety of existing flavours of abstract machines and discuss the motivation for the basic features of that we chose.

5.1 The Variety of Abstract Machines

Abstract machines share a lot of common features with actual hardware. They are programmable devices, and their programs consist of instructions each of which is capable of performing a rather simple operation. On the other hand, abstract machines often provide a direct way of representing and using *meta-information* specific to a certain lanaguage or a group of languages this abstract machine is devised to support. This may include types, object structures, etc. For example, JVM directly supports such construct as virtual method call, which for a real hardware has to be implemented using the knowledge of actual virtual

$$x * y + 3$$

(a) Expression

```
LD x
LD y
MUL
CONST 3
ADD
```

(b) Stack Machine Code

```
MUL x, y, %1
ADD %1, 3, %2
```

(c) Three-address Code

```
MOV y, %1
MUL x, %1
ADD 3, %1
```

(d) Two-address Code

Figure 5.1: An Expression and Its Abstract Machine Code

methods table layout, etc.

There are different flavours of abstract machines. For now, as we are dealing with rather a simple language of straight-line programs, only one essential feature is important: the representation of *temporaries*.

In our language (and the majority of conventional programming languages) expressions can contain an arbitrary number of operators. However, actual hardware (and the majority of abstract machines) cannot evaluate arbitrarily large expressions "in one step". It evaluates them operator by operator, storing somewhere intermediate results, otherwise called *temporary values*. There are two major disciplines to work with temporaries:

- Using a *stack*. With this discipline all temporary values are stored on a stack specifically disignated for this purpose. Thus, all operators take the operands from the stack and put the result back. This, in particular, means that the majority of instructions do not have explicit operands.

- Using a potentially infinite number of named locations (often called *pseudo-registers*). Under this approach the operands of each instruction (if any) are explicitly annotated. There are two commonly used special cases: *three-address code*, when an instruction can have up to three distinct operands (two for arguments and one for the result) and *two-address code*, when an instruction can have no more than two operands, one of which serves simultaneously as an argument and as the result.

All these versions of abstract machines have their merits and shortcomings, and rather easier to switch to and from. Stack code is known to be very compact (indeed, it does not require extra space to specify the operands for the majority of instructions); it is also a little bit easier to generate. As the same time code with explicit operands is easier to analyize and transform, and faster to interpret. It worth mentioning that the distinction stack vs. explicit operands by no means separates abstract machines from real hardware: there some examples

of the latter which implement stack architecture (the most notable, probably, is now late Intel 8087 floating-point coprocessor).

In our case we favoured stack machine over others since the compilation to stack code posesses some nice invariants and its correctness can be formally assessed easily. In addition the archiecture of x86 is very close to two-address code, so dealing with stack abstract machine allows us to consider a broader class of architectures.

5.2 Syntax and Semantics

The syntax of abstract stack machine language is shown below:

$$
\begin{array}{rcl}
\mathscr{I} & = & \texttt{BINOP} \otimes \\
 & & \texttt{CONST}\ \mathbb{N} \\
 & & \texttt{LD}\ \mathscr{X} \\
 & & \texttt{ST}\ \mathscr{X} \\
 & & \texttt{READ} \\
 & & \texttt{WRITE} \\
\mathscr{S} & = & \epsilon \\
 & & \mathscr{I}\,\mathscr{S}
\end{array}
$$

We have here two syntactic categories — *instructions* $\mathscr{I}$ and *programs* $\mathscr{P}$.

There are six types of instructions, and only two types of programs — an empty program ϵ and a composite program which consists of an instruction and a residual program. In essence the programs of stack machine are just lists of instructions. Some of instructions have operands: for `BINOP` an operand is a name of binary operator from the source language, for `CONST` it is a natural number, for `LD` and `ST` — the names of variables. We can notice that the language of stack machine is very simple, much simpler than that for the straight-line programs. Yet it possesses enough expressive power to perform the same calculations as an arbitrary straigh-line program! We assess this property by implementing a compiler and proving its correctness.

Before giving a formal semantics for stack machine we first give an informal description of how it functions. The stack machine operates in a similar environment as straight-line programs. It reads and writes numbers from/to a world, and it posesses an internal state which binds (some) variables names to integer values. Beside that, stack machine has a stack of integers at its discretion. In a nutshell, this stack contains temporary values which are nowhere to place otherwise. The stack machine program executes instruction by instruction starting from the first instruction. Each instruction modifies the configuration of the stack machine (state, stack, or world) and can either succeed or fail (crash). The machine stops when there are no instructions left to execute; in this case the contents of the output stream is taken as the result of stack program evaluation.

We describe this behavior, again, using a big-step operational semantics. First, we define the extended configuration $\mathscr{C}_{SM}$ for stack machine as

$$c \xRightarrow[SM]{\epsilon} c \qquad [\text{STOP}_{SM}]$$

$$\frac{\langle \sigma,\ (x \oplus y)s,\ \omega\rangle \xRightarrow[SM]{p} c'}{\langle \sigma,\ yxs,\ \omega\rangle \xRightarrow[SM]{[\texttt{BINOP}\ \otimes]p} c'} \qquad [\text{BINOP}_{SM}]$$

$$\frac{\langle \sigma,\ zs,\ \omega\rangle \xRightarrow[SM]{p} c'}{\langle \sigma,\ s,\ \omega\rangle \xRightarrow[SM]{[\texttt{CONST}\ z]p} c'} \qquad [\text{CONST}_{SM}]$$

$$\frac{\langle z,\ \omega'\rangle = \textbf{read}\ \omega,\ \langle \sigma,\ zs,\ \omega'\rangle \xRightarrow[SM]{p} c'}{\langle \sigma,\ s,\ \omega\rangle \xRightarrow[SM]{[\texttt{READ}]\,p} c'} \qquad [\text{READ}_{SM}]$$

$$\frac{\langle \sigma,\ s,\ \textbf{write}\ z\ \omega\rangle \xRightarrow[SM]{p} c'}{\langle \sigma,\ zs,\ \omega\rangle \xRightarrow[SM]{[\texttt{WRITE}]\,p} c'} \qquad [\text{WRITE}_{SM}]$$

$$\frac{\langle \sigma,\ (\sigma\ x)s,\ \omega\rangle \xRightarrow[SM]{p} c'}{\langle \sigma,\ s,\ \omega\rangle \xRightarrow[SM]{[\texttt{LD}\ x]p} c'} \qquad [\text{LD}_{SM}]$$

$$\frac{\langle \sigma\,[x \leftarrow z],\ s,\ \omega\rangle \xRightarrow[SM]{p} c'}{\langle \sigma,\ zs,\ \omega\rangle \xRightarrow[SM]{[\texttt{ST}\ x]p} c'} \qquad [\text{ST}_{SM}]$$

Figure 5.2: Big-step operational semantics for stack machine

$$\mathscr{C}_{SM} = St \times \mathbb{Z}^* \times \mathscr{W}$$

Each component of extended configuration — state, stack of integers, word — is familiar to us. Next we need to specify the big-step transition relation "$\underset{SM}{\Rightarrow}$" for stack machines. The rules of the semantics are given in Fig. 5.2. In the rules we additionaly surrounded the head instruction of a program by square brackets [...] to visually separate it from the rest of the program.

The first rule, STOP_{SM}, is at the same time the single axiom in the semantics. It tells us that an empty program does not change the configuration. This, in particular, happens when all instructions of a program were already evaluated and there is nothing left to evaluate.

All other rules follow the same pattern: they describe the effect of the first instruction of the current program, and then prescribe to evaluate the rest of the program using the same evluation relation.

Rule BINOP_{SM} describes the case when the first instruction is a binary operator. To succeed, this instruction requires atleast two integer values, x and y, to reside on the stack. If so, it combines these values using binary operator $\oplus$ and puts the result back on the stack. The correspondence between "$\otimes$" and "$\oplus$" is, of course, the same as for the semantics of straght-line programs (see a table of the page 26).

