Syntax and Semantics of Programming Languages

Lecture Notes

*** DRAF *** DRAT *** DRFT ***

Kent Petersson

May '90

Programming Methodology Group,
Dept. of Computer Science,
Univ. of Göteborg and Chalmers,
S-412 96 Göteborg,
Sweden

Email: kentp@CS.Chalmers.SE

Printed: April 9, 1997

Preface

These lecture notes have been compiled for a course on syntax and semantics of programming languages given at INCO, Universidad de la Republica, Montevideo, Uruguay, during two weeks in April 1990. Some editing has been done after the course in order to correct mistakes and make the presentation a little bit easier to follow.

The notes are based on previous lecture notes from the following courses that I have given at the Department of Computer Science at Chalmers University of Technology:

- Computability [34].
- Semantics of Programming Languages [33].
- Lambda Calculus and Functional Programming Languages.
- Syntax of Typed Programming Languages.

Göteborg, May 1990
Kent Petersson

Contents

Chapter 1

Introduction

This course will deal with different aspects of programming languages and we will start by discussing the question: *Why should we study programming languages?*

Computer science as a subject is centered around the notion of *computation*. The study of computations could be conducted from many viewpoints. One could, for example,

1. describe how to perform a computation to solve a particular problem,
2. study actual machines that perform computations,
3. calculate how much resources that must be spent in particular computations,
4. verify that a particular computation solves a problem or a class of problems, or
5. study abstractions which are important in the solutions of many problems,

It is clear, at least to me, that the subject that deals with how to *express* computations is very fundamental in computer science. In these course notes we shall discuss questions about the languages we use for expressing computations and how these languages are defined.

First I will try to present a little bit about programming languages by introducing some well-known words which are used in connection with programming languages.

Concrete Syntax A language is, from the point of view of concrete syntax, nothing but a set of strings over an alphabet. We use *grammars* to describe the concrete syntax of programming languages (Chomsky grammars). Languages are classified according to the expressiveness of the grammars that describe them.

Parsing A parser determines if a given string belongs to a language or not, and if the string is in the language the parser must also give some evidence of that, for example a derivation or a parse tree. Parsing takes us from the concrete to the abstract syntax.

Abstract Syntax When talking about abstract syntax we are interested in the *structure* of the programs in a language rather than the strings themselves. Instead of seeing a language as an unstructured set of strings we are interested in the structure of the elements and how they are generated. This is related to view a language as an *inductive definition*.

Type systems In grammars we use syntactical categories such as expressions and commands for expressing how the sentences of a programming language are formed. These notions are however not fine enough to give a precise definition of the meaningful expressions. The expressions are therefore further divided into different types and a type system is given which defines how the meaningful expressions may be formed.

The concrete syntax of a programming language is mostly defined by a context free grammar. Those aspects of the syntax, for example type systems, which cannot be expressed by such a grammar have often been called *static semantics* and almost never been formalized properly.

In some logics for programming, for example Per Martin-Löf's type theory [26, 32], the type system is the specification language of the programming logic and types are the same as specifications. Type correctness is then exactly the same as program correctness. An object has a certain type exactly when it satisfies the specification expressed by the type.

Semantics The purpose of the semantics of a programming language is to give *meaning* to every syntactically correct expression in the programming language. The definition of the semantics of a programming language is the basis (specification) for interpreters and compilers for the language.

Programming Logic The programming logic is used to reason about programs in the programming language; to prove correctness, to prove that two programs have equal behavior and so on. (Specification language??)

People often ask themselves where the type system of a programming language should belong. Does it belong to the syntax, semantics or to the logic of the language. There are different views, which perhaps shows that there are different opinions of the rôle of a type system in a programming language.

Syntax One way to view the type system is that it defines the well-formed programs of the language. A type error is essentially nothing but a syntactical error. Examples of this view are programming languages such as ML, ALGOL68, Pascal and Ada.

Semantics In type free languages, there is just one type and therefore no type errors. It is syntactically legal to add a list and pair, and the meaning of this is `type-error`. Programming languages that take this view are for example LISP and Prolog.

Logic In some formalisms the paradigm propositions-as-types (types-as-propositions?) have been used. Types are here used for expressing properties of a program, for example the complete behavior of the program. A type is a *specification* of a program. This is for example the view of types in Per Martin-Löf's type theory.

The traditional way of dividing the study of languages was introduced by the philosopher Charles Morris, who, in the 1930:ies, suggested that one should use the word *semiotik* for the general study of languages. Furthermore he divided the study of languages into three different areas:

1. Syntax — the study of the relation between the symbols of a language without any interpretation at all.

2. Semantics — the study of the meaning of the symbols

3. Pragmatics — the study of how a language is used.

It is not at all clear where to draw the borderline between syntax and semantics. Let us make some quotes from a couple of different authors of books and lecture notes:

> *Chris Wadsworth (Edinburgh, 1978)*
> "Syntax is straightforwardly viewed as being concerned with delineating the legal forms (phrases, expressions, etc.) of the language but too often one finds that "semantics" is then taken as being everything else, confusing "what an expression means" with "how does one compute an expression's value" two equally important aspects of the study of languages but nonetheless distinct called *semantics* and *pragmatics*."

> *Joe Stoy (Denotational Semantics)*
> "We may think of programming languages (and indeed mathematical models in general) in either of two ways. You may regard them as free objects – as objects of study in their own right without any implied meaning – or we may think of them together with an *interpretation* of their meaning. Syntax deals with the free properties of a language. The sort of questions that arises here are: Is X a grammatically correct program? What are the proper subexpressions of an expression? The semantics of a language attempts to give the language an interpretation and to supply a meaning or value to the expressions, programs etc. in the language."

In the sequel we will study different aspects of programming languages as well as different formalisms for defining them.

Chapter 2

Syntax: Grammars and Parsing

Now let us return to the concrete syntax of a programming language. Syntactically a language is nothing but an (infinite) set of strings over an alphabet. The traditional way of describing such a set is a *grammar* (Chomsky grammar, generative grammar). A grammar is a four tuple $\langle V_T, V_N, S, P\rangle$ where

- V_T is a finite set of terminal symbols. This set is sometimes called the *alphabet* of the language. We use the variables a and b to range over terminal symbols.
- V_N is a finite set of nonterminal symbols. We use A and B to range over nonterminal symbols.
- $S \in V_N$ is the startsymbol of the grammar.
- P is a finite set of productions.

In the most general case, a production is of the form

$$\alpha^+ \to \beta^*$$

where α and β ranges over $V = V_T \cup V_N$. One classification of grammars is according to the complexity of the productions in the grammar. This classification is called the Chomsky hierarchy.

- type 3: In a *regular grammar* all productions are of the forms $A \to aB$ and $A \to a$ (or of the forms $A \to Ba$ and $A \to a$).
- type 2: In a *context free grammar* the productions are of the form $A \to \alpha_1\alpha_2\cdots\alpha_n$ where $\alpha_i \in V$.
- type 1: The productions in a *context sensitive grammar* have left hand sides which are shorter than the right hand sides.
- type 0: There are no restrictions on the productions of a type 0 grammar.

A remark which is often heard is that the class of context-free languages is very important for computer science. This is a little bit strange since:

- Our "practical" programming languages are never as expressive as the context free languages.

- It is considered important that it should be possible to construct a parser for a programming language that parses an input string in time proportional to the length of the string. This is not possible for all context free languages.
- Context-free grammars are not powerful enough to express the type system of a traditional programming language

Therefore we could go in two different directions from the context free languages. We could either consider

- more restricted grammars, i.e. grammars which could be efficiently implemented – LL(k) or LR(k) grammars, or
- more expressive grammars which could express the type systems in programming languages – attribute grammars, two level grammars or DCG grammars.

We will in these lecture notes concentrate on the latter since these grammar formalisms are generally less known.

2.1 Grammars

Given a grammar[1] such as

$$\begin{array}{lcl} E & ::= & E\texttt{+}E \\ & | & E\texttt{*}E \\ & | & \texttt{x} \end{array}$$

What language does it define? In order to express this formally, we must first introduce the notions of *derivation* and *parse tree.*

Definition 1 *A* **derivation** *is a sequence of sentential forms (strings of terminals and nonterminals)*

$$S = \alpha_1 \rightarrow \alpha_2 \rightarrow \cdots \rightarrow \alpha_n$$

such that α_{i+1} is equal to α_i except for one occurrence of the left hand side of a production which is replaced by the right hand side of the same production.

Example: Given the grammar

$$\begin{array}{lcl} E & ::= & E\texttt{+}E \\ & | & E\texttt{*}E \\ & | & \texttt{x} \end{array}$$

[1] A grammar is formally, as we previously said, a four tuple but we often just give the productions explicitly and use notational conventions to get the other parts:

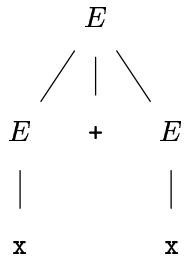
- the terminal symbols are written in typewriter font (`x`, `if`, `while`, ...),
- the nonterminal symbols are either upper case letters written in italic style (E, Cmd, ...) or a word enclosed in brackets (⟨ident⟩, ...), and
- the startsymbol is often obvious, for example if there is only one nonterminal or because of its name.

we will present two examples of derivations (we have indicated the nonterminal that has been expanded by a hat ($\hat{}$).

$$\hat{E} \rightarrow \hat{E}\texttt{+}E \rightarrow \texttt{x+}\hat{E} \rightarrow \texttt{x+x}$$
$$\hat{E} \rightarrow E\texttt{+}\hat{E} \rightarrow \hat{E}\texttt{+x} \rightarrow \texttt{x+x}$$

□

In a *rightmost derivation* it is always the rightmost nonterminal which is replaced and in a *leftmost derivation* it is the leftmost nonterminal. The first example above is a rightmost and the second is a leftmost derivation. Normally it is not important to know the order in which the nonterminals are expanded, so we present the derivation as a *parse tree*.



A parse tree for `x+x`

The *parsing problem*[2] can be defined as the problem to

1. decide if a given string belongs to a language or not, and
2. if the string belongs to the language, give evidence for this, for example by providing a derivation or a parse tree.

[2]Compare the parsing problem with the problem of theorem proving:

> Given a formal theory $\mathcal{T}$ and a proposition $\mathcal{P}$ then decide if $\mathcal{P}$ is provable in $\mathcal{T}$ or not and if it is provable give a derivation or a proof tree.

It is easy to see that a grammar could be viewed as a very simple logical theory. One could compare how a string is derived from the startsymbol using the productions with how a theorem is derived from the axiom by using the inference rules.

Compare also how the notion of derivation in parsing is introduced with how the notion of reduction is introduced in λ calculus. Barendregt[8] introduces reduction in λ-calculus as follows:

1. introduce a notion of reduction, R,
2. it generates a one step computation relation, $\rightarrow$,
3. which generates a many step computation relation, $\rightarrow^*$,
4. which generates an equality, $=$.

For grammars we introduce the following notions

1. a set of productions (compare it with the notion of reduction),
2. a one step derivation relation,
3. a many step derivation relation.

Given a grammar G it defines the language $\mathcal{L}(G)$ over the alphabet Σ such that

$$\mathcal{L}(G) = \{a \in \Sigma^* \mid S \rightarrow^* a\}$$

where $\rightarrow^*$ is the transitive closure of $\rightarrow$.

We often use the same classification of languages as we have already introduced for grammars: We say that a language is of type X if it can be described by a grammar of type X and not of a "simpler" grammar.

An important notion for languages and grammars is the notion of *ambiguity*. We have the following definition

Definition 2 *A grammar is* **ambiguous** *if there is a string in the generated language which have two different parse trees.*

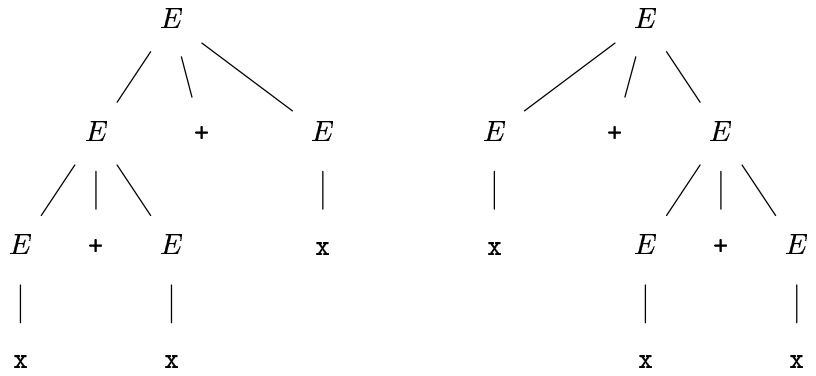
Example: Consider the grammar

$$\begin{array}{lcl} E & ::= & E\texttt{+}E \\ & | & \texttt{x} \end{array}$$

The string `x+x+x` could be generated by the derivations

$$E \rightarrow \hat{E}\texttt{+}E \rightarrow \texttt{x+}\hat{E} \rightarrow \texttt{x+}E\texttt{+}E \rightarrow^2 \texttt{x+x+x}$$
$$E \rightarrow \hat{E}\texttt{+}E \rightarrow E\texttt{+}E\texttt{+}E \rightarrow \texttt{x+}E\texttt{+}E \rightarrow^2 \texttt{x+x+x}$$

where the nonterminal which is expanded is decorated with a hat. The two derivations correspond to the two parse trees



The two parse trees for `x+x+x`

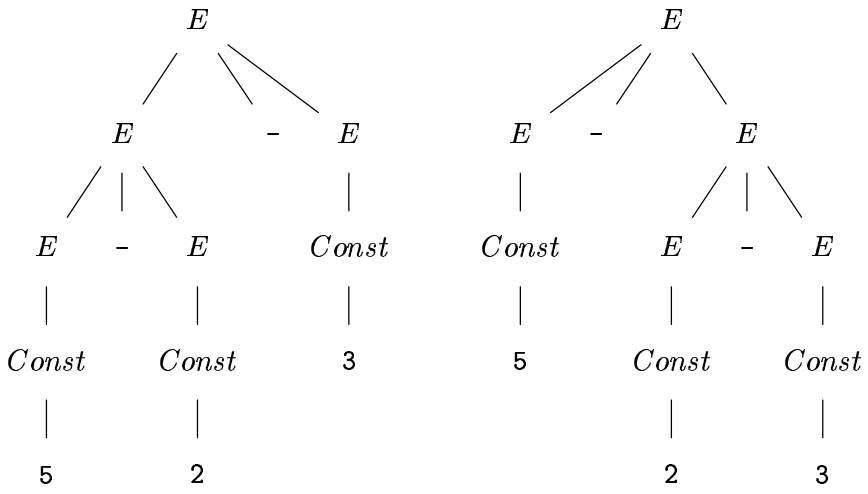
Since there are two different parse trees for the same string, the grammar is ambiguous. □

Ambiguous grammars can be dangerous. When we define the semantics of a language described by an ambiguous grammar, we could accidently give two different meanings to an ambiguous string. The reason is that we almost always define the meaning in terms of the structure (syntax tree, parse tree) of the string.

Example: Consider the grammar

$$
\begin{array}{rcl}
E & ::= & E - E \\
 & | & Const \\
Const & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \cdots
\end{array}
$$

and the string `5-2-3`. The string has two different parse trees:



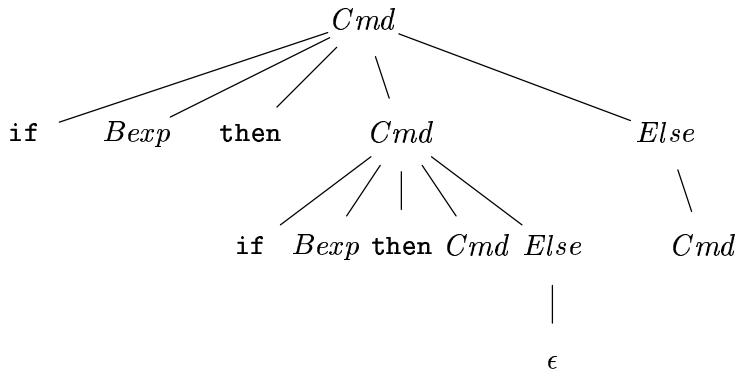
The two parse trees for `5-2-3`

In linear notation we often use parenthesis to describe the structure. The left parse tree corresponds to the string `(5-2)-3` and the right parse tree corresponds to `5-(2-3)`. In the standard interpretation of integers and the minus sign these two expressions clearly have different meaning. □

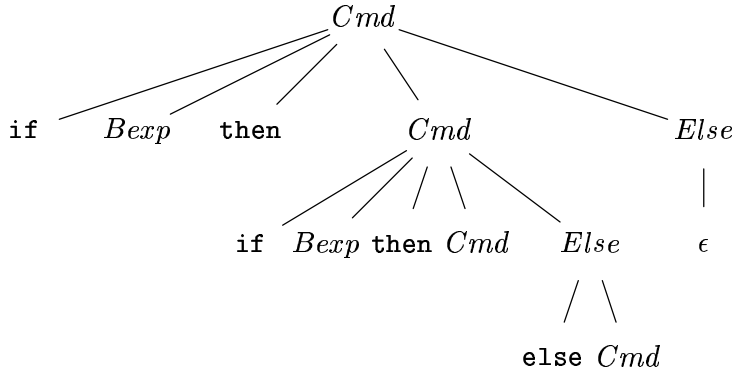
Example: A famous ambiguity is from one of the first definitions of ALGOL60 where if-statements were defined using a grammar of the form

$$
\begin{array}{rcl}
Cmd & ::= & \texttt{if}\ Bexp\ \texttt{then}\ Cmd\ Else \\
Else & ::= & \texttt{else}\ Cmd \\
 & | & \epsilon
\end{array}
$$

where ϵ denotes the empty string. Using this grammar the statement `if B then C if B then C else D` could be derived in two different ways. Intuitively, the question is to which if-statement the else part belongs. The two different parse trees are:



The first parse tree for `if` *Bexp* `then if` *Bexp* `then` *Cmd* `else` *Cmd*



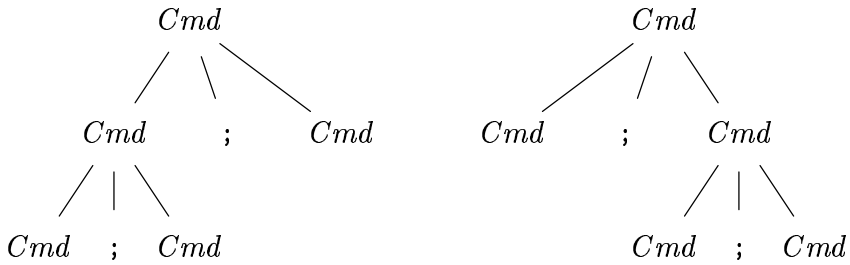
The second parse tree for `if` *Bexp* `then if` *Bexp* `then` *Cmd* `else` *Cmd*

It is clear that the semantics of the string is dependent on which parse trees one chooses as the "correct" one. □

Example: As a final example of ambiguity in programming languages consider the associativity of command sequencing. Given the grammar:

$$Cmd \quad ::= \quad \cdots \quad | \quad Cmd \ ; \ Cmd$$

it is easy to see that it is ambiguous since there are two different parse trees for $Cmd;Cmd;Cmd$:



The two parse trees for Cmd ; Cmd ; Cmd

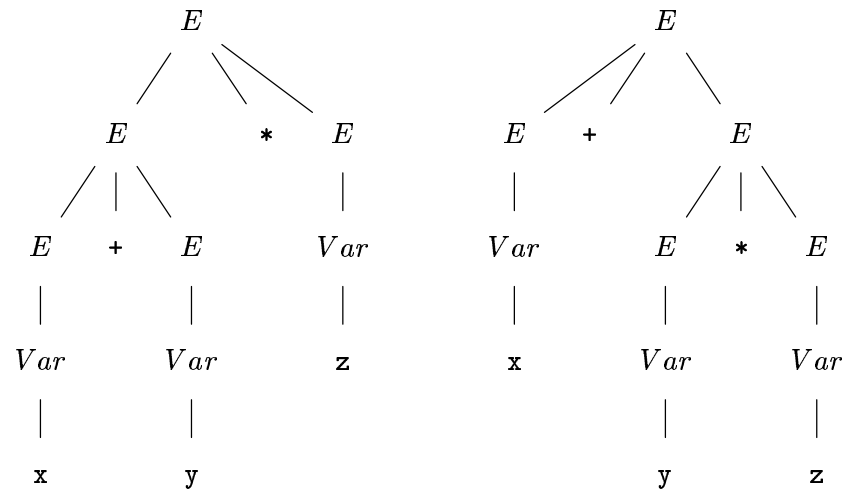
However, in most programming languages this ambiguity is not dangerous since a compound statement `c1;c2;c3` usually has the same meaning regardless of how it it parsed. The same, of course, holds for all associative operators such as + and $*$.[3] □

There are examples when we want to use other mechanisms than grammars to define languages. In some situations the grammar gets very complicated although the language is quite simple. Sometimes it is therefore better first to define a language by an ambiguous grammar and then add some mechanism that resolves the ambiguity. One example of this method is when we define arithmetic expressions by a grammar such as:

$$\begin{aligned} E \;::=\; & Const \mid Var \mid \texttt{-}E \mid E\texttt{+}E \\ \mid\; & E\texttt{-}E \mid E\texttt{*}E \mid E\texttt{/}E \mid \texttt{(}E\texttt{)} \end{aligned}$$

Here we do not take any consideration to the binding power of the operators. Afterwards we add a notion of precedence to the grammar that selects one of the different parse trees for a sentence as the correct one.