Rule CONST_{SM} corresponds to the case when the first instruction is `CONST` z. This instruction puts its operand z on the stack.

The next two symmetrical rules, READ_{SM} and WRITE_{SM}, deal with instructions `READ` and `WRITE` respectively. `READ` reads a value from input stream and puts it on the stack; if the input stream is empty the instruction fails. `WRITE` takes a valus from the top of the stack and puts it in the output stream; this instruction fails if the stack is empty.

Finally, two last rules, LD_{SM} and ST_{SM}, desribe the semantics of instructions `LD` and `ST`. Both instructions have an operand, the name of a variable. `LD` x puts the value of variable x, associated in the current state σ, on the stack. It fails if $\sigma\ x$ is undefined. `ST` x takes the top value from the stack and updates the current state σ to associate x with this value. It fails if the stack is empty.

With the transition relation $\underset{SM}{\Rightarrow}$ defined we can specify the "surface" semantics for stack machine programs $[\![\bullet]\!]_{SM}$:

$$[\![\bullet]\!]_{SM} : \mathscr{S} \to \mathbb{Z}^* \to \mathbb{Z}^*$$

$$\frac{\langle \Lambda,\ \epsilon,\ \langle i,\ \epsilon\rangle\rangle \overset{p}{\underset{SM}{\Longrightarrow}} \langle \sigma,\ s,\ \omega\rangle}{[\![p]\!]_{SM}\ i = \textbf{out}\ \omega}$$

In other words, we take an input i, create an initial configuration, consisting of an empty state Λ, empty stack ϵ and an initial world $\langle i,\ \epsilon\rangle$, then run the program and, if it eventually comes to an end with some final configuration $\langle \sigma,\ s,\ \omega\rangle$, we take output stream from this configuration's world as the result of the evaluation.

As always, we can identify the following important properties of the semantics:

- *Determinism:* for each program and each configuration there is at most one rule that can be applied; thus, if (for arbitrary i and p) $[\![p]\!]_{SM}\ i = o$ and $[\![p]\!]_{SM}\ i = o'$ then $o = o'$;
- *Compositionality*: the premise of each rule deals with programs one instruction shorter, than the conclusion. This, again, means that the principle of structural induction can be applied for proving the properties of the semantics.

It is also worth discussing the "shape" of derivation which the rules of the semantics generate. As we can see, each rule (except for the axiom) has exactly one premise, and the final configuration of the primise is at the same time the final configuration of the conclusion. This means that the derivations in this

semantics have the form of a "tower"; the top of the tower corresponds to the application of the single axiom, which transfers its initial configuration to final one unchanged, and after that this final configuration is propagated down the tower in the right-hand side configurations of all involved rules. The effects of individual instructions can be traced in the bottom-up manner in the left-hand side configurations of the rules comprising the tower. To illustrate this structure, consider the following stack machine program:

```
READ
READ
BINOP +
WRITE
```

The derivation "tower" which corresponds to the evaluation of this program for the input $\langle 2, 3\rangle$ is shown in Fig. 5.3. As always, we omit program for space considerations; the solid arrows connect the "floors" of the tower while dashed ones show the configurations' data flow.

5.2.1 Composition property

In order to assess the correctness of compilation we need to formally establish a rather expected property of stack machine programs. Let assume that we have a program p and there are two configurations c and c' such that

$$c \xRightarrow[SM]{p} c'$$

In other words, p, being evaluated for the configuration c, completes succesfully with the final configuration c'. As the instructions of p, according to the semantics, are evaluated succesively one after another this should mean that each *subprogram* (a continuous sublist of instructions) of p completes succesfully for some configurations, derived from c.

In order to reify this expectation into a formal claim we need to define a concatenation of programs "$+\!\!+$":

$$\begin{array}{rcrcl} \epsilon & +\!\!+ & p & = & p \\ \iota\, p & +\!\!+ & p' & = & \iota\,(p +\!\!+ p') \end{array}$$

We need this operation since there is no other way to "join" programs together: we cannot just take program q and program t and consider pq as a program since there is no such syntactic form. Operation "$+\!\!+$" is defined inductively on the first operand; it is, in essence, a list concatenation function.

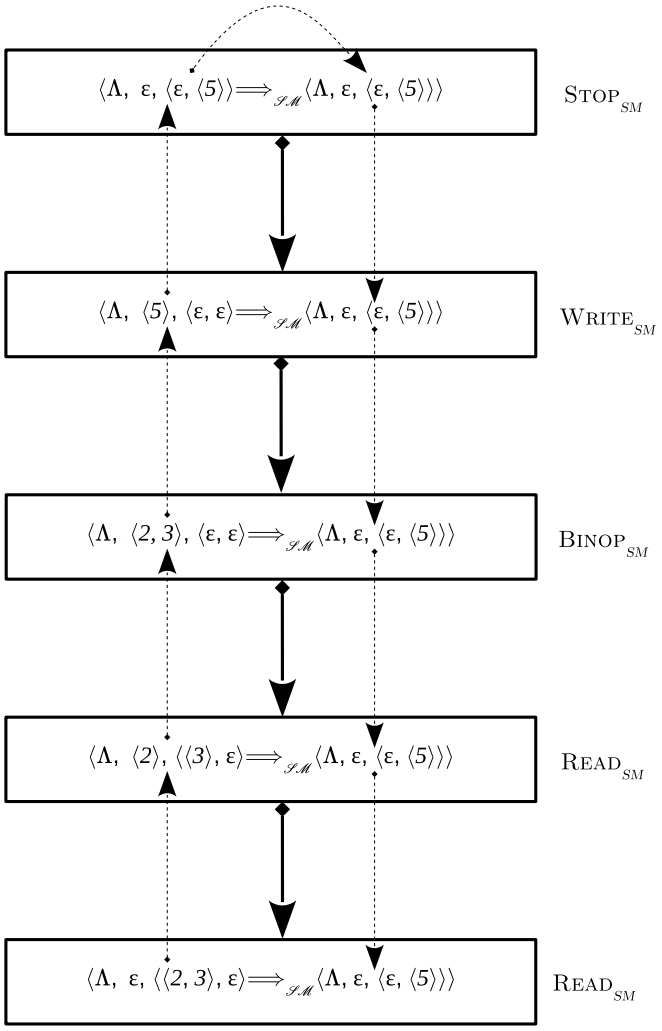


Figure 5.3: Derivation "tower" example

Now we can formulate the following lemma.

Lemma (Composition property)**.** *Let p, q, and t be stack machine programs such that $p = q +\!\!+ t$. Then if for some configurations c and c'*

$$c \xrightarrow[SM]{p} c'$$

then there exists a configuration c'' such that

$$c \xrightarrow[SM]{q} c'' \text{ and } c'' \xrightarrow[SM]{t} c'$$

Proof. The proof, of course, goes by induction on the structure of q.

The base case is $q = \epsilon$. Then $p = \epsilon +\!\!+ t = t$ and the lemma follows vacuously by taking $c'' = c$.

Let $q = \iota\, q'$. Then $p = q +\!\!+ t = \iota\,(q' +\!\!+ t)$. Obviously, there exists a configuration $\widetilde{c}$ such that

$$c \xrightarrow[SM]{\iota\, \epsilon} \widetilde{c}$$

since there is no way to complete the evaluation of p with the final configuration c' if its first instruction fails. So when we evaluate p in configuration c we have to evaluate $q' +\!\!+ t$ in configuration $\widetilde{c}$. But we can apply induction hypothesis here since q' is a subprogram of q. Thus, there exists c'' such that

$$\widetilde{c} \xrightarrow[SM]{q'} c'' \text{ and } c'' \xrightarrow[SM]{t} c'$$

But from

$$c \xrightarrow[SM]{\iota\, \epsilon} \widetilde{c}$$

and

$$\widetilde{c} \xrightarrow[SM]{q'} c''$$

and the determinism of the semantics it follows

$$c \xrightarrow[SM]{\iota\, q' = q} c''$$

which completes the proof. □

The conversion of this property is also true: let p and q be such programs that

$$c \xrightarrow[SM]{p} c' \text{ and } c' \xrightarrow[SM]{q} c''$$

for some configurations c, c' and c''. Then

$$c \xrightarrow[SM]{p +\!\!+ q} c''$$

The proof is similar and left to the reader.

5.3 Stack Machine Interpreter

Our next task is implementing the interpreter for stack machine. Given the simplicity of the language one may wonder if this subject deserves a separate section.

It definitely would not if we, indeed, were going to present only the same kind of interpreter (actually called *simple recursive interpreter*) as for the straight-line programs language. However, in the case of abstract machines the assortment of approaches to interpreter implementation is much broader. Indeed, unlike source-level interpreters, which main feature is to follow the semantics as close as possible, abstract machine interpreters often play the role of a real runtime environment. Thus, the performance issue comes up.

We consider here three types of interpreters: the simple recursive one, iterative non-recursive, and *threaded code* interpreter. It is worth mentioning that, as long as we implement these kinds of interpreters in $\lambda^a\mathcal{M}^a$, their performance will remain roughly the same. However, "the real" abstract machine interpreters as a rule are implemented in a lower level than $\lambda^a\mathcal{M}^a$ languages — C or, perhaps, assembler, and in those languages the techniques we discuss indeed deliver essential speedup.

Our first interpreter (Fig. 5.4) literally encodes the operational semantics. Stack machine program (`insns`) is represented as a list of instructions, and each instruction is directly represented as an S-expression. The big-step transition relation is encoded using nested function `eval` which takes a configuration and a program, branches on the first instruction (if any), calculates the next configuration and recursively calls itself with the new configuration and the remaining part of the program. The branching corresponds to the choice of the semantics' rule, recursive call — to the calculation in rule's primise, and the case of empty list — to the application of the axiom. It is easy to see that the configuration of recursive calls repeats the shape of derivation "tower": unless the program is non-empty the interpreter keeps calling itself with updated configuration, and once the program ends the final configuration is propagated as the return value of resursive calls. The function `evalOp` is exactly the same as in Fig. 4.4, which is unsurprising since it is exactly the same in the semantics as well. The "surface" semantics $[\![\bullet]\!]_{SM}$ is implemented by the function `evalSM`, which takes an input stream, a program, makes initial configuration, runs the machine and returns the output stream of the final configuration (if any).

The interpeter of this kind is an ideal tool to provide a literal encoding for the semantics. However, from the performance standpoint it lacks a lot. First of all, it is easy to see that the depth of recursive calls is equal to the length of the program being interpreted. This means that for long programs the interpreter

```
fun evalSM (input, insns) {
  fun eval (c@[st, s, w], insns) {
    case insns of
      {} → c
    | i : insns →
        eval (
          case i of
            READ       → let [n, w] = readWorld (w) in
                         [n : st, s, w]
          | WRITE      → let n : st = st in
                         [st, s, writeWorld (n, w)]
          | BINOP (op) → let y : x : st = st in
                         [evalOp (op) (x, y) : st, s, w]
          | CONST (n)  → [n : st, s, w]
          | LD    (x)  → [s (x) : st, s, w]
          | ST    (x)  → let n : st = st in
                         [st, s ← [x, n], w]
          esac,
          insns
        )
    esac
  }

  eval ([{}, emptyState, createWorld (input)], insns)[2].getOutput
}
```

Figure 5.4: Simple recursive interpreter

most likely will crush due to the call stack overflow (unless $\lambda^a\mathcal{M}^a$ compiler implements a special transformation called *tail call elimination*). Then, we represent programs as lists, which is completely ok if the objective is to follow the specification literally. However, it is rather obvious that this representation is excessive: as programs do not change in the course of interpretation we pay extra space taken by lists for nothing. In addition lists are not random-access structures — we can not get their arbitrary elements without extra efforts. In our case when programs are executed strictly successively this does not matter as we only take the tail of a list (in *constant* time) each time we switch to the next instruction. However, for more advanced languages with control flow this will no longer be the case, and the slow access to an arbitrary instruction can become an issue. Similarly, we represent states as functions in strong accordance with the semantics; however is is obvious that from performance standpoint this representation is not the best choice — there is only a finite number of variables in each program, and this number does not change.

Thus, our next version is a *simple iterative* iterpreter. We make the following changes to the program and configuration representations:

- we represent programs as *arrays* of instructions rather then lists;
- we represent variables by numbers, not names;
- we represent states as arrays of numbers, indexed by variables.

These changes require a conversion of initial program and configuration representations to the new one — we need, for example, enumerate all varables in given program, etc. The implementation of this simple conversion is left to the reader. The iterative interpreter is shown in Fig. 5.5. The interpreter takes an input, a program as array of instructions, and a number of variables in the program. As the interpreter simply iterates over the instructions' array (using integer variable `ip`, "instruction pointer"), we no longer need nested recursive function. We keep stack, world, and state as mutable data structures and provide helper functions `push` and `pop` to encapsulate conventional operations for the stack. The state is represented as an array of integers, and we use regular array access constructs of $\lambda^a\mathcal{M}^a$ to operate on the state. Otherwise, the implementation resembles that for simple recursive case. We still represent the instructions as S-expressions, although now the arguments of instructions `LD` and `ST` are integers, not strings. If we were implementing the interpreter in a lower level language like C we, probably, would use another enoding for the instructions using integer numbers/bit-field structures, which would improved the performance even more, but as the demonstration of the idea this version is sufficient.

There is, however, one interesting and important observation which we have to make: as we switched from state-as-functions to state-as-arrays representation we lost the ability to detect the use of non-initialized variables! Thus, strictly speaking we *do not* have a fully correct interpreter anymore, but only a partially correct. For example, the semantics of the program

```
LD x
WRITE
```

is undefined function, but simple iterative interpreter would write 0 for each input, this implementing a *different* semantics. We could, of course, resurrect this lost feature by representing states not by arrays of integers, but by arrays of, say, options; this choice, however, would hamper the performance, questioning the very idea of switching the representation for states. This is rather a typical scenario in the field: in order to make the implementation more efficient we sometimes have to deviate from the puristic interpretation of the semantics by allowing programs to ignore some (but not all!) "pathological" situations. We can not, however, take these decisions arbitrarily; partial correctness sets the boundary which we should not cross.

Our final kind of interpreter is *direct threaded code* interpreter. The idea behind this approach is quite simple: we represent each instruction as a function which, being called, performes the same actions as this instuction. Again, as long as we write in $\lambda^a\mathcal{M}^a$ this representation probably would not deliver us any performance gain; however this technique is a yet another good pattern when using lower-level languages. The benefit of threaded code is that we do not need pattern matching anymore: as each instruction "knows" what to do itself in the main loop of the interpreter we just need to call each function from program array. The implementation of the interpreter is shown in Fig. 5.6. Similarly to the previous case we represent the elements of configurations as mutable variables, and for each instruction we provide a function which encodes its semantics. If the instruction in question does not have arguments we can write such a function once and for all; otherwise we need to capture the arguments in the enclosed nested function definition. The main function of the interpreter, again, takes an input, a program in the form of array of functions, and a number of variables. It initializes the configuration and runs throught the program, calling each instruction function.

5.4 Stack Machine Compiler

In this section we describe a compiler from straigh-line programs language to the abstract stack machine and prove its correctness. The first question is what "describe" means. Recall, we already dealt with two understandings of a compiler:

- as a total syntactic transformation from one programming language to another;
- as a *program* which implements such a transformation.