The string `x+y*z` has the two parse trees:



The two parse trees for `x+y*z`

and the disambiguiting rules says that `*` binds harder than `+` so the first tree is the correct one.

Introducing precedences for grammars with one nonterminal and only infix operators is easy; to extend the notion also to grammars with prefix and postfix operators and several nonterminals is however quite complicated [1, 2].

Exercise 1 *Can you give an argument why type systems cannot be defined using context free grammars. Does this argument include all kinds of type systems?*

Exercise 2 *Why is the ambiguity with the dangling else more serious than the ambiguity with the sequencing operator ';'?*

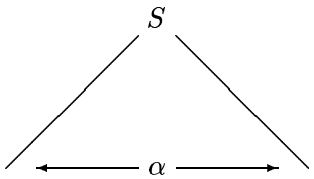
[3] Notice that even if the "mathematical" version of an operator is associative, its implementation in computers could be nonassociative due to things like overflow.

2.2 Parsing

The problem of parsing a string α given a grammar G is to construct a derivation of the string from the startsymbol of the grammar:

$$S \rightarrow \quad \cdots \quad \rightarrow \alpha$$

or equivalently, to construct the corresponding parse tree



In order to do this we can proceed in two ways, either we build the derivation from left (the parse tree is built top-down) or we build it from right (the parse tree is built bottom-up). One example of the first kind of parsing is called LL-parsing (Left-Left, we read the string from the Left and builds a Leftmost derivation), or top-down parsing, since we build the parse tree from the top (that is the root of the tree!!) to the bottom. In a grammar there are often many alternatives to expand a nonterminal (for example the startsymbol) but if it is always possible to determine what alternative to use from the initial part of the input string, then it is possible to construct an LL-parser.

Another parsing technique is to build the tree bottom-up, where we try to match a portion of the input with every right hand side of the productions and then reduce the one that eventually will lead to a successful parsing of the complete string. An important class of parsers which work bottom-up is called LR parsers [23] (the input string is read from Left and the parser builds a Rightmost derivation). Neither LL nor LR-parsers are capable of parsing all context free languages.

For more information about languages and parsing techniques see the books by Aho et al [4], Backhouse [7] and Hopcroft and Ullman [18].

Exercise 3 *Give an informal description of a top-down and a bottom-up parsing of the string* `((x+x)*x)` *using the grammar:*

$$\begin{array}{rcl} E & ::= & \texttt{(}E\texttt{+}E\texttt{)} \\ & | & \texttt{(}E\texttt{*}E\texttt{)} \\ & | & \texttt{x} \end{array}$$

2.2.1 Earley's parsing algorithm

There are many different algorithms for the parsing problem. One interesting such algorithm is Earley's algorithm [15]. It is interesting because it is an algorithm which can cope with all context free languages. It is seldom used in practice for programming languages but has, at least, a theoretical interest.

To every *position*, n in the input string we associate a *set of items*. An item, $\langle P, i \rangle$ consists of a dotted production, P and a position, i, in the input string (which records

the position where the parsing of the production started). The intuition is that an item represents a *parsing possibility* starting from position i in the input string. It is a parsing possibility because the input string between position i and n has been parsed according to the part from the beginning to the dot in the production. In order to be a complete parsing described by the production, the dot must be at the end of the production.

We will describe the Earley's parsing algorithm by an example. The example is very simple but it covers the essential operations of the algorithm.

Example: Consider the grammar

$$\begin{array}{rcl} E & ::= & (E\texttt{+}E) \\ & | & (E\texttt{*}E) \\ & | & \texttt{1} \\ & | & \texttt{2} \end{array}$$

and the input string

input string:	(	1	+	2	)
positions:	1	2	3	4	5

We start the parsing from left by creating the items for position 1. The item set is constructed by starting with the item that consists of the dotted production $\Theta \rightarrow \bullet E\dashv$[4] and the input position 1. We write this item as

Pos 1: $\Theta \rightarrow \bullet E\dashv$, 1

It means that we expect an E (the startsymbol) starting from position 1. We add all possibilities to parse an E to the set of items for position 1. This is called the *predict operation* in the algorithm. So we add the following items:

Pos 1: $E \rightarrow \bullet(E\texttt{+}E)$, 1
$E \rightarrow \bullet(E\texttt{*}E)$, 1
$E \rightarrow \bullet\texttt{1}$, 1
$E \rightarrow \bullet\texttt{2}$, 1

Since the dot now is not in front of any nonterminal we have exhausted all possible parsings from position 1 of the input string. Our next step is to compare the terminal symbol in position 1 of the input string with the terminals after the dots in the items and select those in which the terminals match. This operation is called the *scanner operation* in the parsing algorithm. We get the following parsing possibilities for position 2 in the input string.

Pos 2: $E \rightarrow (\bullet E\texttt{+}E)$, 1
$E \rightarrow (\bullet E\texttt{*}E)$, 1

Since the dot now is in front of a nonterminal in the items, we use the predict operation to get more parsing possibilities for position 2. Because these new items are introduced when we consider position 2 in the input string, the position indicator of the items must be equal to 2.

[4] Θ is a new startsymbol and $\dashv$ means "end of string"

Pos 2: $E \to \bullet\texttt{(}E\texttt{+}E\texttt{)}$, 2
$E \to \bullet\texttt{(}E\texttt{*}E\texttt{)}$, 2
$E \to \bullet\texttt{1}$, 2
$E \to \bullet\texttt{2}$, 2

Now we can start to construct the parsing possibilities for position 3. We compare the terminal symbol after the dot in the items above with the symbol in position 3 in the input string and get:

Pos 3: $E \to \texttt{1}\bullet$, 2

The dot in this item is now positioned at the end of the production. This means that we have completed the parsing of an E starting from position 2. Since we for position 2 had the following two parsing possibilities (items):

Pos 2: $E \to \texttt{(}\bullet E\texttt{+}E\texttt{)}$, 1
$E \to \texttt{(}\bullet E\texttt{*}E\texttt{)}$, 1

and we have parsed an E, we can move the dot over the E and add the two parsing possibilities:

Pos 3: $E \to \texttt{(}E\bullet\texttt{+}E\texttt{)}$, 1
$E \to \texttt{(}E\bullet\texttt{*}E\texttt{)}$, 1

for position 3. This parsing operation is called the *completer operation.*

Continuing with position 4, we get, by using the scanner operation and the '+' symbol in position 3 of the input string, the following item:

Pos 4: $E \to \texttt{(}E\texttt{+}\bullet E\texttt{)}$, 1

The predict operation generates the usual four items:

Pos 4: $E \to \bullet\texttt{(}E\texttt{+}E\texttt{)}$, 4
$E \to \bullet\texttt{(}E\texttt{*}E\texttt{)}$, 4
$E \to \bullet\texttt{1}$, 4
$E \to \bullet\texttt{2}$, 4

Moving on to position 5, we get, by the scanner operation and the symbol '2' in the input string:

Pos 5: $E \to \texttt{2}\bullet$, 4

which, by the completer operation, generates the item

Pos 5: $E \to \texttt{(}E\texttt{+}E\bullet\texttt{)}$, 1

By using the scanner operation on this item, we get

Pos 6: $E \to \texttt{(}E\texttt{+}E\texttt{)}\bullet$, 1

for position 6 and by the completer operation

Pos 1: $\Theta \to E\bullet\dashv$, 1

By scanning the $\dashv$ symbol we have finally completed the parsing. □

Exercise 4 *Parse the strings* `1` *and* `((1+1)*1)` *using Earley's algorithm and the same grammar as in the previous exercise.*

Exercise 5 *Since we construct a* **set of** *items for every position in the input string, the predict operation can only introduce one instance of each production. This means that there is no unbounded recursion in, for example, the grammar*

$$\begin{array}{lcl} E & ::= & E \texttt{ + } E \\ & | & \texttt{1} \end{array}$$

Show how the string `1+1+1` *is parsed by Earley's algorithm.*

2.3 Abstract Syntax

When we define the semantics of a programming language we assign meaning to every syntactically correct program. Since there are infinitely many programs and we want to define the semantics in a "natural" way, we have to assign the meaning in a structural way. We therefore want a syntactical description that emphasizes the structure of the language rather than the actual strings of the language. John McCarthy introduced the notion of *abstract syntax* already in 1962 [28]. He wrote:

> "The Backus normal form that is used in the ALGOL report, describes the morphology of ALGOL programs in a *synthetic* manner. Namely, it describes how various kinds of programs are built up from their parts. This would be better for translating into ALGOL than it is for the more usual problem of translating from ALGOL. The form of syntax we should describe differs from the Backus normal form in two ways. First it is analytic rather than synthetic, it tells how to take programs apart rather than how to put them together. Secondly it is abstract in the way that it is independent of the notation used to represent, say sums, but only affirms that they can be recognized and taken apart."

Let us take an example. Take the following concrete grammar:

$$\begin{array}{lcl} \langle\text{expr}\rangle & ::= & \langle\text{term}\rangle \\ & | & \langle\text{expr}\rangle\texttt{+}\langle\text{term}\rangle \\ \langle\text{term}\rangle & ::= & \langle\text{factor}\rangle \\ & | & \langle\text{term}\rangle\texttt{*}\langle\text{factor}\rangle \\ \langle\text{factor}\rangle & ::= & \langle\text{intconst}\rangle \\ & | & \langle\text{variable}\rangle \\ & | & \texttt{(}\langle\text{expr}\rangle\texttt{)} \end{array}$$

Instead of this grammar, McCarthy wants:

1. Boolean functions that says what form an expression has:

isconst(e)	isvar(e)
issum(e)	isprod(e)

2. If two terms are constants or variables it must be possible to decide if they are the same. We must therefore have Boolean functions

   $\text{eqVar}(e_1, e_2)$ $\text{eqConst}(e_1, e_2)$

   that computes if two constants or variables are equal.

3. If an expression is a sum or a product it must be possible to select the parts. We need functions

   $\text{destsum1}(e)$ $\text{destsum2}(e)$
   $\text{destprod1}(e)$ $\text{destprod2}(e)$

We can summarize McCarthy's view by saying that we want an abstract datatype for programs. One way of defining this datatype is:

```
Expr =
  DATATYPE
     OPERATIONS
        mkconst(c)    :  Expr    if c :  Int
        mkvar(x)      :  Expr    if x :  Var
        mksum(e1,e2)  :  Expr    if e1,e2 :  Expr
        mkprod(e1,e2) :  Expr    if e1,e2 :  Expr
        isconst(e)    :  Bool    if e :  Expr
        isvar(e)      :  Bool    if e :  Expr
        issum(e)      :  Bool    if e :  Expr
        isprod(e)     :  Bool    if e :  Expr
        destconst(e)  :  Int     if e :  Expr
        destvar(e)    :  Var     if e :  Expr
        destsum1(e)   :  Expr    if e :  Expr
        destsum2(e)   :  Expr    if e :  Expr
        destprod1(e)  :  Expr    if e :  Expr
        destprod2(e)  :  Expr    if e :  Expr
     EQUATIONS
        isconst(mkconst(c))   = true
        isconst(mkvar(x))     = false
                    ⋮
        destconst(mkconst(c)) = c
        destvar(mkvar(x))     = x
        destsum1(mksum(e1,e2) = e1
                    ⋮
  END
```

The operations `mkconst`, `mkvar`, `mksum` and `mkprod` are called *constructors*; `isconst`, `isvar`, `issum` and `isprod` are called *predicates* and `destconst`, `destvar`, `destsum1`, `destsum2`, `destprod1` and `destprod2` are called *selectors*.

Datatypes like this one, with no equations between the constructors are called *free* or *inductive* data types.

As we have seen it becomes a little bit awkward to follow McCarthy and define selectors and predicates. So instead of defining a function using predicates and selectors such as

```
nrofvars(e) = if isconst(e) then 0
              if isvar(e)   then 1
              if issum(e)   then
                            nrofvars(destsum1(e))
                            + nrofvars(destsum2(e))
              if isprod(e) then
                            nrofvars(destprod1(e))
                            + nrofvars(destprod2(e))
              else failure
```

we define the abstract syntax in the following way

$e \ ::= \ c \mid v \mid e\texttt{+}e \mid e\texttt{*}e$

where $e \in$ Expr, $c \in$ Int, and $v \in$ Var

and we use pattern matching to define functions, as in the following definition:

```
nrofvars(c) = 0
nrofvars(v) = 1
nrofvars(e1+e2) = nrofvars(e1) + nrofvars(e2)
nrofvars(e1*e2) = nrofvars(e1) + nrofvars(e2)
```

It is important that we are a bit careful when using pattern matching, all patterns must occur and if two patterns are overlapping (they could match the same value) the function must give the same result.

The programming language ML has gone through the same form of development as we have indicated here — from functions for predicates and selectors in the first version [17] to pattern matching in recent versions [29]. In ML the datatype for expressions is defined by

```
datatype Expr
  = const of Int
  | var   of Var
  | plus  of Expr + Expr
  | mult  of Expr * Expr
```

and we could define the function as

```
fun nrofvars(const(_)) = 0
|   nrofvars(var(_)) = 1
|   nrofvars(plus(e1,e2)) = nrofvars(e1) + nrofvars(e2)
|   nrofvars(mult(e1,e2)) = nrofvars(e1) + nrofvars(e2)
```

In LML [6, 5] one could do exactly the same thing with minor syntactical changes. With "conctypes" [3] it is even possible to use the concrete syntax when representing expressions and defining functions

```
conctype Expr
  = [|<Const>|]
  | [|<Var>|]
  | [|<Expr>+<Expr>|]
  | [|<Expr>*<Expr>|]

fun nrofvars[|^(c:Const)|] = 0
|   nrofvars[|^(v:Var)|]   = 1
|   nrofvars[|^e1+^e2|]    = nrofvars(e1) + nrofvars(e2)
|   nrofvars[|^e1*^e2|]    = nrofvars(e1) + nrofvars(e2)
```

Exercise 6 *Given the following grammar:*

⟨program⟩ ::= `program(`⟨var⟩`)` ⟨cmd⟩ `result(`⟨var⟩`)`
⟨cmd⟩ ::= ⟨block⟩
| ⟨assignment⟩
| ⟨ifcmd⟩
| ⟨whilecmd⟩
| ⟨skipcmd⟩
⟨block⟩ ::= `begin` ⟨cmdseq⟩ `end`
⟨cmdseq⟩ ::= ⟨cmd⟩
| ⟨cmd⟩`;`⟨cmdseq⟩
⟨assignment⟩ ::= ⟨name⟩`:=`⟨expr⟩
⟨ifcmd⟩ ::= `if` ⟨bexpr⟩
`then` ⟨cmd⟩
`else` ⟨cmd⟩
⟨whilecmd⟩ ::= `while` ⟨bexpr⟩ `do`
⟨cmd⟩
⟨skipcmd⟩ ::= `skip`
⟨bexp⟩ ::= ⟨expr⟩⟨relop⟩⟨expr⟩
| `not` ⟨bexpr⟩
⟨relop⟩ ::= `<` | `<=` | `=` | `/=` | `>=` | `>`
⟨expr⟩ ::= ⟨expr⟩⟨addop⟩⟨term⟩
⟨addop⟩ ::= `+` | `-`
⟨term⟩ ::= ⟨term⟩⟨mulop⟩⟨factor⟩
⟨mulop⟩ ::= `*` | `/`
⟨factor⟩ ::= ⟨var⟩
| ⟨unsigned number⟩
| `-` ⟨factor⟩
| `(`⟨expr⟩`)`

Construct a suitable abstract syntax. Define a function that translates programs in this abstract syntax into Pascal programs.

Chapter 3

Types: Description Techniques and Type Checking

Let us look at some extensions of the Chomsky grammar formalism that are strong enough to express the typing rules of monomorphic programming languages such as Pascal and Ada. We assume that we have a language with only a finite number of types, for example the language,

⟨expr⟩ ::= ⟨intconst⟩
| ⟨boolconst⟩
| ⟨expr⟩ `&` ⟨expr⟩
| ⟨expr⟩ `+` ⟨expr⟩
| ⟨expr⟩ `=` ⟨expr⟩
| `if` ⟨expr⟩ `then` ⟨expr⟩ `else` ⟨expr⟩

The type system of this simple language is easy to express using a context free grammar (BNF), by dividing the syntactical category of expressions into one for integer expressions, ⟨iexpr⟩, and one for boolean expressions, ⟨bexpr⟩.