Does it not resemble *semantics* and *reference interpreter*? Let us want to implement a compiler

$$comp : L_1 \to L_2$$

```
var st;

fun push (n) {
  st := n :: st
}

fun pop () {
  let x :: st' = st in
  st := st';
  x
}

fun evalSMiterative (input, [insns, numVars]) {
  var ip, w = createWorld (input);
  val s = initArray (numVars, fun (_) {0});

  st := {};

  for ip := 0, ip<insn.length, ip := ip+1 do
    case insn [ip] of
      READ       → let [n, w'] = readWorld (w) in
                   push (n);
                   w := w'
    | WRITE      → w := writeWorld (pop (), w)
    | BINOP (op) → let y = pop () in
                   let x = pop () in
                   push (evalOp (op) (x, y))
    | CONST (n)  → push (n)
    | LD    (x)  → push (s [x])
    | ST    (x)  → s[x] := pop ()
    esac
  done;

  getOutput (w)
}
```

Figure 5.5: Simple iterative interpreter

```
var st, s, w;

fun binop (op) {
  fun () {
    let y = pop () in
    let x = pop () in
    push (evalOp (op) (x, y))
  }
}

fun ld (x) {
  fun () {
    push (s [x])
  }
}

fun st (x) {
  fun () {
    s [x] := pop ()
  }
}

fun const (n) {
  fun () {
    push (n)
  }
}

fun read () {
  let [n, w'] = readWorld (w) in
  s [x] := n;
  w      w'
}

fun write () {
  w := writeWorld (pop (), w)
}

fun evalSMthreaded (input, [insns, numVars]) {
  var ip;

  s  := initArray (numVars, fun (_) {0});
  st := {};
  w  := createWorld (input);

  for ip := 0, ip<insn.length, ip := ip+1 do
    insn [ip] ()
  done;

  getOutput (w)
}
```

Figure 5.6: Threaded code interpreter

$$\begin{array}{rcl} [\![x]\!]^{\mathscr{E}}_{comp} & = & [\texttt{LD}\ x] \\ [\![n]\!]^{\mathscr{E}}_{comp} & = & [\texttt{CONST}\ n] \\ [\![A \otimes B]\!]^{\mathscr{E}}_{comp} & = & [\![A]\!]^{\mathscr{E}}_{comp} +\!\!+ [\![B]\!]^{\mathscr{E}}_{comp} +\!\!+ [\texttt{BINOP}\ \otimes] \end{array}$$

Figure 5.7: Stack machine compiler for expressions

$$\begin{array}{rcl} [\![x{:=}e]\!]_{comp} & = & [\![e]\!]^{\mathscr{E}}_{comp} +\!\!+ [\texttt{ST}\ x] \\ [\![\textbf{read}\ \ (x)]\!]_{comp} & = & [\texttt{READ}\,][\,\texttt{ST}\ x] \\ [\![\textbf{write}\ \ (e)]\!]_{comp} & = & [\![e]\!]^{\mathscr{E}}_{comp} +\!\!+ [\texttt{WRITE}] \\ [\![S_1;\, S_2]\!]_{comp} & = & [\![S_1]\!]_{comp} +\!\!+ [\![S_2]\!]_{comp} \end{array}$$

Figure 5.8: Stack machine compiler for statements

from L_1 to L_2. Obviously, we can consider it as a semantics of L_1 with semantic domain L_2. This might look contrived or exotic, but, in fact, in the earlier days there existed a (rather extreme) standpoint that the semantics of a language is determined by its compiler. If compiler is a semantics, then its implementation as a program is this semantics' interpreter! While we may argue if this point of view is desirable from a methodological perspective, we can atleast agree that even if not we can still use precisely the same tools we used to specify the semantics to specify a compiler.

In a nutshell, we have to define a transformation

$$[\![\bullet]\!]_{comp} : \mathscr{S} \to \mathscr{P}$$

such that for arbitrary program $p \in \mathscr{S}$

$$[\![[\![p]\!]_{comp}]\!]_{SM} = [\![p]\!]_{\mathscr{S}} \qquad (\clubsuit)$$

As our source language consists of two syntactic categories, we additionally need to provide a compiler for expressions $[\![\bullet]\!]^{\mathscr{E}}_{comp}$. In fact, due to the simplicity of both source and target languages, the compiler is quite simple as well. The specification of compiler is shown in Fig. 5.7 and Fig. 5.8. As we can see we used an already familiar denotational style when the result of compilation is directly specified. All expected properties — compositionality and determinism, — are provided trivially, so we atlest can immediately conclude that we specified a total function. But why this function indeed provides a correct compilation? We consider this question in the next section.

5.4.1 Correctness of the Compiler

We are going to prove the correctness of the compiler, the property formally expressed by the equation ($\clubsuit$). Both $[\![\bullet]\!]_{SM}$ and $[\![\bullet]\!]_{\mathscr{S}}$ are defined in terms of corresponding transition relations, so we, apparently, need some statements concerning those relations as well. In addition we need to relate somehow the

transition relation " $\Rightarrow_{SM}$ " for stack machines and denotational semantics for expressions $[\![\bullet]\!]_{\mathscr{E}}$. Fortunately, the overall task turns out to be not so demanding as it might look at the first glance if an appropriate set of lemmas is formulated.

Lemma. *Correcntess of expression compiler. Let $e \in \mathscr{E}$ be an expression, $\sigma \in St$ — some state, $s \in \mathbb{Z}^*$ — stack, $\omega \in \mathscr{W}$ — some world, and $z \in \mathbb{Z}$ — some number. Then*

$$\langle \sigma,\, s,\, \omega \rangle \xRightarrow{[\![e]\!]^{\mathscr{E}}_{comp}}_{SM} \langle \sigma,\, zs,\, \omega \rangle \iff [\![e]\!]_{\mathscr{E}}\, \sigma = z$$

Proof. Both directions can be proven by structural induction on e. We do this only in the reverse direction ("$\Leftarrow$") and leave the other to the reader.

Base case. Let $e = x \in \mathscr{X}$ By the definitions of $[\![\bullet]\!]^{\mathscr{E}}_{comp}$ and $[\![\bullet]\!]_{\mathscr{E}}$ we have:

$$\begin{array}{rcl} [\![x]\!]^{\mathscr{E}}_{comp} & = & [\mathtt{LD}\, x] \\ [\![x]\!]_{\mathscr{E}}\, \sigma & = & \sigma\,(x) \end{array}$$

By the condition of the lemma $\sigma\,(x) = z$. Finally, by the definition of " $\Rightarrow_{SM}$ " we have

$$\langle \sigma,\, s,\, \omega \rangle \xRightarrow{[\mathtt{LD}\, x]}_{SM} \langle \sigma,\, zs,\, \omega \rangle$$

The second case ($e = n \in \mathbb{N}$) is established similarly.

Induction step. Let $e = l \otimes r$ and let the lemma holds for l and r. Again, by the definitions of $[\![\bullet]\!]^{\mathscr{E}}_{comp}$ and $[\![\bullet]\!]_{\mathscr{E}}$ we have:

$$\begin{array}{rcl} [\![l \otimes r]\!]^{\mathscr{E}}_{comp} & = & [\![l]\!]^{\mathscr{E}}_{comp} +\!\!+ [\![r]\!]^{\mathscr{E}}_{comp} +\!\!+ [\mathtt{BINOP}\, \otimes] \\ [\![l \otimes r]\!]_{\mathscr{E}}\, \sigma & = & [\![l]\!]_{\mathscr{E}}\, \sigma \oplus [\![r]\!]_{\mathscr{E}}\, \sigma \end{array}$$

From the condition of the lemma we know that $[\![l]\!]_{\mathscr{E}}\, \sigma = x$ and $[\![r]\!]_{\mathscr{E}}\, \sigma = y$ for some x and y, and that

$$\langle \sigma,\, s,\, \omega \rangle \xRightarrow{[\![l]\!]^{\mathscr{E}}_{comp}}_{SM} \langle \sigma,\, xs,\, \omega \rangle$$
$$\langle \sigma,\, xs,\, \omega \rangle \xRightarrow{[\![r]\!]^{\mathscr{E}}_{comp}}_{SM} \langle \sigma,\, yxs,\, \omega \rangle$$

Applying the composition property twice we get

$$\langle \sigma,\, s,\, \omega \rangle \xRightarrow{[\![l]\!]^{\mathscr{E}}_{comp} +\!\!+ [\![r]\!]^{\mathscr{E}}_{comp}}_{SM} \langle \sigma,\, yxs,\, \omega \rangle$$
$$\langle \sigma,\, s,\, \omega \rangle \xRightarrow{[\![l]\!]^{\mathscr{E}}_{comp} +\!\!+ [\![r]\!]^{\mathscr{E}}_{comp} +\!\!+ \mathtt{BINOP} \oplus}_{SM} \langle \sigma,\, (x \otimes y)s,\, \omega \rangle$$

But $x \oplus y = [\![l \otimes r]\!]\,\sigma$ which exists by the condition of the lemma.

□

This lemma ensures the correctness of compliation for expressions: a compiled from an expression stack machine program being executed for some configuration succesfully finishes and leaves on the stack the value of this expression in this configuration's state and if and only if the same value for the same state exists according the semantics of expressions. Neither state nor world is changed, and all the stack values below the top are preserved.

Lemma. *Correctness of compiler. Let $p \in \mathscr{S}$ be a straight-line program, σ, $\sigma' \in St$ — some states, ω, $\omega' \in \mathscr{W}$ — some worlds, and s, $s' \in \mathbb{Z}^*$ — some stacks. Then*

$$\langle \sigma,\, \omega\rangle \xRightarrow[\mathscr{S}]{p} \langle \sigma',\, \omega'\rangle \iff \langle \sigma,\, s,\, \omega\rangle \xRightarrow[SM]{[\![p]\!]_{comp}} \langle \sigma',\, s',\, \omega'\rangle$$

Proof. Now we prove the lemma in a forward direction, and leave the opposite one for the reader. The proof, of course, goes by structural induction on p.

Base case. We only prove the base case for assignment since the other cases can be carbon-copied. Let us have $p = x{:=}e$. By the condition of the lemma we have

$$\langle \sigma,\, \omega\rangle \xRightarrow[\mathscr{S}]{x := e} \langle \sigma',\, \omega'\rangle$$

By the definition of "$\Rightarrow_{\mathscr{S}}$"

$$\begin{array}{rcl} \sigma' & = & \sigma\,[x \leftarrow [\![e]\!]_{\mathscr{E}}\,\sigma] \\ \omega' & = & \omega \end{array}$$

By the definition of $[\![\bullet]\!]_{comp}$

$$[\![x{:=}e]\!]_{comp} = [\![e]\!]^{\mathscr{E}}_{comp} +\!\!+ [\mathtt{ST}\,x]$$

By the correctness of the expression compiler

$$\langle \sigma,\, s,\, \omega\rangle \xRightarrow[SM]{[\![e]\!]^{\mathscr{E}}_{comp}} \langle \sigma,\, ([\![e]\!]_{\mathscr{E}}\,\sigma)s,\, \omega\rangle$$

By the definition of "$\Rightarrow_{SM}$"

$$\langle \sigma,\, ([\![e]\!]_{\mathscr{E}}\,\sigma)s,\, \omega\rangle \xRightarrow[SM]{\mathtt{ST}\,x} \langle \sigma\,[x \leftarrow [\![e]\!]_{\mathscr{E}}\,\sigma],\, s,\, \omega\rangle$$

By the composition property for stack machine programs

$$\langle \sigma,\, s,\, \omega\rangle \xrightarrow[SM]{[\![e]\!]^{\mathscr{E}}_{comp} +\!+ [\mathtt{ST}\, x]} \langle \sigma\,[x \leftarrow [\![e]\!]_{\mathscr{E}}\,\sigma],\, s,\, \omega\rangle$$

which completes the proof for the base case.

Induction step. There is only one construct which requires induction step to be proven. Let us have $p = s_1\,;s_2$. By the definition of " $\underset{\mathscr{S}}{\Rightarrow}$ " there exist $\sigma'' \in St$ and $\omega'' \in \mathscr{W}$ such that

$$\frac{\langle \sigma,\, \omega\rangle \xrightarrow[\mathscr{S}]{s_1} \langle \sigma'',\, \omega''\rangle \quad \langle \sigma'',\, \omega''\rangle \xrightarrow[\mathscr{S}]{s_2} \langle \sigma',\, \omega'\rangle}{\langle \sigma,\, \omega\rangle \xrightarrow[\mathscr{S}]{s_1\,;s_2} \langle \sigma',\, \omega'\rangle}$$

By induction hypotheses

$$\langle \sigma,\, s,\, \omega\rangle \xrightarrow[SM]{[\![s_1]\!]_{comp}} \langle \sigma'',\, s'',\, \omega''\rangle$$

and

$$\langle \sigma'',\, s'',\, \omega''\rangle \xrightarrow[SM]{[\![s_2]\!]_{comp}} \langle \sigma',\, s',\, \omega'\rangle$$

for some stack s''. By the composition property of stack machine programs the lemma follows.

□

One might notice that the stack behaviour in the last lemma is somewhat weird: no certain conditions are put on it contents. This is because the stack, actually, is only temporarily used within the execution of stack machine subprogram, compiled from a statement. Once the execution of each such subprogram is finished, no extra values remain on the stack. In other words, if (in the conditions of the lemma)

$$\langle \sigma,\, s,\, \omega\rangle \xrightarrow[SM]{[\![p]\!]_{comp}} \langle \sigma',\, s',\, \omega'\rangle$$

then $s = s'$. We leave the proof to the reader.

5.4.2 Optimality Property

The final question we are going to address is how good the compiler we've described is. Obviously it is good enough to be total and fully correct, but isn't it what we expect from any compiler? To be more specific, we are interested in the *efficiency* of compiled programs.

In more formal terms, let μ be some way to measure the "efficiency" of programs in a language $\mathscr{M}$:

$$\mu : \mathscr{M} \to \mathbb{R}$$

The nature of μ is not important for now; it is sufficient to assume that μ if total and allows us to compare programs (the less μ the "better" program is). Let us have a compiler

$$comp : \mathscr{L} \to \mathscr{M}$$

We say that *comp* is *optimal* w.r.t. μ iff for arbitrary $p \in \mathscr{L}$ and arbitrary $q \in \mathscr{M}$ such that $p \equiv q$ the following relation holds:

$$\mu\,(comp\,(p)) \leq \mu\,(q)$$

In other words, *comp* provides the "best" (w.r.t. μ) equivalent to p program in the target language.

How hard is to build an optimal (in this sense) compiler? From the computability theory we know, that if $\mathscr{L}$ is Turing-complete, then this problem is undecidable and thus no optimal compiler can exist[1]. But what about our case? The language of straight-line programs is obviously not Turing-complete; in fact, it is very weak — we can not, for example, even calculate exponent, which is primitively recursive. Actually, we can only calculate *polynomials* of multiple variables. Let take the simplest measure for stack machine program — its length:

$$\begin{aligned} \mu\,(\epsilon) &= 0 \\ \mu\,(\iota\,p) &= 1 + \mu\,(p) \end{aligned}$$

Do we have an optimal compiler w.r.t. this concrete μ?

A quick analysis reveals — no, we don't. Indeed, take for example the program **write** (2+3). By the definition of our compiler

$$[\![\textbf{write}\ \ (2{+}3)]\!]_{comp} = [\texttt{CONST 2; CONST 3; BINOP+; WRITE}]$$

while stack machine program

$$[\texttt{CONST 5; WRITE}]$$

does exactly the same and shorter. But maybe we can implement some improvements like constant folding etc., which will fix the inoptimality?

[1]For non-trivial μ; we can take, for example, $\mu \equiv 0$, which would make *any* compiler optimal.