⟨iexpr⟩ ::= ⟨intconst⟩
| ⟨iexpr⟩ `+` ⟨iexpr⟩
| `if` ⟨bexpr⟩ `then` ⟨iexpr⟩ `else` ⟨iexpr⟩

⟨bexpr⟩ ::= ⟨boolconst⟩
| ⟨bexpr⟩ `&` ⟨bexpr⟩
| ⟨iexpr⟩ `=` ⟨iexpr⟩
| ⟨bexpr⟩ `=` ⟨bexpr⟩
| `if` ⟨bexpr⟩ `then` ⟨bexpr⟩ `else` ⟨bexpr⟩

When the language ALGOL-W ([44]) was defined, the authors (Hoare and Wirth) wanted to define the type system in this way. However, they soon realized that the definition became too big and therefore very complicated to understand. To remedy this problem they invented the notion of *schematic nonterminals*. This notion had also been suggested by van Wijngaarden. Instead of the "constant" nonterminals used in the grammar above, they also used nonterminals with variables as in the following grammar:

$$\begin{array}{lcl} \langle X \text{ expr}\rangle & ::= & \langle X \text{ const}\rangle \\ \langle X_0 \text{ expr}\rangle & | & \langle X_1 \text{ expr}\rangle \texttt{ + } \langle X_2 \text{ expr}\rangle \end{array}$$

where $X \in \{\text{integer}, \text{real}, \text{complex}\}$ and X_0 could be computed from X_1 and X_2 by using the following table

		$X_2 =$		
		integer	real	complex
$X_1 =$	integer	integer	real	complex
	real	real	real	complex
	complex	complex	complex	complex

By using this *notation* we could define our language in the following way:

$$\begin{array}{lcl} \langle \text{expr}(X)\rangle & ::= & \langle \text{const}(X)\rangle \\ & | & \texttt{if } \langle \text{expr(bool)}\rangle \texttt{ then } \langle \text{expr}(X)\rangle \texttt{ else } \langle \text{expr}(X)\rangle \\ \langle \text{expr(int)}\rangle & ::= & \langle \text{expr(int)}\rangle \texttt{ + } \langle \text{expr(int)}\rangle \\ \langle \text{expr(bool)}\rangle & ::= & \langle \text{expr(bool)}\rangle \texttt{ \& } \langle \text{expr(bool)}\rangle \\ & | & \langle \text{expr}(X)\rangle \texttt{ = } \langle \text{expr}(X)\rangle \end{array}$$

In the notation we should perhaps indicate where variables are bound and what they range over. By using a more explicit notation, we could write the first production as follows:

$$\begin{array}{l} (\forall X \in \{\text{int}, \text{bool}\}) \\ \qquad \langle \text{expr}(X)\rangle \quad ::= \quad \langle \text{const}(X)\rangle \end{array}$$

$$\begin{array}{l} (\forall X \in \{\text{int}, \text{bool}\}) \\ \qquad \langle \text{expr}(X)\rangle \quad ::= \quad \texttt{if } \langle \text{expr(bool)}\rangle \texttt{ then } \langle \text{expr}(X)\rangle \texttt{ else } \langle \text{expr}(X)\rangle \end{array}$$

When the number of elements in the sets the variables range over is finite, this could be seen as a simplified *notation* for a usual context-free grammar. If, however, we want to use the same notation for a language with an infinite set of types, then we go beyond the expressiveness of BNF grammars and the context-free languages. Since most languages contain infinitely many types the context-free languages is too weak to define type systems. Every language with a type constructor that takes a type into a type, such as `array`, `pointer to`, `set of`, `->`, `record`, `struct`, ..., contains infinitely many types.

3.1 Two level grammars

When defining ALGOL68, the authors had the ambition to express *all* syntactic restrictions in the definition of the language. They therefore introduced a new grammar formalism called *two level grammars* (or van Wijngaarden grammars[1]) [11, 42]. Seen as a formalism to define sets, two level grammars are very powerful. All type-0 languages, i.e. all recursive enumerable sets can be defined using two level grammars. This indicates that the formalism is perhaps a bit too powerful, since we could define languages which are impossible to analyze in a mechanical way.

A two level grammar is a context-free grammar in which the nonterminals consists of strings of characters, variables and spaces. The nonterminals may therefore be infinitely

[1] sometimes abbreviated vW-grammars and therefore also called VolksWagen-grammars!!

many and are generated by a metagrammar. Our previous example can be defined by the following two level grammar:

metagrammar:

X ::= bool | int

(object)grammar:

⟨expr of type X⟩ ::= ⟨const of type X⟩
| **if** ⟨expr of type bool⟩
then ⟨expr of type X⟩
else ⟨expr of type X⟩

The *production schemas (hyper rules)* in this grammar stands for all productions which could be obtained by substituting the strings generated by the metagrammar for the variable X. Since the variables are universally quantified in each production, substitution means that *all* occurances of a variable in a production must be replaced by the same string in the substitution operation.

Example: A bigger example perhaps shows the expressiveness of two level grammars. We shall define the language

$$\mathcal{L}(\mathcal{T}) = \{\alpha^n\beta^n\alpha^n \mid n \geq 1\}$$

which is a well-known example of a language which is not context-free.

metagrammar:

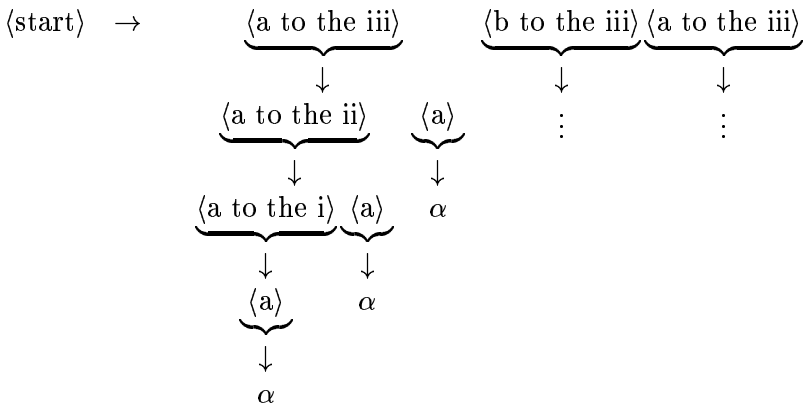
N ::= i | Ni
X ::= a | b

N is essentially a coding of the natural numbers ($3 = iii$), or, perhaps in better words, N is used as a variable ranging over the natural numbers.

grammar:

⟨start⟩ ::= ⟨a to the N⟩⟨b to the N⟩ ⟨a to the N⟩
⟨X to the Ni⟩ ::= ⟨X to the N⟩⟨X⟩
⟨X to the i⟩ ::= ⟨X⟩
⟨a⟩ ::= α
⟨b⟩ ::= β

We can derive $\alpha\alpha\alpha\beta\beta\beta\alpha\alpha\alpha$ by the following derivation:



□

Exercise 7 *Define a meta grammar for natural numbers with the standard decimal notation: 1, 2, 3, ...*

More formally a two level grammar consists of the following components:

Metasystem

- $\mathcal{M}$ a finite set of *metavariables (metanotions)* – syntactic variables (nonterminals) in the metagrammar.
- $\mathcal{T}_0$ a finite set of *metaterminals.*
- $\Pi \subseteq \mathcal{M} \times (\mathcal{M} \cup \mathcal{T}_0)^*$ – context-free productions in the metagrammar
- $\mathcal{L}_X = \{u \in \mathcal{T}_0^* \mid X \rightarrow_\Pi^* u\}$ – the language generated by the metanonterminal $X \in \mathcal{M}$. We use the set of productions as an index to the rightarrow in order to clarify which productions could be used in the derivation.

Objectsystem

- $\mathcal{T} \supseteq \mathcal{T}_0$ – the alphabet that builds up the nonterminals
- Σ a finite set of terminal symbols.
- $\overline{N} \subseteq \{\langle \overline{u} \rangle \mid \overline{u} \in (\mathcal{M} \cup \mathcal{T})^+ \}$ – nonterminal schemas
- $\overline{P} \subseteq \overline{N} \times (\overline{N} \cup \Sigma)^*$ – the production schemas
- $\mathcal{L} \subseteq \Sigma^*$ – the language generated by the object system in the two-level grammar.
- $S \in \overline{N}$ – the startsymbol

In our example we had:

Metasystem

$$
\begin{aligned}
M &= \{N, X\} \\
T_0 &= \{\mathrm{i}, \mathrm{a}, \mathrm{b}\} \\
\Pi &= \{N ::= \mathrm{i}, N ::= N\mathrm{i}, X ::= \mathrm{a}, X ::= \mathrm{b}\} \\
\mathcal{L}_\mathrm{N} &= \{\mathrm{i}, \mathrm{ii}, \mathrm{iii}, \ldots\} = \{\mathrm{i}\}^+ \\
\mathcal{L}_\mathrm{X} &= \{\mathrm{a}, \mathrm{b}\}
\end{aligned}
$$

Objectsystem

$$
\begin{aligned}
T &= \{\mathrm{to}, \mathrm{the}, \ldots\} \\
\Sigma &= \{\alpha, \beta\} \\
\overline{N} &= \{\langle \mathrm{a\ to\ the\ } N\rangle, \langle X \mathrm{\ to\ the\ i}\rangle, \ldots\} \\
\overline{P} &= \{\langle \mathrm{start}\rangle ::= \langle \mathrm{a\ to\ the\ } N\rangle\langle \mathrm{b\ to\ the\ } N\rangle\langle \mathrm{a\ to\ the\ } N\rangle, \\
&\quad \langle X \mathrm{\ to\ the\ } N\mathrm{i}\rangle ::= \langle X \mathrm{\ to\ the\ } N\rangle\langle X\rangle, \ldots\} \\
\mathcal{L} &= \{\alpha\beta\alpha, \alpha\alpha\beta\beta\alpha\alpha, \alpha\alpha\alpha\beta\beta\beta\alpha\alpha\alpha, \ldots\} \\
S &= \langle \mathrm{start}\rangle
\end{aligned}
$$

The productions of the two level grammar are obtained by substituting all phrases of the language $\mathcal{L}_{\mathcal{X}}$ for each metavariable X in the nonterminal schemas. In our example we get the following infinite set of productions:

$$
\begin{aligned}
P \;=\; \{&\langle\text{start}\rangle \to \langle\text{a to the i}\rangle\langle\text{b to the i}\rangle\langle\text{a to the i}\rangle,\\
&\langle\text{start}\rangle \to \langle\text{a to the ii}\rangle\langle\text{b to the ii}\rangle\langle\text{a to the ii}\rangle,\\
&\langle\text{start}\rangle \to \langle\text{a to the iii}\rangle\langle\text{b to the iii}\rangle\langle\text{a to the iii}\rangle,\\
&\qquad\vdots\\
&\langle\text{a to the i}\rangle \to \langle\text{a}\rangle,\\
&\langle\text{b to the i}\rangle \to \langle\text{b}\rangle,\\
&\langle\text{a to the ii}\rangle \to \langle\text{a to the i}\rangle\langle\text{a}\rangle,\\
&\langle\text{b to the ii}\rangle \to \langle\text{b to the i}\rangle\langle\text{b}\rangle,\\
&\qquad\vdots\\
&\langle\text{a}\rangle \to \alpha,\\
&\langle\text{b}\rangle \to \beta\\
\}&
\end{aligned}
$$

A two level grammar W defines a language $\mathcal{L}(\mathcal{W})$ as usual:

$$\mathcal{L}(\mathcal{W}) = \{\sigma\in\Sigma^* \mid \langle\text{start}\rangle \to^* \sigma\}$$

The only difference from context-free languages is that the set of productions could be infinite and that this set is presented by using schemes and variables. But this is irrelevant for the definition of the notions: one-step derivation and derivation. Furthermore since the startsymbol in the two level grammar may be a schema, a two level grammar may generate a family of languages instead of just one language. Another way to see a two level grammar is to say that the grammar defines a language (set) which is the union of the family. A string is in the language if there is a startsymbol and a derivation which generates the string.

Example: Let us describe another example of a language defined by a two level grammar:

objectgrammar

⟨bitstring of length with value⟩
::= ϵ

⟨bitstring of length N_1i with value N_2N_2⟩
::= ⟨bitstring of length N_1 with N_2 value⟩

⟨bitstring of length N_1i with value N_2N_2i⟩
::= ⟨bitstring of length N_1 with N_2 value⟩ 1

Where the metavariables are defined by the following metagrammar:

metagrammar

N ::= N i | ϵ
N_1 ::= N
N_2 ::= N

Furthermore let all nonterminals of the form

⟨bitstring of length N_1 with value N_2⟩

be startsymbols and $\mathcal{L}(\mathcal{W})(N_1, N_2)$ the defined language. We have the following example of a derivation:

⟨bitstring of length iii with value ii⟩ →
⟨bitstring of length ii with value i⟩`0` →
⟨bitstring of length i with value⟩`10` →
⟨bitstring of length with value⟩`010` →
`010`

Notice that many languages, for example, $\mathcal{L}(\mathcal{W})(\epsilon, i)$ are empty since there are many nonterminals for which there are no productions, for example:

⟨bitstring of length with value i⟩

□

Let us now try to use two level grammars to define the typed version of the following programming language.

⟨expr⟩ ::= ⟨var⟩
| **true**
| **false**
| (⟨expr⟩ & ⟨expr⟩)
| **fn** ⟨var⟩ : ⟨type⟩ => ⟨expr⟩
| ⟨expr⟩ (⟨expr⟩)

We need a nonterminal for each type and each type context. We get this by giving a metagrammar that defines what we mean by type and type context.

metagrammar

T ::= bool | $(T_1 \to T_2)$
C ::= $V : T; C$ | ϵ
V ::= x1 | x2 | ...

Now we can write productions that define expressions of different types

objectgrammar

$$
\begin{array}{lcl}
\langle \text{expr}(C, \text{bool})\rangle & ::= & \texttt{true} \\
 & | & \texttt{false} \\
 & | & \texttt{(}\langle \text{expr}(C, \text{bool})\rangle\ \texttt{\&}\ \langle \text{expr}(C,\text{bool})\rangle\ \texttt{)} \\
\langle \text{expr}(C, (T_1 \to T_2)\rangle & ::= & \texttt{fn}\ \langle V\rangle\ \texttt{:}\ \langle T_1\rangle\ \texttt{=>}\ \langle \text{expr}(V : T_1\ ;\ C, T_2)\rangle \\
\langle \text{expr}(C, T_2)\rangle & ::= & \langle \text{expr}(C, (T_1 \to T_2))\rangle\ \texttt{(}\ \langle \text{expr}(C, T_1)\rangle\ \texttt{)} \\
\langle \text{expr}(V : T\ ;\ C, T)\rangle & ::= & \langle V\rangle \\
\langle \text{expr}(V : T_1\ ;\ C, T_2)\rangle & ::= & \langle \text{expr}(C,T_2)\rangle \\
\langle \text{bool}\rangle & ::= & \texttt{bool} \\
\langle (T_1 \to T_2)\rangle & ::= & \texttt{(}\ \langle T_1\rangle\ \texttt{->}\ \langle T_2\rangle\ \texttt{)} \\
\langle \text{x1}\rangle & ::= & \texttt{x1} \\
\langle \text{x2}\rangle & ::= & \texttt{x2} \\
 & \vdots &
\end{array}
$$

Example: A derivation in this grammar could look like:

$\langle \text{expr}(\epsilon,(\text{bool} \to \text{bool})\rangle$

$\to$ `fn` $\langle V\rangle$`:bool =>`$\langle \text{expr}(V\text{:bool};\ \epsilon,\ \text{bool}\rangle$

$\to$ `fn` $\langle \text{x1}\rangle$`:bool =>`$\langle \text{expr}(\text{x1:bool};\ \epsilon,\ \text{bool}\rangle$

$\to$ `fn x1:bool => (`$\langle \text{expr}(\text{x1:bool};\ \epsilon,\ \text{bool})\rangle$`&`$\langle \text{expr}(\text{x1:bool};\ \epsilon,\ \text{bool})\rangle$`)`

$\to$ `fn x1:bool => (x1 & false)`

□

That two level grammars are very powerful can be seen from the following example from [14], which defines the language of provable formulas in first order logic. Notice how predicates are defined, and how inductive types such as lists are introduced. Also notice that the language which is defined is nontrivial.

Example: First define the calculus. We take the formal system for natural deduction instead of the Hilbert calculus which was used in [14]. The situation is very similar to the situation when one wants to represent a formal calculus in a logical framework – the logical framework of two level grammars just contain a construction for inductive types and pattern matching.

First we define "inductive types" for judgement sequences (Js), contexts (Γ), propositions (formulas) (F), propositional constants (P), term sequences (Ts), function constants (Fc), individual constants (C) and variables (X) in the meta grammar. For simplicity, we assume we have a metanotion N for natural numbers.

We represent a context as a sequence of propositions and a proposition is defined by an inductive type as usual.

Judgement sequences:
$Js \quad ::= \quad \epsilon \quad | \quad \Gamma \vdash F \quad | \quad Js_1; Js_2$

Contexts:
$\Gamma \quad ::= \quad \epsilon \quad | \quad F \quad | \quad \Gamma, \Gamma$

Formulas:
$F \quad ::= \quad P \quad | \quad P(Ts)$
$\quad | \quad (F_1 \,\&\, F_2) \quad | \quad (F_1 \vee F_2) \quad | \quad (F_1 \supset F_2)$
$\quad | \quad \neg F \quad | \quad (\forall X.F) \quad | \quad (\exists X.F)$

Predicates:
$P \quad ::= \quad P_N$

Terms:
$T \quad ::= \quad C \quad | \quad X \quad | \quad Fc(Ts)$

Term sequences:
$Ts \quad ::= \quad T \quad | \quad T,\ Ts$

Constants:
$C \quad ::= \quad C_N$

Variables:
$X \quad ::= \quad X_N$

Function constants:
$Fc \quad ::= \quad Fc_N$

In the object grammar, we represent the rules for how we can derive judgements in the system.

Startsymbol:
$\langle \text{start} \rangle$
$\quad ::= \quad \langle \text{truths}() \rangle$

General rules:
$\langle \text{truths}(Js) \rangle$
$\quad ::= \quad \langle \text{truths}(Js;\ \ F \vdash F) \rangle$
$\langle \text{truths}(Js_1\,;\ \Gamma \vdash F_2;\ \ Js_2) \rangle$
$\quad ::= \quad \langle \text{truths}(Js\,;\ \Gamma, F_1 \vdash F_2;\ \ Js_2) \rangle$
$\langle \text{truths}(Js_1;\ \Gamma_1, F_1, \Gamma_2, F_1, \Gamma_3 \vdash F_2;\ \ Js_2) \rangle$
$\quad ::= \quad \langle \text{truths}(Js_1;\ \Gamma_1, F_1, \Gamma_2, \Gamma_3 \vdash F_2;\ \ Js_2) \rangle$
$\langle \text{truths}(Js_1;\ \Gamma_1 \vdash F_1;\ \ Js_2;\ \ \Gamma_2, F_1 \vdash F_2;\ \ Js_3) \rangle$
$\quad ::= \quad \langle \text{truths}(Js_1;\ \Gamma_1 \vdash F_1;\ \ Js_2;\ \ \Gamma_2, F_1 \vdash F_2;\ \ Js_3;\ \ \Gamma_1, \Gamma_2 \vdash F_2) \rangle$

Conjunction:
$\langle \text{truths}(Js_1;\ \Gamma_1 \vdash F_1;\ \ Js_2;\ \ \Gamma_2 \vdash F_2;\ \ Js_3) \rangle$
$\quad ::= \quad \langle \text{truths}(Js_1;\ \Gamma_1 \vdash F_1;\ \ Js_2;\ \ \Gamma_2 \vdash F_2;\ \ Js_3;\ \ \Gamma_1, \Gamma_2 \vdash (F_1 \,\&\, F_2)) \rangle$
$\langle \text{truths}(Js_1;\ \Gamma_1 \vdash (F_1 \,\&\, F_2);\ \ Js_2) \rangle$
$\quad ::= \quad \langle \text{truths}(Js_1;\ \Gamma_1 \vdash (F_1 \,\&\, F_2);\ \ Js_2;\ \ \Gamma_1 \vdash F_1) \rangle$
$\langle \text{truths}(Js_1;\ \Gamma_1 \vdash (F_1 \,\&\, F_2);\ \ Js_2) \rangle$
$\quad ::= \quad \langle \text{truths}(Js_1;\ \Gamma_1 \vdash (F_1 \,\&\, F_2);\ \ Js_2;\ \ \Gamma_1 \vdash F_2) \rangle$

Disjunction:

$$\langle \text{truths}(Js_1;\ \Gamma_1 \vdash F_1;\ Js_2)\rangle$$
$$::= \quad \langle \text{truths}(Js_1;\ \Gamma_1 \vdash F_1;\ Js_2;\ \Gamma_1 \vdash (F_1 \vee F_2))\rangle$$
$$\langle \text{truths}(Js_1;\ \Gamma_2 \vdash F_2;\ Js_2)\rangle$$
$$::= \quad \langle \text{truths}(Js_1;\ \Gamma_2 \vdash F_2;\ Js_2;\ \Gamma_2 \vdash (F_1 \vee F_2))\rangle$$
$$\langle \text{truths}(Js_1;\ \Gamma_1 \vdash (F_1 \vee F_2);\ Js_2;\ \Gamma_2, F_1 \vdash F_3;\ Js_3;\ \Gamma_3, F_2 \vdash F_3;\ Js_4)\rangle$$
$$::= \quad \langle \text{truths}(Js_1;\ \Gamma_1 \vdash (F_1 \vee F_2);\ Js_2;\ \Gamma_2, F_1 \vdash F_3;\ Js_3;\ \Gamma_3, F_2 \vdash F_3;$$
$$Js_4;\ \Gamma_1, \Gamma_2, \Gamma_3 \vdash F_3)\ \rangle$$

Implication:

$$\langle \text{truths}(Js_1;\ \Gamma_1, F_1 \vdash F_2;\ Js_2)\rangle$$
$$::= \quad \langle \text{truths}(Js_1;\ \Gamma_1, F_1 \vdash F_2;\ Js_2;\ \Gamma_1 \vdash (F_1 \supset F_2))\rangle$$
$$\langle \text{truths}(Js_1;\ \Gamma_1 \vdash (F_1 \supset F_2);\ Js_2;\ \Gamma_2 \vdash F_1;\ Js_3)\rangle$$
$$::= \quad \langle \text{truths}(Js_1;\ \Gamma_1 \vdash (F_1 \supset F_2);\ Js_2;\ \Gamma_2 \vdash F_1;\ Js_3;\ \Gamma_1, \Gamma_2 \vdash F_2)\rangle$$

Negation:

$$\langle \text{truths}(Js_1;\ \Gamma_1, F_1 \vdash F_2;\ Js_2;\ \Gamma_2, F_1 \vdash \neg F_2)\rangle$$
$$::= \quad \langle \text{truths}(Js_1;\ \Gamma_1, F_1 \vdash F_2;\ Js_2;\ \Gamma_2, F_1 \vdash \neg F_2;\ \Gamma_1, \Gamma_2 \vdash \neg F_1)\rangle$$
$$\langle \text{truths}(Js_1;\ \Gamma_1 \vdash F_1;\ Js_2;\ \Gamma_2 \vdash \neg F_1)\rangle$$
$$::= \quad \langle \text{truths}(Js_1;\ \Gamma_1 \vdash F_1;\ Js_2;\ \Gamma_2 \vdash \neg F_1;\ \Gamma_1, \Gamma_2 \vdash F_2)\rangle$$

We leave the rules for the universal and existential quantifiers as exercises.

We have defined sequences of judgements and sequences of formulas in a context by clauses for an empty sequence (ϵ), for an one element sequence and for an append operation (; and , respectively). We get a lot of ugly sequences by this method, since we may append any number of empty lists everywhere. Having an invisible append operator (and an invisible empty sequence) gives a nicer notation, but it gets a little bit too subtle. Instead we add the following rules to get rid of extra semicolons and rules to add necessary semicolons in judgement sequences.

$$\langle \text{truths}(;\ Js)\rangle \quad ::= \quad \langle \text{truths}(Js)\rangle$$
$$\langle \text{truths}(Js_1\ ;\ ;\ Js_2)\rangle \quad ::= \quad \langle \text{truths}(Js_1\ ;\ Js_2)\rangle$$
$$\langle \text{truths}(Js\ ;)\rangle \quad ::= \quad \langle \text{truths}(Js)\rangle$$

$$\langle \text{truths}(Js)\rangle \quad ::= \quad \langle \text{truths}(;\ Js)\rangle$$
$$\langle \text{truths}(Js_1\ ;\ Js_2)\rangle \quad ::= \quad \langle \text{truths}(Js_1\ ;\ ;\ Js_2)\rangle$$
$$\langle \text{truths}(Js)\rangle \quad ::= \quad \langle \text{truths}(Js\ ;)\rangle$$

We do the same for commas in contexts.

$$\langle \text{truths}(Js_1\ ;\ ,\Gamma \vdash F\ ;\ Js_2)\rangle \quad ::= \quad \langle \text{truths}(Js_1\ ;\ \Gamma \vdash F\ ;\ Js_2)\rangle$$
$$\langle \text{truths}(Js_1\ ;\ \Gamma_1, ,\Gamma_2 \vdash F\ ;\ Js_2)\rangle \quad ::= \quad \langle \text{truths}(Js_1\ ;\ \Gamma_1, \Gamma_2 \vdash F\ ;\ Js_2)\rangle$$
$$\langle \text{truths}(Js_1\ ;\ \Gamma, \vdash F\ ;\ Js_2)\rangle \quad ::= \quad \langle \text{truths}(Js_1\ ;\ \Gamma \vdash F\ ;\ Js_2)\rangle$$

$$\langle \text{truths}(Js_1\ ;\ \Gamma \vdash F\ ;\ Js_2)\rangle \quad ::= \quad \langle \text{truths}(Js_1\ ;\ ,\Gamma \vdash F\ ;\ Js_2)\rangle$$
$$\langle \text{truths}(Js_1\ ;\ \Gamma_1, \Gamma_2 \vdash F\ ;\ Js_2)\rangle \quad ::= \quad \langle \text{truths}(Js_1\ ;\ \Gamma_1, ,\Gamma_2 \vdash F\ ;\ Js_2)\rangle$$
$$\langle \text{truths}(Js_1\ ;\ \Gamma \vdash F\ ;\ Js_2)\rangle \quad ::= \quad \langle \text{truths}(Js_1\ ;\ \Gamma, \vdash F\ ;\ Js_2)\rangle$$

Finally we need some productions to generate the text from a derived nonterminal.

Theorems:

$\langle \text{truths}(Js_1;\ \Gamma \vdash F;\ Js_2) \rangle$ ::= $\langle \text{printJ}(\Gamma \vdash F) \rangle$

Judgements:

$\langle \text{printJ}(\Gamma \vdash F) \rangle$ ::= $\langle \text{printC}(\Gamma) \rangle \vdash \langle \text{printF}(\text{F}) \rangle$

Conclusions:

$\langle \text{printC}(\epsilon) \rangle$ ::= ϵ
$\langle \text{printC}(\text{F}) \rangle$ ::= $\langle \text{printF}(\text{F}) \rangle$
$\langle \text{printC}(\Gamma, F) \rangle$ ::= $\langle \text{printC}(\Gamma) \rangle$ `,` $\langle \text{printF}(\text{F}) \rangle$

Formulas:

$\langle \text{printF}(P(Ts)) \rangle$ ::= $\langle \text{printP}(\text{P}) \rangle$ `(` $\langle \text{printTs}(\text{Ts}) \rangle$ `)`
$\langle \text{printF}((F_1 \,\&\, F_2)) \rangle$ ::= `(` $\langle \text{printF}(F_1) \rangle$ `&` $\langle \text{printF}(F_2) \rangle$ `)`
$\langle \text{printF}((F_1 \vee F_2)) \rangle$ ::= `(` $\langle \text{printF}(F_1) \rangle$ $\vee$ $\langle \text{printF}(F_2) \rangle$ `)`
$\langle \text{printF}((F_1 \supset F_2)) \rangle$ ::= `(` $\langle \text{printF}(F_1) \rangle$ $\supset$ $\langle \text{printF}(F_2) \rangle$ `)`
$\langle \text{printF}(\neg F) \rangle$ ::= $\neg$ $\langle \text{printF}(F) \rangle$
$\langle \text{printF}((\forall X.F)) \rangle$ ::= `(` $\forall$ $\langle \text{printX}(X) \rangle$ `.` $\langle \text{printF}(F) \rangle$ `)`
$\langle \text{printF}((\exists X.F)) \rangle$ ::= `(` $\exists$ $\langle \text{printX}(X) \rangle$ `.` $\langle \text{printF}(F) \rangle$ `)`

Predicates:

$\langle \text{printP}(P_N) \rangle$ ::= $\texttt{P}_\texttt{N}$

Terms:

$\langle \text{printT}(C) \rangle$ ::= $\langle \text{printC}(C) \rangle$
$\langle \text{printT}(X) \rangle$ ::= $\langle \text{printX}(X) \rangle$
$\langle \text{printT}(Fc(Ts)) \rangle$ ::= $\langle \text{printFc}(Fc) \rangle$ `(` $\langle \text{printTs}(Ts) \rangle$ `)`

Term sequences:

$\langle \text{printTs}(T) \rangle$ ::= $\langle \text{printT}(T) \rangle$
$\langle \text{printTs}(T, Ts) \rangle$ ::= $\langle \text{printT}(T) \rangle$ `,` $\langle \text{printTs}(Ts) \rangle$

Constants:

$\langle \text{printC}(C_N) \rangle$ ::= $\texttt{c}_\texttt{N}$

Variables:

$\langle \text{printX}(X_N) \rangle$ ::= $\texttt{x}_\texttt{N}$

Function constants:

$\langle \text{printFc}(Fc_N) \rangle$ ::= $\texttt{f}_\texttt{N}$

Let us end this example by a derivation. We show in detail how the judgement

$$((P_1 \,\&\, P_2) \supset (P_1 \vee P_2))$$

can be derived using the two-level grammar defined above. Some of the steps are there just to introduce the necessary semicolons and commas for the sequents to match the rules.

$\langle \text{start} \rangle$
$\rightarrow$ $\langle \text{truths}() \rangle$
$\rightarrow$ $\langle \text{truths}(;\ (P_1 \,\&\, P_2) \vdash (P_1 \,\&\, P_2)) \rangle$
$\rightarrow$ $\langle \text{truths}(;\ (P_1 \,\&\, P_2) \vdash (P_1 \,\&\, P_2);) \rangle$
$\rightarrow$ $\langle \text{truths}(;\ (P_1 \,\&\, P_2) \vdash (P_1 \,\&\, P_2);;\ (P_1 \,\&\, P_2) \vdash P_1) \rangle$
$\rightarrow^2$ $\langle \text{truths}(;\ (P_1 \,\&\, P_2) \vdash (P_1 \,\&\, P_2);;\ (P_1 \,\&\, P_2) \vdash P_1;) \rangle$
$\rightarrow$ $\langle \text{truths}(;\ (P_1 \,\&\, P_2) \vdash (P_1 \,\&\, P_2);;\ (P_1 \,\&\, P_2) \vdash P_1;;\ ,(P_1 \,\&\, P_2) \vdash (P_1 \vee P_2)) \rangle$
$\rightarrow$ $\langle \text{truths}(;\ (P_1 \,\&\, P_2) \vdash (P_1 \,\&\, P_2);;\ (P_1 \,\&\, P_2) \vdash P_1;;\ ,(P_1 \,\&\, P_2) \vdash (P_1 \vee P_2);; \vdash (P_1 \,\&\, P_2) \supset (P_1 \vee P_2)) \rangle$
$\rightarrow$ $\langle \text{truths}(;\ (P_1 \,\&\, P_2) \vdash (P_1 \,\&\, P_2);;\ (P_1 \,\&\, P_2) \vdash P_1;;\ ,(P_1 \,\&\, P_2) \vdash (P_1 \vee P_2);; \vdash (P_1 \,\&\, P_2) \supset (P_1 \vee P_2);) \rangle$
$\rightarrow$ $\langle \text{printJ}(\vdash (P_1 \,\&\, P_2) \supset (P_1 \vee P_2)) \rangle$
$\rightarrow$ $\langle \text{printC}() \rangle \vdash \langle \text{printF}((P_1 \,\&\, P_2) \supset (P_1 \vee P_2)) \rangle$
$\rightarrow$ $\vdash (\ \langle \text{printF}((P_1 \,\&\, P_2)) \rangle\ \supset\ \langle \text{printF}((P_1 \vee P_2)) \rangle\ )$
$\rightarrow$ $\vdash (\ (\langle \text{printF}(P_1) \rangle\ \&\ \langle \text{printF}(P_2) \rangle)\ \supset\ (\langle \text{printF}(P_1) \rangle\ \vee\ \langle \text{printF}(P_2) \rangle)\ )$
$\rightarrow$ $\vdash (\ (\langle \text{printP}(P_1) \rangle\ \&\ \langle \text{printP}(P_2) \rangle)\ \supset\ (\langle \text{printP}(P_1) \rangle\ \vee\ \langle \text{printP}(P_2) \rangle)\ )$
$\rightarrow$ $\vdash (\ (\mathtt{P}_1\ \&\ \mathtt{P}_2)\ \supset\ (\mathtt{P}_1\ \vee\ \mathtt{P}_2)\ )$

□

Exercise 8 *Indicate what rules and what substitutions that have been used in the derivation of the judgement*

$((\mathtt{P}_1 \,\&\, \mathtt{P}_2) \supset (\mathtt{P}_1 \vee \mathtt{P}_2))$

3.1.1 Syntaxanalysis of languages described by two-level grammars

Let us see how parsing could be performed for languages defined by two-level grammars. Consider Earley's algorithm which we introduced earlier. Remember that it uses three different operations during the parsing:

predict when the dot is in front of a nonterminal all items for that nonterminal is added to the item set,

scanner when the dot is in front of a terminal the dot is moved one step in those items that match the next terminal in the input string; the items is moved to the next item set.

complete when the dot is at the end of a production the dot is moved one step forward for all items which once expected the nonterminal (those items in the item set pointed to by the backpointer which have the dot in front of the nonterminal that just has been parsed); these items are moved to the present item set.

If we extend our grammar formalism with nonterminal schemas as we have done for the two level grammars, and if we want to use Earley's parsing algorithm we have to generalize the predict and complete operations.

In the *predict operation* we must add all production schemas whose LHS *unifies* with the nonterminal the dot is in front of. If we have the situation

... •⟨expr(X1:bool;,bool)⟩ ...

then we must, for example, add the dotted production

⟨expr(X1:bool;,bool)⟩ ::= •`true`

which is an instance of the production

⟨expr(C,bool)⟩ ::= `true`

and also, which is more complicated, for every type T add the dotted productions

⟨expr(X1:bool;, bool)⟩ ::= ⟨expr(X1:bool;, $(T \to \text{bool})$⟩`(`⟨expr(X1:bool;, T)⟩`)`

This example indicates in short the problems when parsing languages defined by two level grammars. As we have already said, the two level grammars are capable of defining any recursively enumerable language (set), so we can not expect to find a parser for all the languages defined by two level grammars.

Exercise 9 *Write the productions that corresponds to the inference rules for the universal and existential quantifiers in the example on page 27.*

Notice that you can "program" the side condition that a variable must not occur in an assumption list in the same way as you program a predicate in Prolog. The same holds for the substitution operation which is also needed.

Exercise 10 *Define the typed version of the following language of expressions by a two level grammar.*

⟨expr⟩ ::= `let` ⟨var⟩ `:` ⟨type⟩ `=` ⟨expr⟩
`in` ⟨expr⟩
`end`
| ⟨expr⟩ `+` ⟨expr⟩
| `1` | `2` | `3` | `4`
| `(` ⟨expr⟩ `,` ⟨expr⟩ `)`
| `fst` ⟨expr⟩
| `snd` ⟨expr⟩
⟨var⟩ ::= `x` | `y` | `z`
⟨type⟩ ::= `Int` | `(`⟨type⟩ `*` ⟨type⟩`)`

3.2 Attribute grammars

To define a programming language or a set you can use two different methods:

1. Define a process that generates all programs in the language.
2. Define the set of programs by giving a superset and a condition that must be satisfied.

Examples of set definitions that use the first method is the definition of inductive sets, the context-free languages and language definitions by two level grammars. An example of the second method is when we define a subset of a set which we already know exists

$$\begin{aligned} \text{Odd} &= \{x \in \text{N} \mid (\exists y \in \text{N})\, 2 * y + 1 = x\} \\ \text{ML} &= \{x \in \text{SyntaxML} \mid \text{TypeCorrect}(x)\ \&\ \ldots\} \end{aligned}$$

A grammar formalism which is based on the second method is the *attribute grammar* formalism [22].

In some situations it is very convenient to define a language by this method, for example, if we want to define the set of identifiers (being sequences of letters for simplicity)

$$Id^- = \{x \in (a-z)^+\} \setminus \{\texttt{let}, \texttt{in}, \texttt{end}\}$$

Exercise 11 *Define the language of identifiers by a BNF-grammar. The reserved words* `let`, `in`, `end` *are not identifiers.*

Using subset notation we can just write

$$Id^- = \{x \in (a-z)^+ \mid x \neq \texttt{let} \;\&\; x \neq \texttt{in} \;\&\; x \neq \texttt{end}\}$$

An attribute grammar is a kind of subset notation for languages. It consists of a BNF grammar, which generates the base set, together with programs (evaluation rules) which associates a value of type $A_{N_1} \times A_{N_n} \times \text{Bool}$ to every node in the parse trees. We call the first n components of the tuple *the attributes* and the last *the condition*. A parse tree is *correct* if all nodes in the tree have conditions which are true. The attributes associated to each node in the parse tree are defined by giving *evaluation rules* together with the productions of the grammar. The attributes are only used to indirectly determine if a parse tree is correct since the conditions are defined in terms of the attributes.

Attribute grammars, and similar formalisms, are often used for defining *syntax oriented computations*, for example using parser generators such as Yacc. Here the attributes are more interesting in themselves.

Example: Let us define the typed version of the language

⟨expr⟩ ::= ⟨var⟩
| `true`
| `false`
| (⟨expr⟩ `&` ⟨expr⟩)
| `fn` ⟨var⟩ `:` ⟨type⟩ `=>` ⟨expr⟩
| ⟨expr⟩ `(`⟨expr⟩`)`

by an attribute grammar. Compare the definition with the definition using two level grammar on page 26. We associate attributes to the nonterminals as follows:

name (⟨var⟩.name): Associate a *string* to the nonterminal ⟨var⟩. The string is intended to represent the name of the variable.

context (⟨expr⟩.context): Associate a *type context* (a list of pairs of variables and types) to the nonterminal ⟨expr⟩. The context is intended to represent the type of all variables which could be used in the expression

type (⟨expr⟩.type, ⟨type⟩.type): We also associate a *type* to the nonterminal ⟨expr⟩ and the nonterminal ⟨type⟩. This attribute represents the type of the expression or the name of the type.

We write the evaluation rules inside braces after every production. The condition is defined using a variable with name cond and the evaluation rules are written in a ML like language with a *let* construction, a cons (::) operator and an equality operator $\equiv$.