The theory says no. Indeed, let $p(x_1, \ldots, x_k)$ be a polynomial of multiple natural variables and natural coefficients. An equation

$$p(x_1, \ldots, x_k) = 0$$

is called *diophantine*. Can we find its roots? As each x_i spans a countable set, so do all tuples $x_1, \ldots, x_k$. We can systematically enumerate all these tuples and evaluate the value of p for each. This simple procedure will deviler us a root provided that the root exists. However, if there is no natural root, the procedure will continue infinitely. Can there be other, more advanced procedure to find the roots of a diophantine equation, which would not loop forever if no roots exist?

This very question is known as 10th Hilbert Problem, and it took almost 70 years to solve it negatively. Now we know that the problem of determining the *lack* of roots for diophantine equation is undecidable.

Let's now assume that we have an optimal compiler for straight-line programs language. Let's take an arbitrary polynomial $p(x_1, \ldots, x_k)$ and compile the following program:

read (x_1);
...
read (x_k);
write $(p(x_1, \ldots, x_k)$!= $0)$

If p does not have roots, the expression $p(x_1, \ldots, x_k)$!= 0 will always be evaluated to 1. Look at the compiled stack machine program: if it consists of only two instructions [`CONST` 1; `WRITE`] then p does not have roots. Thus, assuming the existence of an optimal compiler we acquired the decision procedure for a problem which is already known to be undecidable, which means that optimal compilation is undecidable as well. This proof technique is called "Turing reduction"; it constitutes one of the main tools to establish undecidabilities.

Chapter 6

A Compiler for x86/32

In this section, finally, we will consider a codegenerator for x86/32 processor, the principal study object of the whole course. In this particular chapter we will only deal with the simplest compiler for straight-line programs, but as the source language evolves the native-code compiler will get more and more features.

Of course the main question for now is what implementing a native-code compiler amounts to. Luckily, as we will see shortly, this task is not much different from what we already can do. In order to turn our source program into machine code we only need to convert it into a text in an *assembly* language. Then we entrust the GCC toolchain to make the rest of the work — compile this assembly program into object file, link (multiple) compiled object files and some libraries into executable, etc. This approach is by no means exotic — nowdays the majority of compilers are implemented exactly following this roadmap which makes it possible to reuse all stages of compilation starting from object file generation by many compilers (and thus avoid of making a lot of similar errors anew).

An important methodological difference of what we are going to do in this chapter is that we *will not* discuss the operational semantics of the assembly language. The motivation is that this language is very similar in its generic features to stack machine language, which we already dealt with. As we generate native code from stack machine, the compiler assumed to be very simple, and its formal description is expected to be superfluous. On the other hand there are a lot of tiny simple details in machine architecture which would make formal description boring and cumbersome, so we better discuss them in informal terms. But this does not mean that assembly languages cannot be properly described in formal terms, and there are lots of witnesses of the opposite.

6.1 Hardware Architecture

In this section we consider some basic principles of digital hardware organization, describe how actual hardware works, what components it is comprised

Figure 6.1: John von Neumann

of, and how it interprets machine program. All these subjects are topics of interest for the field of *hardware architecture*; we only scratch the surface of this very interesting and important domain to the minimal extent needed to understand the essence of computations performed on actual hardware.

From a birds-eye view a hardware computer consists of *memory*, *central processor unit* (CPU), and *input-output* subsystem. This very general decomposition is called "von Neumann architecture" named after John von Neumann. In von Neumann architecture programs are kept in the same memory as data; this is sometimes contrasted to so-called *Harvard architecture* where a separate memory is dedicated solely to keep programs. While the majority of general-purpose computers follow von Neumann architecture the Harvard one can be found in some embedded devices. The real implementations of both approaches, of course, are much more complicated then the generic scheme: there can be more than one CPU core, the CPU core itself contains one or more *arithmetic-logic units* (ALU) and *multiplexers* (MUX) to perform conditional computations, memory subsystem is implemented using *memory controller* and can incorporate one or more levels of *cache*, there is, as a rule, a separate subsystems to handle software and hardware *interrupts*, support *virtual memory*, *virtualization*, etc. However, the general construct of a *hardwired electronic interpreter* of programs represented as sequences of bits kept in memory can be easily discovered in all digital programmable devices. This is, by the way, justifies the central role of the concept of interpreter: there is actually no way to evaluate a program other than to run it on some (perhaps, hardware) interpreter.

An important observation is that not all details of hardware organization

are visible at the application level; some of them are intentionally designed to be completely transparent to an end-user application. For example, hardware interrupts, virtual memory, cache, etc., as a rule are invisible for a regular application, which means that a compiler in the *majority of cases* (but not always!) can ignore the presence of these subsystems. Moreover, a compiler as a rule does not deal with a certain part of hardware functionality and instruction set just because the source language does not have corresponding abstractions. Finally, as a regular interpreter can be implemented in various ways even using the same implementation language, hardware platforms also can have different implementations while sharing the same "surface" architecture and instruction set. This surface, or *macroarchitecture* of x86/32 is a subject of our close attention in this chapter. But, first, we consider the very principles digital programmable hardware is based on.

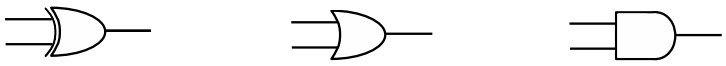
6.1.1 Principles of Digital Hardware Organization

Digital hardware in the vast majority of cases operates on *binary* data. In this form all sorts of data a computer deals with is represented as sequences of integer numbers in binary representation, i.e. in some variant of positional encoding radix 2; we assume the reader's familiarity with this construct. The advantage of this representation is that it only requires from a physical system representing one digit to be in one of two stable states. In real hardware these two states (0 and 1) are usually represented as different voltage levels measured against some baseline (ground). Binary representation, however, is not a unique solution from either historical or practical standpoint: for example, the Babbage's analytical machine (the first known, yet unfinished, project of a Turing-complete computational device) was designed to operate on decimal numbers, and many modern hardware digital architectures (including x86) support so-called binary-decimal representation in which a group of four bits represents (with some redundancy) one decimal digit.

To understand how a hardware can perform computations we first consider a simplest possible arithmetic operation: the addition of two one-bit numbers x and y. As there are only four combinations of all possible values for x and y the result of $x + y$ can be summarized with the following *finite* table:

x	y	$carry\,(x+y)$	$x \dot{+} y$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

The somewhat unexpected extra column contains so-called *carry bit*. Indeed, the sum of two one-bit numbers sometimes occupies more than one bit: 1+1=2(decimal)=10(binary). In general case the result of addition of two n-bit numbers can (at most) occupy $n + 1$ bit. When we limit the size of numbers by, say, 32 bits we can eventually arrive at a situation when the result of some computation can no longer be represented by a 32-bit number. In such case



(a) "xor" (b) "and" (c) "or"

Figure 6.2: Logical gates

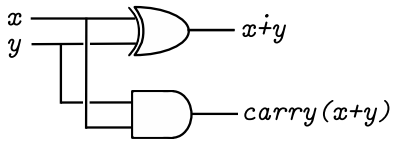
an extra, 33th carry bit can be used to identify an overflow, a situation when further computations start to delived unreliable results.

Thus, in a general case the addition of two one-bit numbers delivers *two* bits: the carry one and the partial sum (hence the denotation "$\dot{+}$"). If we look closely at the table above we can discover that the carry bit is the conjunction of the operands and the partial sum is exclusive or:

$$\begin{aligned} carry\,(x+y) &= x \wedge y \\ x \dot{+} y &= x \oplus y \end{aligned}$$

This constitutes an important observation: we will be able to perform arithmetic computations in hardware as long as we manage to implement boolean operators. There is a whole theory of *boolean circuits* which studies the theoretical properties of computations performed by the networks of interconnected primitive boolean connectives.

Let us assume that we, indeed, can implement boolean operations using some primitive hardware units (*gates*); depict those as shown in Fig. 6.2. Each gate has two *inputs* (wires on the left side) and one *output* (a wire on the right side). Then the following interconnection of gates constitutes a one-bit *adder*:



Indeed, this simple circuit literally encodes the table for ont-bit addition.