Declaration of attributes:

the nonterminal ⟨var⟩ has an attribute:
name : String

the nonterminal ⟨expr⟩ have attributes
context : List(String×Type), type : Type

the nonterminal ⟨type⟩ has an attribute
type : Type

where datatype Type = bool | Type → Type

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \texttt{true} \mid \texttt{false} \\
 & & \{ \quad \langle \text{expr} \rangle.\text{type} = \text{bool} \\
 & & \qquad \text{cond} = \text{true} \\
 & & \} \\
\langle \text{expr}_i \rangle & ::= & \texttt{(}\ \langle \text{expr}_j \rangle\ \texttt{\&}\ \langle \text{expr}_k \rangle\ \texttt{)} \\
 & & \{ \quad \langle \text{expr}_j \rangle.\text{context} = \langle \text{expr} \rangle.\text{context} \\
 & & \quad \langle \text{expr}_k \rangle.\text{context} = \langle \text{expr} \rangle.\text{context} \\
 & & \quad \langle \text{expr}_i \rangle.\text{type} = \text{bool} \\
 & & \qquad \text{cond} = \langle \text{expr}_j \rangle.\text{type} \equiv \text{bool}\ \& \\
 & & \qquad\qquad \langle \text{expr}_k \rangle.\text{type} \equiv \text{bool}\ \} \\
\langle \text{expr}_i \rangle & ::= & \texttt{fn}\ \langle \text{var} \rangle\ \texttt{:}\ \langle \text{type} \rangle\ \texttt{=>}\ \langle \text{expr}_j \rangle \\
 & & \{ \quad \langle \text{expr}_j \rangle.\text{context} = \langle \text{var} \rangle.\text{name} \in \langle \text{type} \rangle.\text{type} :: \langle \text{expr}_i \rangle.\text{context} \\
 & & \quad \langle \text{expr}_i \rangle.\text{type} = (\langle \text{type} \rangle.\textit{type} \to \langle \text{expr}_j \rangle.\textit{type}) \\
 & & \qquad \text{cond} = \text{true} \\
 & & \} \\
\langle \text{expr}_i \rangle & ::= & \langle \text{expr}_j \rangle\ \texttt{(}\ \langle \text{expr}_k \rangle\ \texttt{)} \\
 & & \{ \quad \langle \text{expr}_j \rangle.\text{context} = \langle \text{expr}_i \rangle.\text{context} \\
 & & \quad \langle \text{expr}_k \rangle.\text{context} = \langle \text{expr}_i \rangle.\text{context} \\
 & & \quad \langle \text{expr}_i \rangle.\text{type} = \textit{let}\ (\alpha \to \beta) = \langle \text{expr}_j \rangle.\text{type}\ \textit{in}\ \beta \\
 & & \qquad \text{cond} = \langle \text{expr}_j \rangle.\text{type} \equiv (\alpha \to \beta)\ \& \\
 & & \qquad\qquad \langle \text{expr}_k \rangle.\text{type} \equiv \alpha \\
 & & \} \\
\langle \text{expr} \rangle & ::= & \langle \text{var} \rangle \\
 & & \{ \quad \langle \text{expr} \rangle.\text{type} = \text{lookup}(\langle \text{var} \rangle.\text{var}, \langle \text{expr} \rangle.\text{context}) \\
 & & \qquad \text{cond} = \text{member}(\langle \text{var} \rangle.\text{var}, \langle \text{expr} \rangle.\text{context}) \\
 & & \}
\end{array}
$$

The function lookup returns the type which is associated with a variable in a context and the function member determines if a variable is defined in a context. □

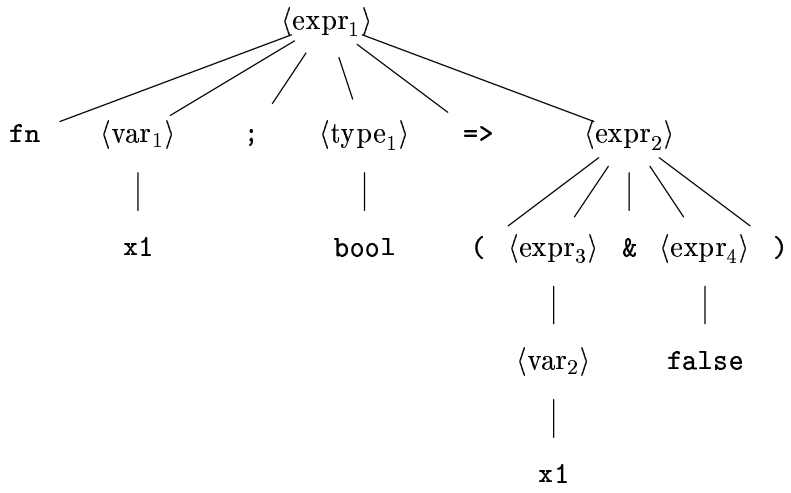
The attribute *type* is a *synthesized* attribute since it is defined for a nonterminal in the left hand side of the productions using attributes of nonterminals in the right hand side of the production. The attribute *context*, on the other hand, is an *inherited* attribute since it is defined for a nonterminal in the right hand side of the productions in terms of attributes of other nonterminals in the same production. In terms of parse trees, a synthezised attribute of a node is defined in terms of attributes of the children in the tree

and an inherited attribute is defined in terms of attributes of siblings and parents in the tree.

Let us see a derivation using the attribute grammar above. We start by using the ordinary derivation of `fn x1:bool => (x1 & false)`.

$$\begin{aligned}\langle\text{expr}\rangle &\rightarrow \texttt{fn}\langle\text{var}\rangle\texttt{:}\langle\text{type}\rangle\texttt{=>}\langle\text{expr}\rangle\\ &\rightarrow \texttt{fn x1:bool => (}\langle\text{expr}\rangle\texttt{\&}\langle\text{expr}\rangle\texttt{)}\\ &\rightarrow \texttt{fn x1:bool => (x1 \& false)}\end{aligned}$$

The parse tree corresponding to this derivation is



The ordinary parse tree for `fn x1:bool => (x1 & false)`

We regard the nonterminals ⟨var⟩ and ⟨type⟩ as primitive and assume that their attributes are defined. Furthermore we let the attribute context of the root node have the empty list as value.

$$\begin{aligned}\langle\text{type}_1\rangle\text{.type} &= \text{bool}\\ \langle\text{var}_1\rangle\text{.name} &= \text{x1}\\ \langle\text{var}_2\rangle\text{.name} &= \text{x1}\\ \langle\text{expr}_1\rangle\text{.context} &= \text{nil}\end{aligned}$$

Given these initial values we can compute the attributes of the other nodes in the parse tree.

$$
\begin{array}{rcl}
\langle \text{type}_1 \rangle.\text{type} & = & \text{bool} \\
\langle \text{var}_1 \rangle.\text{name} & = & \text{x1} \\
\langle \text{var}_2 \rangle.\text{name} & = & \text{x1} \\
\langle \text{expr}_2 \rangle.\text{context} & = & \langle \text{var}_1 \rangle.\text{name} \in \langle \text{type}_1 \rangle.\text{type} :: \text{nil} \\
 & = & [\text{x1} \in \text{bool}] \\
\langle \text{expr}_3 \rangle.\text{context} & = & \langle \text{expr}_2 \rangle.\text{context} \\
 & = & [\text{x1} \in \text{bool}] \\
\langle \text{expr}_4 \rangle.\text{context} & = & \langle \text{expr}_2 \rangle.\text{context} \\
 & = & [\text{x1} \in \text{bool}] \\
\langle \text{expr}_4 \rangle.\text{type} & = & \text{bool} \\
\langle \text{expr}_3 \rangle.\text{type} & = & \text{lookup}(\text{x1}, [\text{x1} \in \text{bool}]) \\
 & = & \text{bool} \\
\langle \text{expr}_2 \rangle.\text{type} & = & \text{bool} \\
\langle \text{expr}_1 \rangle.\text{type} & = & (\text{bool} \rightarrow \text{bool})
\end{array}
$$

Finally we can compute the condition of each node in the parse tree to see if the tree is a legal parse tree and the corresponding string belongs to the language.

$$
\begin{array}{rcl}
\langle \text{expr}_1 \rangle.\text{cond} & = & \text{true} \\
\langle \text{expr}_2 \rangle.\text{cond} & = & \langle \text{expr}_3 \rangle.\text{type} \equiv \text{bool} \ \& \ \langle \text{expr}_4 \rangle.\text{type} \equiv \text{bool} \\
 & = & \text{true} \\
\langle \text{expr}_3 \rangle.\text{cond} & = & \text{member}(\langle \text{var}_2 \rangle.\text{name}, \langle \text{expr}_3 \rangle.\text{context}) \\
 & = & \text{member}(\text{x1}, [\text{x1} \in \text{bool}]) \\
 & = & \text{true} \\
\langle \text{expr}_4 \rangle.\text{cond} & = & \text{true}
\end{array}
$$

To parse a string using an attribute grammar is not easy. To parse a string with the underlying context free grammar is of course no problem, but deciding in what order the attributes should be computed is in general an undecidable problem(?) [**?**, 20].

3.3 Type systems as logical theories

A third method to define a type system is to see it as a "logical theory" and use rules such as those in Martin-Löf's type theory to define the type system. Cardelli [10] and Damas/Milner [12] has described the type system of a functional programming language by this method.

First let us see what do we want from a type system in a programming language. There are three important aspects of type systems in programming languages.

Decidability It should be decidable if a program is type correct or not.

Expressiveness It is important that the type system does not stop the programmer from using sound and efficient methods of programming.

Soundness We separate what we may demand in terms of soundness.

1. The evaluation must not "go wrong", i.e. a type correct program must never apply an operator to an operand for which the operation is not defined.

2. The evaluation must terminate. Some people require that a type correct program must always terminate.

Let us begin by defining a simple type system:

$$
\begin{array}{lcl}
\langle \text{type} \rangle & ::= & \texttt{Int} \\
& | & \texttt{Bool} \\
& | & \texttt{List(}\ \langle \text{type} \rangle\ \texttt{)} \\
& | & \langle \text{typevariable} \rangle \\
& | & \texttt{(}\ \langle \text{type} \rangle\ \texttt{->}\ \langle \text{type} \rangle\ \texttt{)} \\
& | & \forall\ \langle \text{var} \rangle\ \texttt{.}\ \langle \text{type} \rangle
\end{array}
$$

Since we have typevariables in the type expression we have a polymorphic type system.

The rules for the system are (see Cardellis paper [10], page 8) in the notation used in papers about Martin-Löf's type theory

$$\frac{}{x \in \alpha \quad [x \in \alpha]}\ \text{Var}$$

$$\frac{e_1 \in \texttt{Bool} \quad e_2 \in \alpha \quad e_3 \in \alpha}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \in \alpha}\ \text{Cond}$$

$$\frac{e \in \beta \quad [x \in \alpha]}{\texttt{fn}\ x\ \texttt{=>}\ e \in \alpha \to \beta}\ \text{Abs}$$

$$\frac{e_1 \in \alpha \to \beta \quad e_2 \in \alpha}{e_1\texttt{(}e_2\texttt{)} \in \beta}\ \text{App}$$

$$\frac{e_1 \in \alpha \quad e_2 \in \beta \quad [x \in \alpha]}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \in \beta}\ \text{Let}$$

$$\frac{e \in \alpha \to \beta \quad [x \in \alpha \to \beta]}{\texttt{rec}\ x\ \texttt{=>}\ e \in \alpha \to \beta}\ \text{Rec}$$

$$\frac{e \in \beta}{e \in \forall\alpha.\beta}\ \text{Gen} \qquad \text{where } \alpha \text{ is not free in any assumption}$$

$$\frac{e \in \forall\alpha.\beta}{e \in \beta[\alpha := \gamma]}\ \text{Spec}$$

Note that we can express the rules in two different ways concerning the assumption lists. One is good for doing proofs from the assumptions to the conclusion and the other when we want to go from the conclusion to the assumptions. Expressing the assumption lists explicitly in the rules, we can either write:

$$\frac{e_1 \in \alpha \to \beta\ [\Gamma] \quad e_2 \in \alpha\ [\Gamma]}{e_1\texttt{(}e_2\texttt{)} \in \beta\ [\Gamma]}\ \text{App}$$

saying that all components should have the same assumptions, or

$$\frac{e_1 \in \alpha \to \beta\ [\Gamma_1] \quad e_2 \in \alpha\ [\Gamma_2]}{e_1\,(e_2) \in \beta\ [\Gamma_1, \Gamma_2]}\ \text{App}$$

saying that we compute the assumption list for the conclusion from the assumption lists of the premisses. Understanding the rules as in the first case, we need a *thinning rule*

$$\frac{e \in \alpha\ [\Gamma]}{e \in \alpha\ [x \in A, \Gamma, y \in B]}\ \text{Thin}$$

that expresses that we can always extend the assumption list with more assumptions.

Here are some examples of how one can use the type rules to derive that particular expressions have particular types.

Example: We can for example use the rules to type check the program `fn x => x`:

$$\cfrac{\cfrac{\cfrac{}{x \in \alpha\ \ [x \in \alpha]}\ \text{Var}}{\texttt{fn}\ x\ \texttt{=>}\ x \in \alpha \to \alpha}\ \text{Abs}}{\texttt{fn}\ x\ \texttt{=>}\ x \in \forall\alpha.\alpha \to \alpha}\ \text{Gen}$$

□

In order for the typesystem to work we must put one extra condition on the types.

> We only allow *shallow* types, i.e. types in which all quantifiers appear on the outermost level. For example as in $\forall\alpha.\forall\beta.\alpha \to \beta$.

Without this restriction one may, for example, infer that the function `fn x.x(x)` has a type. The only real problem concerning the typesystem for the functional language is the treatment of the `let` expression. In many situation one treats `let` expressions such as `let x = e in e'` as another notation for the application `(fn x => e)(e)`. When we look at typesystems it is often beneficial to treat the `let` expressions differently because we then can type more expressions. The situation is that in the `let` expression we already know what argument the abstraction will be applied on and the abstraction does not have to work for any other expressions. When we type an ordinary abstraction, we do not know in what circumstances it will be used and we have to be conservative and only allow such typings that will work for all arguments.

Example: The expression `fn x => x(x)` is not type correct since it must be applied to an expression which has both the type α and the type $\alpha \to \alpha$. And there is no way of expressing this in an ordinary polymorphic type system. If we on the other hand consider the expression `let f = (fn x => x) in f(f) end` we can see that it works because "the abstraction" is here applied to the identity function and that function has both the types $\beta \to \beta$ and $(\beta \to \beta) \to (\beta \to \beta)$. It is here possible to see exactly what the abstraction is applied to and it could never be used in any other situation. □

In type checking we often talk about *generic* type variables. Briefly a typevariable is generic if it does not appear in the types of any enclosing λ-binder (it does not appear in the set of assumptions). The generic type variables are important because it is those variables which can be universally quantified.

Type checking

One problem when we want to do type checking is to decide when we should use the SPEC and GEN rules. This is a problem because those rules are always applicable. Another problem is to know what set of assumptions to use in the derivation. If the language requires type declarations of the bound variables everything is quite simple, but if that is not required we can use the heuristic that if a rule requires two types to be equal, we make them equal by unifying them. For more information about type derivation see [30]

Exercise 12 *Construct a proof that infers the type of the function* `length` *using the inference rules and the heuristics we have discussed.* `length` *is defined as follows*

```
let length =
    rec length =>
        fn x => if null(x)
                then 0
                else plus(1)(length(tl(x)))
```

and the initial context contains the typings

```
null :  ∀α.  (List(α) -> Bool)
0    :  Int
1    :  Int
plus :  (Int -> (Int -> Int))
tl   :  ∀α.  (List(α) -> List(α))
```

Chapter 4

Semantics

Why should we be interested in the semantics of programming languages? What can be achieved?

The first and obvious thing is of course to *describe* programming languages in a simple and rigorous way. Both to users and to implementors. If we cannot describe the semantics of a programming language, then one of the most important goals with high level languages could not be achieved, – the goal that the programs should be independent of the machine on which they are executed. If there is no rigorous language definition, different implementations would most probably be different and implement slightly different languages.

Besides as a descriptive tool, the semantic definition could also be a prescriptive tool because constructions which are hard to describe, are probably also hard to grasp for human programmers. One reason for the popularity of functional programming languages is their simple semantics.

A semantical description could also be a basis of a programming logic in which one could specify, verify and derive programs. The rules of such a logic could be justified from the semantics (compare Martin-Löf's type theory).

Finally, the semantical description is the basis for a specification of an interpreter and a compiler for the language. The formal description is therefore fundamental if we want to verify those programs formally.

There are different ways of defining the meaning of a programming language, we shall look at two ways which are used for defining the semantics of programming languages. The first is to describe the meaning in terms of how a program is evaluated – this kind of semantical description technique is called *operational semantics*. The other is to translate the syntactical objects of a programming language into mathematical objects which are supposed to be better understood – this technique is called *denotational semantics*.

4.1 Iterative Operational Semantics

When giving an operational semantics to a programming language, you specify, at least in principle, a machine that evaluates an arbitrary program. The semantics of the language is the machine (or the description of the machine).

Operational semantics was the first technique used for defining the semantics of programming language, for example, Peter Landin's SECD-machine, and his attempts to

describe the semantics of ALGOL [24, 25]. Another early use of operational semantics was VDL (Vienna Definition Language) and the description of PL/1 [43]. More recent examples of the use of operational semantics are Martin-Löf's type theory [26, 27, 32], and different process calculus [31].

One of the major problems with operational descriptions is to choose an appropriate level of abstraction, i.e. to choose the primitive operations of the machine. How one chooses depends on what the semantical description is supposed to be used for:

- Describe the language for human beings.
- Define an execution model which rather easy could be implemented on practical computers.

Let us illustrate this point by giving two different semantical descriptions of a strict functional programming language, the SC and the SECD semantics. The language is ordinary λ-calculus.

$$e \ ::= \ c \mid v \mid (\lambda v.e) \mid (e'e'')$$
where $e \in$ Expr, $c \in$ Const, $v \in$ Var

As a background to the method we can say that it is an *operational* method based on *syntactic* transformations of programs and *simple* operations on data. We define the "machines" as terminating transition systems $\langle \Gamma, \rightarrow, T \rangle$ where:

Γ is a set of *configurations* (or states).

$\rightarrow$ is a binary (transition) relation on the set of states, that is $\rightarrow \ \in \Gamma \times \Gamma$.

T is a set final configurations, therefore a subset of Γ. The final configurations are such that $\forall \ \gamma \in T. \ \forall \ \gamma' \in \Gamma. \gamma \not\rightarrow \gamma'$

The notation $\gamma \rightarrow \gamma'$ means that there is a transition from the configuration γ to the configuration γ'.

4.1.1 SC-semantics

Take substitution of λ-expressions as a primitive operation of our machine:

$e[x := e']$ is the expression where all free occurrences of x is replaced by e' and no free variables in e' become bound.

We also assume we have an operation that gives the meaning of functional constants:

$$\text{Constapply} \ \in \ \text{Const} \times \text{Const} \rightarrow \text{Value}$$

A configuration consists of two components:

S a stack (list) of values (the results of evaluated expressions)

C a stack of unevaluated expressions

The transition relation is defined by

$$\begin{aligned}
\langle S, c :: C\rangle &\rightarrow \langle c :: S, C\rangle \\
\langle S, (\lambda x.e) :: C\rangle &\rightarrow \langle (\lambda x.e) :: S, C\rangle \\
\langle S, (e\, e') :: C\rangle &\rightarrow \langle S, e :: e' :: \mathbf{appop} :: C\rangle \\
\langle e :: (\lambda x.e') :: S, \mathbf{appop} :: C\rangle &\rightarrow \langle S, e'[x := e] :: C\rangle \\
\langle c_1 :: c_2 :: S, \mathbf{appop} :: C\rangle &\rightarrow \langle \mathrm{Constapply}(c_2, c_1) :: S, C\rangle
\end{aligned}$$

appop is here an internal constant in the machine and :: is the infix cons-operator on lists (stacks).

To evaluate an expression e we start by the initial configuration $\langle \mathrm{nil}, e :: \mathrm{nil}\rangle$, then we use the transition relation "as long as possible", that is until we reach a terminal configuration $\langle e :: S, \mathrm{nil}\rangle$. Configurations of this form is the only configurations for which the transition relation is not applicable. The result of the evaluation is the value e which is on the top of the stack.

Let us see what we have done. We have, compared to the ordinary reduction relation for λ-calculus defined a *reduction order*, call-by-value, and expressed that no reductions should be performed inside λ:s.

It is very easy to implement this semantics on a machine with a substitution operation, the semantics is also easy to understand if we take the meaning of substitution as intuitively understood. The problem, however, is that there are hardly no machines with a substitution operation and that most people, at least after a while, thinks that the substitution operation is too complicated to take as a primitive.

Another problem is that we in an interpreter often want to perform the substitution in a "lazy" manner. We do not want to perform substitutions if we do not need the result. This could of course be viewed as just a clever implementation of substitution which could be proved sound according to the semantics given. However if we are interested in the semantics in detail, for example to estimate the resources in terms of memory and time needed to evaluate a certain program, we cannot abstract away from details of how the substitution is actually performed. So in order to define also the semantics of substitution, we have to define the semantics in a more complicated way.

Exercise 13 *Use the* SC *machine to evaluate the expression* $(((\lambda x.x)\,(\lambda x.x))\,3)$.