Imagine now that we want to add two *two-bits* numbers x_1x_0 and y_1y_0. By the very nature of binary positional encoding these two-bits numbers represent natural numbers $2 \times x_1 + x_0$ and $2 \times y_1 + y_0$. Let us calculate their sum:

$$2 \times x_1 + x_0 + 2 \times y_1 + y_0 = 2 \times (x_1 + y_1) + (x_0 + y_0)$$

But, again, we have to make sure that all coefficients of the degrees of 2 are either 0 or 1; in the expression above this requirement can be violated (take $x_0 = y_0 = 1$). To fix this remember that we can express the results of one-bit additions $x_1 + y_1$ and $x_0 + y_0$ in terms of partial sums and carry bits. The easiest way to sort things out is to determine which coefficients of degrees of 2 these bits contribute to:

- $x_0 \dot{+} y_0$ contributes to the coefficient of 2^0;

- $carry\,(x_0 + y_0)$ contributes to the coefficient of 2^1;
- $x_1 \dot{+} y_1$ also contributes to the coefficient of 2^1;
- $carry\,(x_1 + y_1)$ contributes to the coefficient of 2^2;

Thus, $x_0 \dot{+} y_0$ gives us the least significant bit of the overall sum. As both $carry\,(x_0 + y_0)$ and $x_1 \dot{+} y_1$ contribute to the same coefficient, we have to add them as well, obtaining yet another partial sum and carry bits. The partial sum

$$(carry\,(x_0 + y_0)) \dot{+} (x_1 \dot{+} y_1)$$

now is the only bit contributing to the coefficient of 2^1 (i.e. second-to-the least significant bit of the overall sum). However, the carry bit

$$carry\,(carry\,(x_0 + y_0), x_1 \dot{+} y_1)$$

now contributes to the coefficient of 2^2 and has to be added to $carry\,(x_1+y_1)$, which would potentially give us another two bits. Fortunately, it can be easily observed that

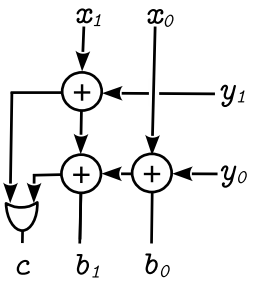
$$carry\,(carry\,(x_0 + y_0), x_1 \dot{+} y_1)$$

and

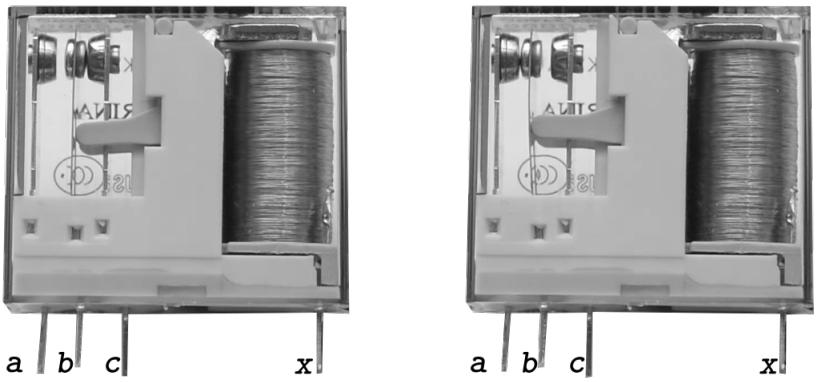
$$carry\,(x_1 + y_1)$$

can never be equal 1 at the same time. Indeed, let $carry\,(x_1 + y_1) = 1$. But then $x_1 \dot{+} y_1 = 0$, and $carry\,(carry\,(x_0 + y_0), x_1 \dot{+} y_1) = 0$. Thus, their sum is just a disjunction.

These observations can be summarized with the following boolean circuit for *two-bits adder*:



Here encircled nodes abbreviate one-bit adders with horizontal output edges associated with carry bits and vertical ones — with partial sum bits. With proper efforts (and enough time) these nodes can be substituted by the actual circuits for one-bit adders which would give a complete decomposition for two-bit adder in terms of primitive gates.



(a) Idle state (b) Powered state

Figure 6.3: A relay

Use similar ideas we can scale up the expressive power of boolean circuits almost infinitely. Combining two-bit adders gives us three-bit adders, etc. Being capable of adding numbers we can multiply (as multiplication of finite numbers can be expressed in terms of finite number of additions); using the representation in binary complement form we can use complement and addition for subtraction, which allows us to express division and taking a reminder, etc. Finally, we can easily express conditional calculations in binary logic: let us have a one-bit condition ("true" or "false") c and two n-bit numbers $x = x_{n-1} \dots x_0$ and $y = y_{n-1} \dots y_0$. Then

$$(x_{n-1} \wedge c) \dots (x_0 \wedge c) \vee (y_{n-1} \wedge \neg c) \dots (y_0 \wedge \neg c)$$

corresponds to a conditional construct

if c **then** x **else** y **fi**

Generalizing this construct we can implement a choice of one of 2^n values depending on n-bit condition, which gives us an implementation of memory. Thus, with just a few types of logical gates we can implement an impressive variety of computations.

But how the gates themselves are implemented? In the evolution of hardware multiple ways were explored and utilized starting from purely mechanical appliances made of gears, springs, shafts, etc., through simple electrical devices with relays and vaccuum tubes to the nowdays semiconductor technologies. We consider a very simple implementation of logical gates using *relays* since it does not require any specific knowledge of electronics.

A relay is a simple electro-mechanical device which typically consists of a *solenoid* (an electrical wire coiled over a cylindrical conductor core) and a group of controlled electrical contacts, one of which is mechanically connected to the solenoid's core. A relay has a few inputs/outputs:

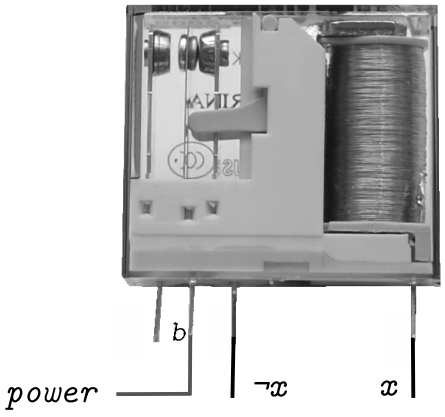


Figure 6.4: Relay-based negation implementation

- control input wires which directly connected to the solenoid;
- controlled inputs/outputs wires which connected to the contacts the relay controls.

A relay usually operates in two states. In the idle state when no electrical current flows the solenoid the contact group commutes controlled wires in some default manner. But when electric power is applied to the solenoind it produces magnetic field which makes the core to reswitsh the contacts and change the way the controlled contacts commute.

As an example consider a simple relay in Fig. 6.3. Here the contact x is control one, a, b and c — controlled. In the default state (Fig. 6.3a) when no power is given to x the contacts b and c are commuted, b and a are not. However, when an electric power is applied to x (Fig. 6.3b) the commuting scheme between a, b, and c is changed: now a and b are commuted, b and c are not.

The simplest gate we can implement with the relay is negation (Fig. 6.4). Here we left pin a unconnected, b is constantly connected to the power source. Now, in the idle state when no signal is given to the pin x there is a signal on the pin c. When a signal is given to the pin x the relay recommutes the controlled pins, and the signal on c disappears. Thus, c holds a signal when x does not and visa versa.

To implement, say, a conjunction we need *two* relays as there is two input arguments which require two input pins (one of each relay). We can successively connect power source and pins a and b of each relay as it is shown in Fig. 6.5. Now, in order to have a signal on the pin b_1 we need two signals on pins x_0 and x_1 simultaneously, which means that $b_1 = x_0 \wedge x_1$.

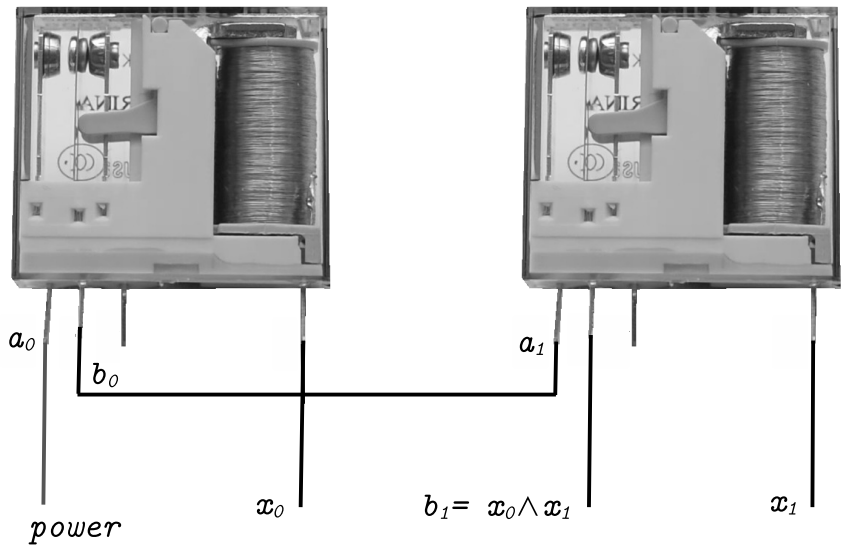


Figure 6.5: Relay-based conjunction implementation

It is well-known that conjunction and negation form a complete set of boolean connectives: any other can be expressed using these two. For example,

$$\begin{aligned} x \vee y &= \neg(\neg x \wedge \neg y) \\ x \oplus y &= (x \vee y) \wedge \neg(x \wedge y) \end{aligned}$$

etc. Thus, we already can have a relay-based implementation (albeit not very efficient and reliable) for all imaginable hardware.

In practice nowadays the majority of hardware devices is manufactured in the form of *integrated circuits*: meshes of millions of primitive devices printed (or, more specifically, *etched*) on a common semiconductor substrate (*wafer*). As a rule multiple individual integrated circuits are placed on a single wafer, which is later cut into smaller pieces — *dies*. Each die occupies just a few square millimiters of area, but using a microscope its layout can be visually observed and even photographed, resulting in a *die shot* (see Fig. 6.6). Finally, each die is packed into a protective plastic or ceramic case with a number of electrical pins sticking out, which finally constitutes a microchip. We do not present any picture of a microchip here since if you never saw one you probably have chosen a wrong course to study[1].

Althought technologically the whole process is drastically different from manual combining of relays, ideologically it is very similar and each primitive semiconductor device (such as transistor) logically functions pretty much like a relay.

[1] As of 2023.

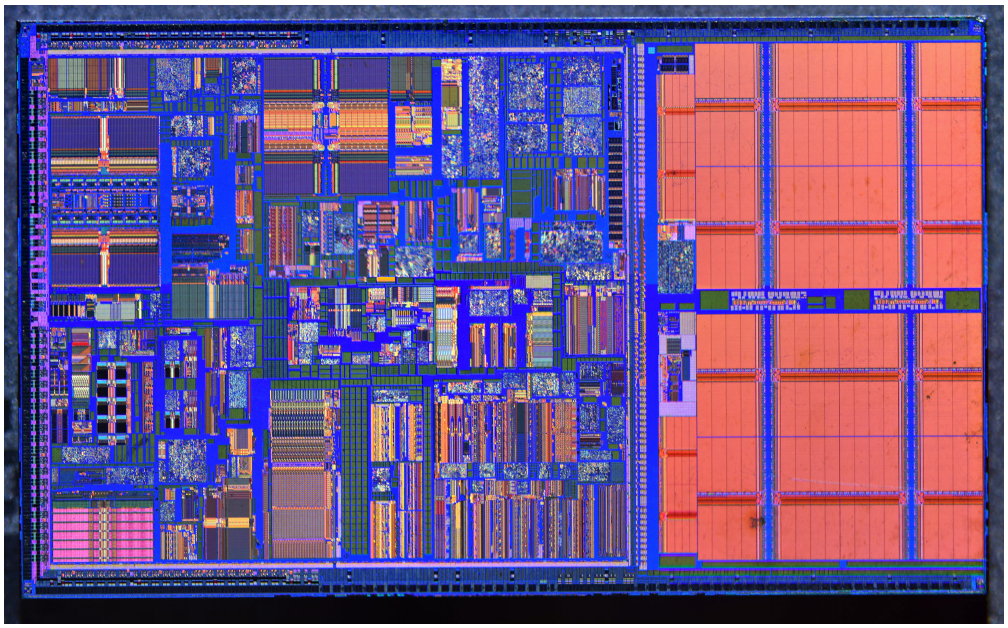


Figure 6.6: Intel Pentium III Die Shot

6.1.2 Synchronous Design

Now we know how to implement a computation in hardware: we can construct a circuit of gates which, given a certain set of binary signals on its inputs, will provide an expected result in the form of binary signals on its outputs. There is, however, a number of important observations:

- A hardware is rarely (if ever) designed to perform just a single computation. As a rule we expect it to work for a large amount of inputs which means that these inputs are going to change over time.
- All hardware gates have some *latency* — correct output signals do not appear immediately, they take some time to establish. Moreover, this time can be different for different output signals since the length of a *datapath* from inputs to a concrete output can be different. Moreover, this time can depend on the concrete state of inputs due to conditional computations.
- Since digital signal is just a voltage level when it changes (say, from 0 encoded as 0V to 1 encoded as +3V) it, actually, takes all intermediate voltage levels from 0V to +3V.

Thus, when we change the signals on inputs the output values start *incoherently* change over time. In some time the correct output values are established, but this very moment we can change the inputs, again, destructing this correct

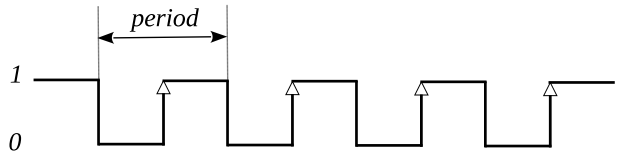


Figure 6.7: Clock Diagram

output value. Thus, when the input changes unpredictably the output values also oscillate unpredictably, making it impossible to filter out correct states from intermediate ones.

One way to cope with this problem is *synchronuous design*. In synchronuously designed hardware the whole mesh of gates is diveded into synchronized blocks. Within these blocks the signals are propagated asynchronuously, i.e. as we already described. However, on the inputs and outputs of these blocks the signals are *latched* using certain gate types and a dedicated periodical signal called *clock*. This signal is generated by devices similar to those used in conventional quartz wristwatches; its distinctive feature is square-shaped waveform with constant period. In Fig. 6.7 an example of clock signal diagram is shown. The clock signal changes its value from 0 to 1 and back over and over again with a given period. The change of the clock signal from 0 to 1 (or visa versa) is used to toggle the latches to re-lock their values (indicated on the figure by white triangles). Thus, input and outputs signals of asynchronuous blocks remain constant during the period which eliminated their oscillation. The clock period is carefully chosen in such a way that each asynchronuous block have enough time to stabilize its correct outputs by the end of the period regardless the contents of its input signals.

From this we can derive a number of important observations:

- Synchronuous hardware logically operates in a discrete time: it consumes inputs discretely (once per a clock period) and outputs results discretely (once per a clock period).
- This time discretization is global (the same for the whole device)[2].
- The period is determined by an asynchronuous block with the highest latency. Thus, contrary to a conventional beleif, in hardware there is spoiling a wedding for one that is missing.

Besides synchronuous design there is also a symmetrical idea of *asynchronuous* hardware organization according to which different computational units individually agree with each other on the reliability of their results. This idea has its merits and drawbacks; however, the vast majority of hardware devices nowdays is synchronuos, so we stick with this model for the time being.

[2]There actually can be designs in which different subparts of a device function with different clock rates, but this require special efforts.

6.1.3 Programmability

6.1.4 x86 Macroarchitecture

6.2 Code Generation with Symbolic Interpreter

6.3