4.1.2 The SECD machine

We start by defining what we mean by a *closure* and an *environment.*

Definition 3 *If* x_1, x_2, ..., x_n *are different variables and* Cl_1, Cl_2, ..., Cl_n *are closures then*

$$\{\langle x_1, Cl_1\rangle, \langle x_2, Cl_2\rangle \ldots, \langle x_n, Cl_n\rangle\}$$

is an **environment** *(a finite function from variables to closures)*

If E *is an environment and* e *is an expression such that* $FV(e) \subseteq \mathrm{dom}(E)$[1] *then*

$$\langle e, E\rangle$$

[1] $FV(e)$ denotes the free variables of e and $\mathrm{dom}(\{\langle x_1, Cl_1\rangle, \ldots, \langle x_n, Cl_n\rangle\}) = \{x_1, \ldots, x_n\}$

is a **closure** *(the "value" of a λ-expression).*

We will use the following notation

$E(x_i)$ *for the closure* Cl_i *when* E *is the environment* $\{\langle x_1, Cl_1\rangle, \ldots \langle x_n, Cl_n\rangle\}$
$E[x \mapsto Cl]$ *for the environment* E' *such that*

$$E'(y) = \begin{cases} Cl & \text{if } x \equiv y \\ E(y) & \text{otherwise} \end{cases}$$

In the SECD semantics a configuration consists of four components

S a value stack,

E a list of environments,

C a list of unevaluated expressions,

D a dump intuitively consisting of a list of $\langle S, E, C\rangle$ triples that describe suspended computations. We will however define it directly by $D ::= \text{nil} \mid \langle S, E, C, D\rangle)$

The semantics is now defined by the following transition relation:

$$\begin{array}{rcl}
\langle Cl :: S, E, \text{nil}, \langle S', E', C', D'\rangle\rangle & \rightarrow & \langle Cl :: S', E', C', D'\rangle \\
\langle S, E, x :: C, D\rangle & \rightarrow & \langle E(x) :: S, E, C, D\rangle \\
\langle S, E, c :: C, D\rangle & \rightarrow & \langle\langle c, E\rangle :: S, E, C, D\rangle \\
\langle S, E, (\lambda x.e) :: C, D\rangle & \rightarrow & \langle\langle \lambda x.e, E\rangle :: S, E, C, D\rangle \\
\langle S, E, (e\, e') :: C, D\rangle & \rightarrow & \langle S, E, e' :: e :: \mathbf{appop} :: C, D\rangle \\
\langle\langle c', E'\rangle :: \langle c'', E''\rangle :: S, E, \mathbf{appop} :: C, D\rangle & \rightarrow & \langle \text{Constapply}(c', c'') :: S, E, C, D\rangle \\
\langle\langle \lambda x.e, E'\rangle :: Cl :: S, E, \mathbf{appop} :: C :: D\rangle & \rightarrow & \langle \text{nil}, E'[x \mapsto Cl], e :: \text{nil}, \langle S, E, C, D\rangle\rangle
\end{array}$$

We define the functions:

$$\begin{array}{rcl}
\text{Real} & \in & \text{Closure} \rightarrow \text{Expr} \\
\text{Real}(\langle e, E\rangle) & = & e[x_1 := \text{Real}(E(x_1))] \cdots e[x_n := \text{Real}(E(x_n))] \\
 & & \text{where } \{x_1, \ldots, x_n\} = \text{FV}(e) \\
 & & \\
\text{Load} & \in & \text{Expr} \rightarrow \text{Dump} \\
\text{Unload} & \in & \text{Dump} \rightarrow \text{Expr} \\
\text{Load}(e) & = & \langle \text{nil}, \{\}, e :: \text{nil}, \text{nil}\rangle \\
\text{Unload}(\langle Cl :: C, E, \text{nil}, \text{nil}\rangle) & = & \text{Real}(Cl)
\end{array}$$

Now we can define the *semantic function* for Expr:

$$\mathcal{E}(e) = \begin{cases} e' & \text{if there is a } D \text{ such that } \text{Load}(e) \rightarrow^* D \text{ and } e' = \text{Unload}(D) \\ \perp^2 & \text{otherwise} \end{cases}$$

[2] we use $\perp$ to denote that the function is undefined

We call these definitions *iterative* since the transition relation is defined just by a collection of primitive clauses, and evaluation is just the transitive closure of the transition relation. An important property of such a semantical description is that a machine does not need more memory than the memory needed to store one configuration. An evaluation is simply a sequence of configurations

$$\gamma_1 \to \gamma_2 \to \cdots \to \gamma_n$$

such that γ_n is in T, i.e. is a final configuration.

There are several examples of languages and machines defined in this way. Gordon Plotkin defines the semantics of an imperative language in [35] by using the SMC-machine (State, Memory and Control). Another example is the G-machine [19] which is a graph reduction machine for lazy functional languages. We conclude the description of iterative operational semantics by stating a few problems:

- The computation steps are fixed and often too small.
- The definition is not always inductive over the syntactical definition of the programming language (the definition contains internal operations such as **appop**).

The second item is clearly a problem; if the first item is a problem or not depends on what level you want the description to be on. If you want the description to be as abstract as possible it is a problem.

Exercise 14 *Evaluate the expression* $((\lambda x.x)\,3)$ *using the* SECD*-machine.*

4.2 Structural (Recursive) Operational Semantics

Another way to define the semantics by an operational method is, at least if the semantics is intended for human beings, to use the Structural Operational Semantics introduced by Gordon Plotkin in [35], or a variant which is based on the way the semantics of Martin-Löf's type theory [32] is given. Another source of inspiration is the definition of the reduction relation in the lambda calculus [8]. This kind of semantical definition is very suitable to be, more or less, automatically implemented in Prolog [13, 21, 41]

4.2.1 Using a one step evaluation relation

The semantics is defined as a terminating transition system, but the rules for the transition system is given as inference rules instead of as axioms. This leads to a more comprehensible definition – the difference between iterative and structural operational semantics definitions is similar to the difference between Hilbert style formalizations of predicate logic compared to formalizations by natural deduction.

We will define the semantics of a simple imperative languages. The syntactical categories are

- expressions,
- conditions and
- commands.

We do not give a complete syntactical description here, but define the different categories together with the semantical rules.

Semantics of expressions

Let us describe the semantics of simple expressions

$$e \quad ::= \quad m \quad | \quad e_1\texttt{+}e_2$$

where $e \in \text{Exp}, m \in \text{IntConst}$

We would like to express the semantics by saying something like:

- an integer constant m is already evaluated
- a sum $e_1\texttt{+}e_2$ is evaluated by evaluating e_1 to yield an integer constant m_1, then evaluating e_2 to yield an integer constant m_2, and finally add m_1 and m_2 which gives m as result

We can formalize this by the three rules:

$$\frac{e_1 \to e_1'}{e_1\texttt{+}e_2 \to e_1'\texttt{+}e_2}\ \text{L+}$$

$$\frac{e_2 \to e_2'}{m_1\texttt{+}e_2 \to m_1\texttt{+}e_2'}\ \text{R+}$$

$$\frac{}{m_1\texttt{+}m_2 \to m}\ \text{V+} \qquad (\text{if } m_1 + m_2 = m)$$

The last rule should perhaps be written

$$m_1\texttt{+}m_2 = m$$

$$\begin{array}{ll} \text{where} & \phi(m_1) + \phi(m_2) = m' \\ \text{and} & m = \phi^{-1}(m') \end{array}$$

and where ϕ is the interpretation function for integer constants, mapping (syntactical) numerals to (semantic) numbers. We shall in the sequel not distinguish syntactic constants from mathematical constants and freely use mathematical operations on syntactical objects such as m_1 and m_2 in the rule above.

Example: Using the rules we could evaluate the simple expression `(1+2)+(3+2)` by using the rules for expressions and the transitivity of the transition relation.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{(1+2)} \to \texttt{3}}\ \text{V+}}{\texttt{(1+2)+(3+2)} \to \texttt{3+(3+2)}}\ \text{L+}}{\texttt{(1+2)+(3+2)} \to^* \texttt{3+(3+2)}}\ *1 \qquad \dfrac{\texttt{(3+2)} \to \texttt{5}}{\texttt{3+(3+2)} \to \texttt{3+5}}\ \text{R+}}{\texttt{(1+2)+(3+2)} \to^* \texttt{3+5}}\ *2 \qquad \dfrac{}{\texttt{3+5} \to \texttt{8}}\ \text{V+}}{\texttt{(1+2)+(3+2)} \to^* \texttt{8}}\ *2$$

where we in some inferences have used the transitivity rules for $\to$.

$$\frac{\gamma_1 \to \gamma_2}{\gamma_1 \to^* \gamma_2}\ *1$$

$$\frac{\gamma_1 \to^* \gamma_2 \qquad \gamma_2 \to \gamma_3}{\gamma_1 \to^* \gamma_3}\ *2$$

We can also see that the evaluation is not "flat" any more.

Another way of formulating the transitivity rule is

$$\frac{\gamma_1 \to \gamma_2 \qquad \gamma_2 \to \gamma_3 \qquad \cdots \qquad \gamma_{n-2} \to \gamma_{n-1} \qquad \gamma_{n-1} \to \gamma_n}{\gamma_1 \to^* \gamma_n}\ *$$

Using this formulation our proof is written:

$$\frac{\dfrac{\dfrac{}{\texttt{(1+2)} \to \texttt{3}}\ \text{V+}}{\texttt{(1+2)+(3+2)} \to \texttt{3+(3+2)}}\ \text{L+} \qquad \dfrac{\texttt{(3+2)} \to \texttt{5}}{\texttt{3+(3+2)} \to \texttt{3+5}}\ \text{R+} \qquad \dfrac{}{\texttt{3+5} \to \texttt{8}}\ \text{V+}}{\texttt{(1+2)+(3+2)} \to^* \texttt{8}}\ *$$

□

If we introduce more complicated expressions with variables, the configurations also have to become more complicated. Let us consider expressions

$$e \quad ::= \quad m \quad | \quad x \quad | \quad e\texttt{+}e \quad | \quad e\texttt{-}e \quad | \quad e\texttt{*}e$$

where $e \in \text{Exp}, m \in \text{IntConst}, x \in \text{Var}$

To give an operational semantics we have to introduce a component in the configuration which associates values with variables. The set of configurations is defined as

$$\Gamma \quad = \quad \text{Expr} \times \text{Mem}$$

where $\text{Mem} \quad = \quad \text{Var} \to \text{Int}$

We will use the operations

$$\begin{aligned} \text{update} &\in \text{Var} \times \text{Int} \times \text{Mem} \to \text{Mem} \\ \text{valof} &\in \text{Var} \times \text{Mem} \to \text{Int} \end{aligned}$$

which updates a memory with a new variable/value association and retrieves the value of a variable, respectively. We also use the constant memory, σ_0, which associates the value 0 with all variables.

$$\begin{aligned} \text{valof}(x, \sigma_0) &= 0 \\ \text{valof}(x, \text{update}(x, n, \sigma)) &= n \\ \text{valof}(y, \text{update}(x, n, \sigma)) &= \text{valof}(x, \sigma) \qquad (\text{if } x \not\equiv y) \end{aligned}$$

The transition relation is

$$\langle e, \sigma \rangle \quad \to \quad \langle e', \sigma \rangle$$

where $e \in \text{Exp}, \sigma \in \text{Mem}$

which we also can write as

$$\to \quad \subseteq \quad (\text{Expr} \times \text{Mem}) \times (\text{Expr} \times \text{Mem})$$

or, with some notational abuse,

$$(\text{Expr} \times \text{Mem}) \to (\text{Expr} \times \text{Mem})$$

Notice that the arrow ($\rightarrow$) does not mean the function space here, but the one step transition relation itself!! The final configurations have the form

$$\langle m, \sigma \rangle \in \text{IntConst} \times \text{Mem}$$

In the semantic definition it is always the same memory to the left and to the right of the arrow since evaluating an expression does not change the memory! The rules defining the semantics are:

m:

There are no rules for evaluating an integer constant because a constant is considered to be evaluated.

x:

$$\frac{}{\langle x, \sigma \rangle \rightarrow \langle \text{valof}(x, \sigma), \sigma \rangle} \text{ var}$$

e_1+e_2:

$$\frac{\langle e_1, \sigma \rangle \rightarrow \langle e_1', \sigma \rangle}{\langle e_1 \texttt{+} e_2, \sigma \rangle \rightarrow \langle e_1' \texttt{+} e_2, \sigma \rangle} \text{ L+}$$

$$\frac{\langle e_2, \sigma \rangle \rightarrow \langle e_2', \sigma \rangle}{\langle m_1 \texttt{+} e_2, \sigma \rangle \rightarrow \langle m_1 \texttt{+} e_2', \sigma \rangle} \text{ R+}$$

$$\frac{}{\langle m_1 \texttt{+} m_2, \sigma \rangle \rightarrow \langle m, \sigma \rangle} \text{ V+} \qquad \text{where } m_1 + m_2 = m$$

e_1-e_2:

$$\frac{\langle e_1, \sigma \rangle \rightarrow \langle e_1', \sigma \rangle}{\langle e_1 \texttt{-} e_2, \sigma \rangle \rightarrow \langle e_1' \texttt{-} e_2, \sigma \rangle} \text{ L-}$$

$$\frac{\langle e_2, \sigma \rangle \rightarrow \langle e_2', \sigma \rangle}{\langle m_1 \texttt{-} e_2, \sigma \rangle \rightarrow \langle m_1 \texttt{-} e_2', \sigma \rangle} \text{ R-}$$

$$\frac{}{\langle m_1 \texttt{-} m_2, \sigma \rangle \rightarrow \langle m, \sigma \rangle} \text{ V-} \qquad \text{if } m_1 \geq m_2 \text{ and } m = m_1 - m_2$$

We say that a configuration is *stuck* if it is not a final configuration and there is no rule that is applicable for evaluating the configuration. When we considering the evaluation of expressions, this means that we can make the following definition.

Definition 4 *A configuration $\langle e, \sigma \rangle$ is* stuck *if $e \not\equiv m$ and there is no configuration γ such that $\langle e, \sigma \rangle \rightarrow \gamma$*

A simple example of a stuck configuration is $\langle 5 + (2 - 3), \sigma \rangle$. The computational intuition behind a stuck configuration is that some kind of "run time error" has occurred during

the computation. A more explicit way is to see the error as something natural and specify the behavior explicitly by a rule such as

$$\frac{}{\langle m_1\texttt{-}m_2, \sigma\rangle \to \texttt{error}}\ \text{V-} \qquad \text{if } m_1 \leq m2$$

We have now defined rules that determines how an expression is evaluated in a memory σ, or, in other words, we have defined a transition relation on configurations. From this relation we can define a (partial) function that defines the behavior (semantics) of expressions

$$\mathcal{E}(e) \quad = \quad \begin{cases} \lambda\sigma.m & \text{if } \langle e, \sigma\rangle \to^* \langle m, \sigma\rangle \\ \bot & \text{otherwise} \end{cases}$$

We can also define a semantic equality between expressions

$$e_1 \ \approx_E \ e_2 \iff \forall\sigma.\mathcal{E}(e_1) \ \cong \ \mathcal{E}(e_2)$$

where $e \cong e'$ if and only if both are undefined or if both are defined and have equal values.

Exercise 15 *Define the rules for $e_1\texttt{*}\,e_2$ in the style of $e_1\texttt{+}\,e_2$ above.*

Exercise 16 *We said above that the memory to the left and to the right of the arrow is always the same in the evaluation of expressions. Prove that this is true.*

Semantics of conditions (boolean expressions):

Let us assume we have the following abstract syntax for conditions:

$$b \quad ::= \quad t \quad | \quad e\texttt{=}e \quad | \quad b\ \texttt{or}\ b \quad | \quad \texttt{not}\ b$$

where $b \in$ Bexp, $t \in$ BoolConst, $e \in$ Exp

We define the semantics in the same way as for ordinary expressions. The transition relation has the "type"

$$\text{Bexp} \times \text{Mem} \to \text{Bexp} \times \text{Mem}$$

and the final configurations

$$\text{BoolConst} \times \text{Mem}$$

We are, strictly speaking, defining one transition relation for each different syntactical category. So, we should really use different symbols, $\to_{Expr}$, for expressions and $\to_{Bexp}$ for conditions. This, however, leads to a too heavy notational burden, so we are a bit sloppy and use the "overloaded" symbol $\to$ for all relations.

There are no rules for constants since they are considered evaluated conditions. The evaluation relation is defined for = by the following rules:

$e_1\texttt{=}e_2$:

$$\frac{\langle e_1, \sigma\rangle \to \langle e_1', \sigma\rangle}{\langle e_1\texttt{=}e_2, \sigma\rangle \to \langle e_1'\texttt{=}e_2, \sigma\rangle}\ \text{L=}$$

$$\frac{\langle e_2, \sigma\rangle \to \langle e_2', \sigma\rangle}{\langle m_1\texttt{=}e_2, \sigma\rangle \to \langle m_1\texttt{=}e_2', \sigma\rangle}\ \text{R=}$$

$$\frac{}{\langle m_1\texttt{=}m_2, \sigma\rangle \to \langle b, \sigma\rangle}\ \text{V=} \qquad \text{where } b = m_1 = m_2\text{!!}$$

There are at least four different ways of defining the rules for the `or` operation. We could have:

1. complete evaluation,
2. left sequential evaluation,
3. right sequential evaluation, or
4. parallel evaluation.

To define complete evaluation of the parts of the expression we give the following rules (compare with the rules for `+` and `*`).

b_1 `or` b_2 (complete evaluation):

$$\frac{\langle b_1, \sigma\rangle \to \langle b_1', \sigma\rangle}{\langle b_1 \texttt{ or } b_2, \sigma\rangle \to \langle b_1' \texttt{ or } b_2, \sigma\rangle}\ \text{LorC}$$

$$\frac{\langle b_2, \sigma\rangle \to \langle b_2', \sigma\rangle}{\langle t_1 \texttt{ or } b_2, \sigma\rangle \to \langle t_1 \texttt{ or } b_2', \sigma\rangle}\ \text{RorC}$$

$$\frac{}{\langle t_1 \texttt{ or } t_2, \sigma\rangle \to \langle t, \sigma\rangle}\ \text{VorC} \qquad \text{where } t = t_1 \vee t_2$$

If we want left- (or right-) sequential operation we use the following rules:

b_1 `or` b_2 (left sequential evaluation):

$$\frac{\langle b_1, \sigma\rangle \to \langle b_1', \sigma\rangle}{\langle b_1 \texttt{ or } b_2, \sigma\rangle \to \langle b_1' \texttt{ or } b_2, \sigma\rangle}\ \text{LorLs}$$

$$\frac{}{\langle \text{true} \texttt{ or } b_2, \sigma\rangle \to \langle \text{true}, \sigma\rangle}\ \text{RorLs}$$

$$\frac{}{\langle \text{false} \texttt{ or } b_2, \sigma\rangle \to \langle b_2, \sigma\rangle}\ \text{VorLs}$$

One example where the complete and the left sequential evaluation would give different results is `true or (2-3)=0`.

To define that the two expressions should be evaluated in parallel, or, in other words, to demand that the complete expression should have a value if either of the two operands evaluates to true, we can give the following rules:

b_1 `or` b_2 (parallel evaluation):

$$\frac{\langle b_1, \sigma\rangle \to \langle b_1', \sigma\rangle}{\langle b_1 \texttt{ or } b_2, \sigma\rangle \to \langle b_1' \texttt{ or } b_2, \sigma\rangle}\ \text{LorP}$$

$$\frac{\langle b_2, \sigma\rangle \to \langle b_2', \sigma\rangle}{\langle b_1 \texttt{ or } b_2, \sigma\rangle \to \langle b_1 \texttt{ or } b_2', \sigma\rangle}\ \text{RorP}$$

$$\frac{}{\langle \text{true} \texttt{ or } b_2, \sigma\rangle \to \langle \text{true}, \sigma\rangle}\ \text{V1orP}$$

$$\frac{}{\langle b_1 \texttt{ or } \text{true}, \sigma\rangle \to \langle \text{true}, \sigma\rangle}\ \text{V2orP}$$

$$\frac{}{\langle \text{false} \texttt{ or } b_2, \sigma\rangle \to \langle b_2, \sigma\rangle}\ \text{V3orP}$$

$$\frac{}{\langle b_1 \texttt{ or } \text{false}, \sigma\rangle \to \langle b_1, \sigma\rangle}\ \text{V4orP}$$

Exercise 17 *Can you say when the parallel evaluation would be different from the left and right sequential rules, respectively? Is it possible to define a function in Pascal or ML which implements the* `or`*-operation defined by the parallel rules?*

Semantics of commands:

In the rules we have given above, the memory never changed. To define the rules for commands this can no longer hold — the purpose of assignment command, for example, is solely to change the memory.

We assume we have the following abstract syntax for commands:

$$\begin{array}{lcl} c & ::= & \texttt{skip} \\ & | & x\texttt{:=}e \\ & | & c\texttt{;}c \\ & | & \texttt{if } b \texttt{ then } c \texttt{ else } c \\ & | & \texttt{while } b \texttt{ do } c \end{array}$$

where $c \in \text{Com}$, $v \in \text{Var}$, $e \in \text{Expr}$ and $b \in \text{Bexp}$

The type of the transition relation for commands is

$$\text{Com} \times \text{Mem} \to \text{Com} \times \text{Mem} \cup \text{Mem}$$

saying that the result of evaluating a command one step is either a configuration consisting of a command and a memory or just a memory. The final configurations are the ones consisting of just the memory.

The rules are as follows:

`skip`:

$$\frac{}{\langle \texttt{skip}, \sigma\rangle \to \sigma}\ \text{skip}$$

$x\texttt{:=}e$:

$$\frac{\langle e, \sigma\rangle \to^* \langle m, \sigma\rangle}{\langle x\texttt{:=}e, \sigma\rangle \to \text{update}(x, m, \sigma)}\ :=$$

$c_1\texttt{;}c_2$:

$$\frac{\langle c_1, \sigma\rangle \to \langle c_1', \sigma\rangle}{\langle c_1\texttt{;}c_2, \sigma\rangle \to \langle c_1'\texttt{;}c_2, \sigma\rangle}\ ;1$$

$$\frac{\langle c_1, \sigma\rangle \to \sigma'}{\langle c_1\texttt{;}c_2, \sigma\rangle \to \langle c_2, \sigma'\rangle}\ ;2$$

`if` b `then` c_1 `else` c_2:

$$\frac{\langle b, \sigma\rangle \rightarrow^* \langle \text{false}, \sigma\rangle}{\langle \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma\rangle \rightarrow \langle c_2, \sigma\rangle}\ \text{if-false}$$

$$\frac{\langle b, \sigma\rangle \rightarrow^* \langle \text{true}, \sigma\rangle}{\langle \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma\rangle \rightarrow \langle c_1, \sigma\rangle}\ \text{if-true}$$

`while` b `do` c:

$$\frac{\langle b, \sigma\rangle \rightarrow^* \langle \text{false}, \sigma\rangle}{\langle \texttt{while}\ b\ \texttt{do}\ c, \sigma\rangle \rightarrow \sigma}\ \text{while-false}$$

$$\frac{\langle b, \sigma\rangle \rightarrow^* \langle \text{true}, \sigma\rangle}{\langle \texttt{while}\ b\ \texttt{do}\ c, \sigma\rangle \rightarrow \langle c; \texttt{while}\ b\ \texttt{do}\ c, \sigma\rangle}\ \text{while-true}$$

Exercise 18 *For expressions and conditions the final configurations were just a subset of the ordinary configurations – the result of evaluating an expression is an integer constant and a store where the integer constant is just a special case of an expression. When evaluating an expression the memory is just a "passive" (read-only) component.*

For commands the situation is a bit different – the result is not a command, and the memory is certainly not passive. However the `skip` *command behaves a bit like evaluating an integer constant. Use this fact in order to define a transition relation*

$$\text{Com} \times \text{Mem} \rightarrow_1 \text{Com} \times \text{Mem}$$

with final configurations of the form $\langle \texttt{skip}, \sigma\rangle$.

Exercise 19 *If we use the rules to implement an interpreter for the language the rules for both the if-statement and the while-statement are not directly suitable since there are two rules applicable given the first part of the conclusion. If we want to evaluate the configuration* $\langle \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma\rangle$ *both the rules if-false and if-true are applicable. A theorem prover (an interpreter) must choose one of the rules and try to prove the premise. If the premise could not be proved the interpreter must backtrack and try the other rule. This behaviour is a source of inefficiency since the condition of the command could be evaluated twice. Can you define the rules in a way which causes the condition to be evaluated exactly once.*

One advantage with Plotkin's structural operational semantics compared with the iterative semantics used in the SC and SECD machines is that it is structural. The computation of a syntactic construction is defined in terms of the computations of its syntactical parts. Another advantage is that the evaluation steps defined by the transition relation are a bit more flexible than the steps defined in the iterative semantics.

4.2.2 Operational semantics with a value-relation

An alternative to the terminal transitions systems, which we have used above, for defining the semantics is to define a relation between *configurations* and *values*.

$$\Rightarrow \quad \subseteq \quad \Gamma \times \text{Value}$$

where $\gamma \Rightarrow v$ means that v is the result of evaluating γ

We have already seen examples of such a relation when we have used $\rightarrow^*$ in the semantic rules for assignment, if-command and while command. As Plotkin says in [35]:

> "A point to watch is to make a distinction between *internal* and *external* behavior. Internally a system's behavior is nothing but the sum of its transitions. (We ignore here the fact that often these transitions make sense only at a certain level; what counts as one transition for one purpose may in fact consist of many steps when viewed in more detail. Part of the spirit of our method is to choose steps of the appropriate "size".)

In some situations it is convenient to have small computation steps, for example if you want to study nonterminating computations within the formal system. But if you are only interested in the result of the computation, it is often more convenient to take a big step and only study the value relation $\Rightarrow$. We give a possibility to compare the two methods by giving a semantical description of the same language as we defined by Plotkin's method above.

Expressions:

Grammar:

$e \quad ::= \quad m \quad | \quad x \quad | \quad e\texttt{+}e \quad | \quad e\texttt{-}e \quad | \quad e\texttt{*}e$

where $e \in \text{Exp}, m \in \text{IntConst}, x \in \text{Var}$

Transition relation:

$\text{Exp} \times \text{Mem} \Rightarrow \text{Int}$

Semantical rules:

m:

$$\langle m, \sigma \rangle \Rightarrow m$$

x:

$$\langle x, \sigma \rangle \Rightarrow \text{valof}(x, \sigma)$$

$e_1\texttt{+}e_2$:

$$\frac{\langle e_1, \sigma \rangle \Rightarrow m_1 \quad \langle e_2, \sigma \rangle \Rightarrow m_2}{\langle e_1\texttt{+}e_2, \sigma \rangle \Rightarrow m} \qquad \text{where } m = m_1 + m_2$$

$e_1\texttt{-}e_2$:

$$\frac{\langle e_1, \sigma \rangle \Rightarrow m_1 \quad \langle e_2, \sigma \rangle \Rightarrow m_2}{\langle e_1\texttt{-}e_2, \sigma \rangle \Rightarrow m} \qquad \text{where } m_1 \geq m_2 \text{ and } m = m_1 - m_2$$

Exercise 20 *Define the rules for $e_1\texttt{*}e_2$ in the same style.*

Conditions:

Grammar:

b ::= t | e=e | b `or` b | `not` b

where $b \in$ Bexp, $t \in$ Boolconst, $e \in$ Exp

Transition relation:

$$\text{Bexp} \times \text{Mem} \Rightarrow \text{Bool}$$

Semantical rules:

e_1=e_2:

$$\frac{\langle e_1, \sigma\rangle \Rightarrow m_1 \quad \langle e_2, \sigma\rangle \Rightarrow m_2}{\langle e_1\texttt{=}e_2, \sigma\rangle \Rightarrow t} \qquad \text{where } t = m_1 == m_2$$

b_1 `or` b_2:

$$\frac{\langle b_1, \sigma\rangle \Rightarrow t_1 \quad \langle b_2, \sigma\rangle \Rightarrow t_2}{\langle b_1 \texttt{ or } b_2, \sigma\rangle \Rightarrow t} \qquad \text{where } t = t_1 \vee t_2$$

If we want a left- (or right-) sequential operation we use the following rules:

b_1 `or` b_2 (left sequential evaluation):

$$\frac{\langle b_1, \sigma\rangle \Rightarrow \text{true}}{\langle b_1 \texttt{ or } b_2, \sigma\rangle \Rightarrow \text{true}}$$

$$\frac{\langle b_1, \sigma\rangle \Rightarrow \text{false} \qquad \langle b_2, \sigma\rangle \Rightarrow t}{\langle b_1 \texttt{ or } b_2, \sigma\rangle \Rightarrow t}$$

which says that if the first operand evaluates to true, then it is no need to evaluate the second operand.

Exercise 21 *Define the rules for* `not` b .

Exercise 22 *How should the* `or`*-rules be formalized to get (or to allow) parallel evaluation of the operands?*

Commands:

Grammar:

c ::= `skip`
| x`:=`e
| c`;`c
| `if` b `then` c `else` c
| `while` b `do` c

where $c \in$ Com, $v \in$ Var, $e \in$ Expr and $b \in$ Bexp

Transition relation:

$$\text{Com} \times \text{Mem} \Rightarrow \text{Mem}$$

Semantical rules:

`skip`:

$$\langle \texttt{skip}, \sigma \rangle \Rightarrow \sigma$$

x`:=`e:

$$\frac{\langle e, \sigma \rangle \Rightarrow m}{\langle x \texttt{:=} e, \sigma \rangle \Rightarrow \text{update}(x, m, \sigma)}$$

c_1`;`c_2:

$$\frac{\langle c_1, \sigma \rangle \Rightarrow \sigma' \quad \langle c_2, \sigma' \rangle \Rightarrow \sigma''}{\langle c_1 \texttt{;} c_2, \sigma \rangle \Rightarrow \sigma''} \ 1$$

`if` b `then` c_1 `else` c_2:

$$\frac{\langle b, \sigma \rangle \Rightarrow \text{false} \quad \langle c_2, \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2, \sigma \rangle \rightarrow \sigma'} \ \text{IF-false}$$

$$\frac{\langle b, \sigma \rangle \Rightarrow \text{true} \quad \langle c_1, \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2, \sigma \rangle \rightarrow \sigma'} \ \text{IF-true}$$

`while` b `do` c:

$$\frac{\langle b, \sigma \rangle \Rightarrow \text{false}}{\langle \texttt{while } b \texttt{ do } c, \sigma \rangle \Rightarrow \sigma} \ \text{W-false}$$

$$\frac{\langle b, \sigma \rangle \Rightarrow \text{true} \quad \langle c, \sigma \rangle \Rightarrow \sigma' \quad \langle \texttt{while } b \texttt{ do } c, \sigma' \rangle \Rightarrow \sigma''}{\langle \texttt{while } b \texttt{ do } c, \sigma \rangle \Rightarrow \sigma''} \ \text{W-true}$$

4.2.3 Input and Output

The programming language we have given meaning to above is very simple and needs to be extended in order to be "realistic". First let us extend it with input and output. Let us assume the language has two more commands, one to input things and one to output things.

$$c \quad ::= \quad \cdots \quad | \quad \texttt{read } x \quad | \quad \texttt{write } e$$

Let us also introduce a special domain of programs separate from the commands in order to assign have different transition relations for programs and commands.

Programs and Commands:

The grammar for programs is quite simple

$$p \quad ::= \quad \texttt{prog}\ c\ \texttt{end}$$

where $p \in \text{Pgm}$ and $c \in \text{Com}$

A program takes a list of integers as input and returns a list of input as result, therefore the transition relation has the type

$$\text{Pgm} \times \text{List(Int)} \Rightarrow \text{List(Int)}$$

The semantic rule is something like

$$\frac{\langle c, \sigma\rangle \Rightarrow \sigma'}{\langle \texttt{prog}\ c\ \texttt{end}, in\rangle \Rightarrow out}$$

In order for this rule to make sense we must say what σ is in terms of in and what out is in terms of σ'. The "memory" must be extended with components for the input and output lists. We call this new memory for a state

$$\text{State} \ = \ \text{Mem} \times \text{List(Int)} \times \text{List(Int)}$$

and define two functions, one to create a state from a list of integers and one that extracts a list from a state.

$$\begin{aligned} \text{mkin} &\in \text{List(Int)} \to \text{State} \\ \text{mkin(in)} &= \langle \sigma_0, in, [\,]\rangle \\ \\ \text{mkout} &\in \text{State} \to \text{List(Int)} \\ \text{mkout}(\langle \sigma, in, out\rangle) &= out \end{aligned}$$

It is now possible to make the definitions we searched for above and we can rewrite the semantic rule for programs

$$\frac{\langle c, \text{mkin}(in)\rangle \Rightarrow \sigma'}{\langle \texttt{prog}\ c\ \texttt{end}, in\rangle \Rightarrow \text{mkout}(\sigma')}\ \text{Pgm}$$

The only thing that remains for this to be perfect is to decide what the initial memory σ_0 should be. We can choose to initially map variables to an error element, we can let the initial memory be undefined, or we can choose to map every variable to a predefined value, for example 0.

In order to complete the definition we must, of course, define the semantic rules for the new commands and say how the transition relation should be modified for the old commands.

Semantic rules:

$$\frac{}{\langle \texttt{read}\ x, \langle \sigma, i :: is, os\rangle\rangle \Rightarrow \langle \text{update}(x, i, \sigma), is, os\rangle}\ \text{read}$$

$$\frac{\langle e, \sigma\rangle \Rightarrow n}{\langle \texttt{write}\ e, \langle \sigma, is, os\rangle\rangle \Rightarrow \langle \sigma, is, n :: os\rangle}\ \text{write}$$

Exercise 23 *Update the semantical rules for the old commands using the state instead of the memory.*

Exercise 24 *Could you see some way of introducing a* `stop`*-command that halts the evaluation. How should the semantics be defined?*

Hint: introduce a Boolean component in the configuration that determines if a command should have its usual semantics or should have the `skip`*-semantics.*

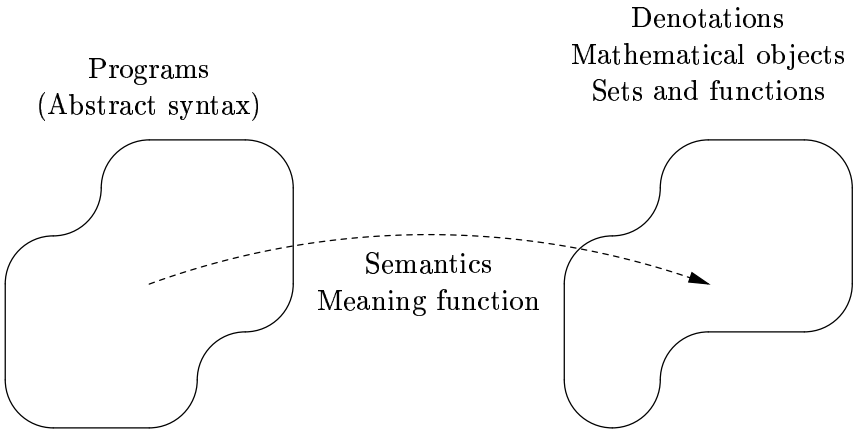
Use the same technique to introduce the `break` *and* `continue` *commands of C.*

4.3 Denotational Semantics

As a reaction to the operational methods of defining semantics in the 60:ies, a method called *denotational semantics* (or sometimes *mathematical semantics*) arouse in the late 60:ies and early 70:ies. The starting point was the opinion that the operational descriptions contained to much information which was related to the implementation of the language. To get away from machines and implementations, one tried to introduce more mathematics and mathematical methods.

The philosophy behind denotational semantics can briefly be summarized in two statements:

1. The semantics of a programming language is a mathematical function that maps every program on the mathematical object which is the denotation (meaning) of the program.



The semantics is a mathematical function from programs to denotations

2. The semantics of a compound syntactical construction is always a combination of the semantics of its parts.

Some program constructions have a natural mathematical meaning, (for example, `1`, `2`, ..., `+`, `*`) but some are hard to view as mathematical objects (recursion, higher order functions with selfapplication, non-terminating constructions, goto:s, ...). The second statement could be compared with Frege's principle for natural languages (1890?):

> The meaning of a sentence is completely determined by the semantics of its parts.

This means that you can replace one part of a sentence with another with the same semantics without affecting the semantics of the complete sentence.

Let us start with a simple example that does not cause any mathematical problems at all. We first define the abstract syntax of the language

variables	$x \in \text{Var}$
numerals	$n \in \text{Num}$
boolean constants	$z \in \text{Bool} = \{\text{true}, \text{false}\}$
expressions	$e \in \text{Expr}$
	$e ::= x \mid n \mid e\texttt{+}e \mid e\texttt{*}e$
conditions	$b \in \text{Bexp}$
	$b ::= t \mid \texttt{not}\ b \mid b\ \texttt{or}\ b \mid b\ \texttt{and}\ b$
commands	$c \in \text{Com}$
	$c ::= x\texttt{:=}e \mid \texttt{skip} \mid c\ \texttt{;}\ c \mid \texttt{if}\ b\ \texttt{then}\ c\ \texttt{else}\ c$

A denotational semantics for this language consists of a *meaning function* for each syntactical category. The meaning function is a usual mathematical function and we use ordinary mathematical notation to define it. But before we can define the meaning functions we must define the mathematical objects that should be the "semantic values", i.e. the codomains of the meaning functions. We call these *semantical domains* and the complexity of these domains depends on the complexity of the language we want to give meaning to. For our very simple language it is enough with ordinary mathematical sets, but more advanced structures are soon needed. For a start, we introduce the following semantic domains:

Var	the set of variables
N	the set of natural number
T	the set of truth values, $\{\text{tt}, \text{ff}\}$

We also define a set of states:

$\text{S} = \text{Var} \to \text{N}$ the set of total functions from variables to the natural numbers

We are now in a position where we can define all meaning functions. For the primitive syntactical categories we have the following functions:

$\mathcal{X}$	$\in$	$\text{Var} \to \text{Var}$	$=$	id_{Var}
$\mathcal{N}$	$\in$	$\text{Num} \to \text{N}$		maps a numeral to the corresponding natural number
$\mathcal{T}$	$\in$	$\text{Bool} \to \text{T}$		similar to $\mathcal{N}$

These meaning functions are trivial and are often excluded from semantical definitions. For compound syntactical categories, the meaning functions are more interesting. The meaning of an expression may depend on the values of the variables in the expression. Therefore the meaning of an expression is a total function from states to natural numbers

$\mathcal{E} \in \text{Expr} \to (\text{S} \to \text{N})$

Similarly, the meaning function for conditions (boolean expression) has the following functionality:

$$\mathcal{B} \in \text{Bexp} \to (\text{S} \to \text{T})$$

and for commands, we have:

$$\mathcal{C} \in \text{Com} \to (\text{S} \to \text{S})$$

Compare with how we defined the configurations and the terminal configurations in the section on operational semantics.

We use λ-notation for mathematical functions and write syntactical objects within $[\![\ldots]\!]$ brackets to improve readability

$$\begin{aligned} \mathcal{E}[\![x]\!] &= \lambda\sigma.\sigma(\mathcal{X}[\![x]\!]) = \lambda\sigma.\sigma(x) \\ \mathcal{E}[\![n]\!] &= \lambda\sigma.\mathcal{N}[\![n]\!] \\ \mathcal{E}[\![e_1\texttt{+}e_2]\!] &= \lambda\sigma.\mathcal{E}[\![e_1]\!]\sigma + \mathcal{E}[\![e_2]\!]\sigma \end{aligned}$$

Note the overloading of the '+'-symbol. First it is a symbol in the programming language, then it is used to denote addition between natural numbers. We have tried to show the difference by using different fonts.

$$\mathcal{E}[\![e_1\texttt{*}e_2]\!] = \lambda\sigma.\mathcal{E}[\![e_1]\!]\sigma * \mathcal{E}[\![e_2]\!]\sigma$$

The meaning function for conditions:

$$\begin{aligned} \mathcal{B}[\![t]\!] &= \lambda\sigma.\mathcal{T}[\![t]\!] \\ \mathcal{B}[\![\texttt{not } b]\!] &= \lambda\sigma.\neg\mathcal{B}[\![b]\!]\sigma \\ \mathcal{B}[\![e_1 \texttt{ or } e_2]\!] &= \lambda\sigma.\mathcal{B}[\![e_1]\!]\sigma \vee \mathcal{B}[\![e_2]\!]\sigma \\ \mathcal{B}[\![e_1\texttt{=}e_2]\!] &= \lambda\sigma.\mathcal{E}[\![e_1]\!]\sigma = \mathcal{E}[\![e_2]\!]\sigma \end{aligned}$$

and for commands:

$$\begin{aligned} \mathcal{C}[\![\texttt{skip}]\!] &= \lambda\sigma.\sigma \\ \mathcal{C}[\![\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2]\!] &= \lambda\sigma.\text{cond}(\mathcal{B}[\![b]\!]\sigma, \mathcal{C}[\![c_1]\!]\sigma, \mathcal{C}[\![c_2]\!]\sigma) \\ \mathcal{C}[\![x\texttt{:=}e]\!] &= \lambda\sigma.\text{update}(\mathcal{X}[\![x]\!], \mathcal{E}[\![e]\!]\sigma, \sigma) \\ \mathcal{C}[\![c_1\,\texttt{;}c_2]\!] &= \mathcal{C}[\![c_2]\!] \circ \mathcal{C}[\![c_1]\!] \end{aligned}$$

where

$$\text{cond} \in \text{T} \times \text{C} \times \text{C} \to \text{C} \qquad \begin{aligned} \text{cond}(\text{ff}, c_1, c_2) &= c_2 \\ \text{cond}(\text{tt}, c_1, c_2) &= c_1 \end{aligned}$$

$$\text{update} \in \text{Var} \times \text{N} \times \text{S} \to \text{S} \qquad \text{update}(x, n, \sigma) = \begin{cases} n & \text{if } x \equiv y \\ \sigma(y) & \text{otherwise} \end{cases}$$

No problems, but that was also the intention!

4.3.1 Nonterminating program constructions

Let us see what happens if we successively extend the language. It is easy to extend the language with a for statement because nothing important happens. We can define the semantics using the same machinery that we already have introduced.

$$\mathcal{C}[\![\texttt{for } i\texttt{:=0 to } e \texttt{ do } c]\!] \;=\; \lambda\sigma.\mathrm{forstmt}(\mathcal{E}[\![e]\!]\sigma, \sigma)$$

where

$$\begin{aligned} \mathrm{forstmt}(0,\sigma) &= \mathcal{C}[\![c]\!](\mathrm{update}(i,0,\sigma)) \\ \mathrm{forstmt}(n+1,\sigma) &= \mathcal{C}[\![c]\!](\mathrm{update}(i,n+1,\mathrm{forstmt}(n,\sigma))) \end{aligned}$$

Notice that the function forstmt is defined by primitive recursion over n.

It is however not as easy to add a while statement. It is clear that the meaning of a command no longer can be a *total function* on states. For example, what total function should be the meaning of the command

```
while true do skip
```

The most natural way to treat a command like this is to say that the meaning is a partial function on states. But instead of introducing "real" partial functions, we introduce a mathematical object $\perp$ (bottom) and model a partial function on states by a total function on extended states (states together with bottom). Now we can give meaning to the non-terminating program above:

$$\mathcal{C}[\![\texttt{while true do skip}]\!] \;=\; \lambda\sigma.\perp$$

and

$$\mathcal{C} \;\in\; \mathrm{C} \to \mathrm{S}_\perp \to \mathrm{S}_\perp$$

where $\mathrm{S}_\perp \;=\; \mathrm{S} \cup \{\perp\}$

We also want the semantics of the other constructions to be bottom preserving, that is, if the state before a command is $\perp$ then it should be $\perp$ also after the command is executed. Otherwise $\perp$ would not model partial functions. The definition of the meaning function for commands becomes

$$\begin{aligned} \mathcal{C}[\![\texttt{skip}]\!] &= \lambda\sigma.\sigma \;\in\; \mathrm{S}_\perp \to \mathrm{S}_\perp \\ \mathcal{C}[\![\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2]\!] &= \lambda \;\perp \;\Rightarrow\; \perp^{3} \\ &\quad\;\; \mid\; \sigma \;\Rightarrow\; \mathrm{cond}(\mathcal{B}[\![b]\!]\sigma, \mathcal{C}[\![c_1]\!]\sigma, \mathcal{C}[\![c_2]\!]\sigma) \\ \mathcal{C}[\![x\texttt{:=}e]\!] &= \lambda \;\perp \;\Rightarrow\; \perp \\ &\quad\;\; \mid\; \sigma \;\Rightarrow\; \mathrm{update}(\mathcal{X}[\![x]\!], \mathcal{E}[\![e]\!]\sigma, \sigma) \\ \mathcal{C}[\![c_1\,;c_2]\!] &= \mathcal{C}[\![c_2]\!] \circ \mathcal{C}[\![c_1]\!] \\ \mathcal{C}[\![\texttt{while } b \texttt{ do } c\;]\!] &= \phi \end{aligned}$$

where

$$\phi(\sigma) \;=\; \begin{cases} \sigma' & \text{if } (\exists n)\ (((\forall m \le n)\mathcal{C}[\![c]\!]^m\sigma \neq \perp)\ \&\ \mathcal{C}[\![c]\!]^n\sigma = \sigma' \\ & \qquad \&\ \mathcal{B}[\![b]\!]^n\sigma' = \mathrm{ff}\ \&\ (\forall m < n)(\mathcal{B}[\![b]\!] \circ \mathcal{C}[\![c]\!])^m\sigma = \mathrm{tt}) \\ \perp & \text{otherwise} \end{cases}$$

---

[3] The notation $\begin{array}{llll} \lambda & \perp & \Rightarrow & \perp \\ \mid & \sigma & \Rightarrow & e \end{array}$ means the function that maps $\perp$ to $\perp$ and otherwise maps a state σ to e.

Notice that the meaning functions for conditions and expressions can be exactly the same as before.

4.3.2 Input and Output

Let us now add commands to handle input and output to our little language.

$$c \ ::= \ \cdots \ | \ \texttt{write}\ e \ | \ \texttt{read}\ x$$

We also introduce a syntactical category for programs so we are able to distinguish the semantics of a program from the semantics of a command.

$$p \ \in \ \text{Pgm} \qquad p \ ::= \ c$$

The semantic domain for programs should be the set of functions from lists of integers to lists of integers together with bottom and an error element.

$$\mathcal{P} \ \in \ \text{Pgm} \to (\text{List(Int)} \to \text{List(Int)}_\perp \cup \{\texttt{input-error}\})$$

To describe the meaning for commands we have to extend the state with components for input and output

$$\begin{aligned} \text{S} &= \text{Mem} \times \text{List(Int)} \times \text{List(Int)} \\ \text{Mem} &= \text{Var} \to \text{Int} \end{aligned}$$

We also have to rewrite the update and valof functions and write new functions input and output.

$$\begin{aligned} \text{update} &\in \text{Var} \times \text{Int} \times \text{S} \to \text{S} \\ \text{valof} &\in \text{Var} \times \text{S} \to \text{Int} \\ \text{input} &\in \text{S} \to ((\text{S} \times \text{Int}) \cup \{\texttt{input-error}\}) \\ \text{output} &\in \text{S} \times \text{Int} \to \text{S} \end{aligned}$$

We will use $\sigma[x \mapsto e]$ as a synonym for $\text{update}(x, e, \sigma)$ The meaning functions then have the types (where $A_{\perp,e} = A \cup \{\perp, \texttt{input-error}\}$)

$$\begin{aligned} \mathcal{E} &\in \text{Expr} \to (\text{S}_{\perp,e} \to \text{Int}_{\perp,e}) \\ \mathcal{B} &\in \text{Bexp} \to (\text{S}_{\perp,e} \to \text{T}_{\perp,e}) \\ \mathcal{C} &\in \text{Com} \to (\text{S}_{\perp,e} \to \text{S}_{\perp,e}) \\ \mathcal{P} &\in \text{Prog} \to (\text{List(Int)}_{\perp,e} \to \text{List(Int)}_{\perp,e}) \end{aligned}$$

and the following definitions

$$\begin{aligned} \mathcal{B}[\![t]\!] &= \lambda_{\perp,e}\sigma.\mathcal{T}[\![t]\!]^4 \\ \mathcal{B}[\![\neg b]\!] &= \lambda_{\perp,e}\sigma.\neg\mathcal{B}[\![b]\!]\sigma \\ \mathcal{B}[\![b_1 \ \texttt{or}\ b_2]\!] &= \lambda_{\perp,e}\sigma.\mathcal{B}[\![b_1]\!]\sigma \vee \mathcal{B}[\![b_2]\!]\sigma \\ \mathcal{B}[\![e_1 \ \texttt{=}\ e_2]\!] &= \lambda_{\perp,e}\sigma.\mathcal{E}[\![e_1]\!]\sigma \vee \mathcal{E}[\![e_2]\!]\sigma \end{aligned}$$

$$\begin{aligned} \mathcal{E}[\![x]\!] &= \lambda_{\perp,e}\sigma.\text{valof}(x, \sigma) \\ \mathcal{E}[\![n]\!] &= \lambda_{\perp,e}\sigma.\mathcal{N}[\![n]\!] \end{aligned}$$

$$
\begin{array}{rcl}
\mathcal{E}[\![e_1 \texttt{ + } e_2]\!] & = & \lambda_{\perp,e}\sigma.\mathcal{E}[\![e_1]\!]\sigma + \mathcal{E}[\![e_2]\!]\sigma \\
\mathcal{E}[\![e_1 \texttt{ * } e_2]\!] & = & \lambda_{\perp,e}\sigma.\mathcal{E}[\![e_1]\!]\sigma * \mathcal{E}[\![e_2]\!]\sigma \\
\\
\mathcal{C}[\![\texttt{skip}]\!] & = & \lambda_{\perp,e}\sigma.\sigma \\
\mathcal{C}[\![\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2]\!] & = & \lambda_{\perp,e}\sigma.\mathrm{cond}(\mathcal{B}[\![b]\!]\sigma, \mathcal{C}[\![c_1]\!]\sigma, \mathcal{C}[\![c_2]\!]\sigma) \\
\mathcal{C}[\![x\texttt{:=}e]\!] & = & \lambda_{\perp,e}\sigma.\mathrm{update}(\mathcal{X}[\![x]\!], \mathcal{E}[\![e]\!]\sigma, \sigma) \\
\mathcal{C}[\![c_1\,\texttt{;}c_2]\!] & = & \mathcal{C}[\![c_2]\!] \circ \mathcal{C}[\![c_1]\!] \\
\mathcal{C}[\![\texttt{write } e]\!] & = & \lambda_{\perp,e}\sigma.\mathrm{output}(\sigma, \mathcal{E}[\![e]\!]\sigma) \\
\mathcal{C}[\![\texttt{read } x]\!] & = & \lambda_{\perp,e}\sigma.\gamma \\
 & & \text{where} \\
 & & \gamma = \begin{cases} \texttt{input-error} & \text{if } \mathrm{input}(\sigma) = \texttt{input-error} \\ \mathrm{update}(x, n, \sigma') & \text{if } \mathrm{input}(\sigma) = \langle n, \sigma' \rangle \end{cases} \\
\\
\mathcal{P}[\![c]\!] & = & \lambda l.\mathcal{C}[\![c]\!]\langle \lambda x.0, l, [\,] \rangle
\end{array}
$$

One reason why we want to map programs and program parts to mathematical objects is that we have a well developed theory about mathematical objects. In this theory we can prove properties about mathematical objects and therefore also about programs and program parts, for example that the meaning of two expressions are equal. A mapping of programs to mathematical objects induces a semantical equality on all the syntactical categories

$$c \approx_C c' \qquad \text{if and only if} \qquad \mathcal{C}[\![c]\!] = \mathcal{C}[\![c']\!]$$

When we have a semantics, we can also "go backwards" and see if it is the "natural" equality. If we look at our semantical definition above, we can see that it implies for example that all nonterminating programs are equal. Since

$$
\begin{array}{rcl}
\mathcal{C}[\![\texttt{while true do skip}]\!] & = & \lambda\sigma.\perp \\
\mathcal{C}[\![\texttt{while true do write 1}]\!] & = & \lambda\sigma.\perp
\end{array}
$$

then

$$\texttt{while true do skip} \approx_C \texttt{while true do write 1}$$

A "better" semantic definition, at least in my opinion, would give the meaning

$$\lambda\sigma.1^\omega \qquad (1^\omega \text{ is the infinite list of ones})$$

to the program `while true do write 1` This is one reason why we will introduce domain theory later.

4.3.3 Block structure and local variables

Let us continue with our languages and introduce a block structure

$$c \;::=\; \cdots \;\mid\; \texttt{begin}\; d\;\texttt{;}\; c\;\texttt{end}$$

[4] $\lambda_{\perp,e}\sigma.e$ maps $\perp$ on $\perp$, `input-error` on `input-error` and otherwise σ on e.

where $d \in$ Decl is defined by

$$d \ ::= \ \texttt{var}\ x \mid d\ \texttt{;}\ d$$

To define the semantics of this language we have to introduce yet more complicated semantic domains. There are problems with programs like

```
begin
   var x; var y;
   x:=10; y:=20;
   begin
      var x;
      x:=100; y:=200;
   end;
   write x; write y
end
```

The assignment to `x` in the inner block should not be visible in the outer block in contrast to the assignment to `y`. Evaluating the program should give $[10, 200]$ as result.

In order to handle this problem the state has to be divided into an environment and a store.

$$\begin{aligned} \text{Env} &= (\text{Id} \to \text{Loc}) \times \text{Loc} \\ \text{Store} &= \text{Mem} \times \text{List(Int)} \times \text{List(Int)} \\ \text{Mem} &= \text{Loc} \to \text{Int} \end{aligned}$$

The environment maps identifiers to locations (addresses) and the memory maps locations to values. The second component of the environment represents the address of the last used location in the store. We need a way to allocate new unused locations in the memory. A simple, but of course inefficient, way of doing this is to keep track of the last used memory location.

We also introduce some operations on the environment and the store instead of manipulating them directly.

$$\begin{aligned} \text{updateS} &\in \text{Loc} \times \text{Int} \times \text{Store} \to \text{Store} \\ \text{valofS} &\in \text{Loc} \times \text{Store} \to \text{Int} \\ \text{inputS} &\in \text{Store} \to ((\text{Store} \times \text{Int}) \cup \{\texttt{input-error}\}) \\ \text{outputS} &\in \text{Store} \times \text{Int} \to \text{Store} \\ \text{updateE} &\in \text{Id} \times \text{Loc} \times \text{Env} \to \text{Env} \\ \text{locofE} &\in \text{Id} \times \text{Env} \to \text{Int} \end{aligned}$$

We now have a language with the following abstract syntax

$$p \ ::= \ c$$

$$\begin{aligned} c \ ::= \ & \texttt{skip} \mid x\texttt{:=}e \mid c\ \texttt{;}\ c \\ \mid \ & \texttt{begin}\ d\ \texttt{;}\ c\ \texttt{end} \mid \texttt{if}\ b\ \texttt{then}\ c\ \texttt{else}\ c \\ \mid \ & \texttt{while}\ b\ \texttt{do}\ c \mid \texttt{write}\ e \mid \texttt{read}\ x \end{aligned}$$

$$
\begin{array}{lcl}
d & ::= & \texttt{var}\ x \mid d\ \texttt{;}\ d \mid \texttt{skip} \\
e & ::= & x \mid n \mid e\texttt{+}e \mid e\texttt{*}e \\
b & ::= & t \mid \texttt{not}\ b \mid b\ \texttt{or}\ b \mid b\ \texttt{and}\ b
\end{array}
$$

where $p \in \text{Pgm}$, $c \in \text{Com}$, $x \in \text{Id}$, $e \in \text{Exp}$, $d \in \text{Decl}$, $n \in \text{Num}$, $t \in \text{Bool}$

The meaning functions have the following types

$$
\begin{array}{lcl}
\mathcal{P} & \in & \text{Prog} \to (\text{List(Int)} \to \text{List(Int)}_{\perp,e}) \\
\mathcal{C} & \in & \text{Com} \to \text{Env} \to (\text{Store}_{\perp,e} \to \text{Store}_{\perp,e}) \\
\mathcal{D} & \in & \text{Decl} \to \text{Env} \to \text{Env} \\
\mathcal{E} & \in & \text{Expr} \to \text{Env} \to (\text{Store}_{\perp,e} \to \text{Int}_{\perp,e}) \\
\mathcal{B} & \in & \text{Bexp} \to \text{Env} \to (\text{Store}_{\perp,e} \to \text{T}_{\perp,e})
\end{array}
$$

and they are defined in the following way

$$
\begin{array}{rcl}
\mathcal{B}[\![t]\!] & = & \lambda_{\perp,e}\rho\sigma.\mathcal{T}[\![t]\!] \\
\mathcal{B}[\![\neg b]\!] & = & \lambda_{\perp,e}\rho\sigma.\neg\mathcal{B}[\![b]\!]\rho\sigma \\
\mathcal{B}[\![b_1\ \texttt{or}\ b_2]\!] & = & \lambda_{\perp,e}\rho\sigma.\mathcal{B}[\![b_1]\!]\rho\sigma \vee \mathcal{B}[\![b_2]\!]\rho\sigma \\
\mathcal{B}[\![e_1\texttt{=}e_2]\!] & = & \lambda_{\perp,e}\rho\sigma.\mathcal{E}[\![e_1]\!]\rho\sigma \vee \mathcal{E}[\![e_2]\!]\rho\sigma \\
 & & \\
\mathcal{E}[\![x]\!] & = & \lambda_{\perp,e}\rho\sigma.\text{valofS}(\text{locofE}(x,\rho),\sigma) \\
\mathcal{E}[\![n]\!] & = & \lambda_{\perp,e}\rho\sigma.\mathcal{N}[\![n]\!] \\
\mathcal{E}[\![e_1\texttt{+}e_2]\!] & = & \lambda_{\perp,e}\rho\sigma.\mathcal{E}[\![e_1]\!]\rho\sigma + \mathcal{E}[\![e_2]\!]\rho\sigma \\
\mathcal{E}[\![e_1\texttt{*}e_2]\!] & = & \lambda_{\perp,e}\rho\sigma.\mathcal{E}[\![e_1]\!]\rho\sigma * \mathcal{E}[\![e_2]\!]\rho\sigma \\
 & & \\
\mathcal{D}[\![\texttt{skip}]\!] & = & \lambda\rho.\rho \\
\mathcal{D}[\![\texttt{var}\ x]\!] & = & \lambda\langle e,h\rangle.\langle\text{updateE}(x,h+1,\langle e,h\rangle),h+1\rangle \\
\mathcal{D}[\![d_1\,\texttt{;}\,d_2]\!] & = & \mathcal{D}[\![d_2]\!] \circ \mathcal{D}[\![d_1]\!] \\
 & & \\
\mathcal{C}[\![\texttt{skip}]\!] & = & \lambda_{\perp,e}\rho\sigma.\sigma \\
\mathcal{C}[\![\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!] & = & \lambda_{\perp,e}\rho\sigma.\text{cond}(\mathcal{B}[\![b]\!]\rho\sigma,\mathcal{C}[\![c_1]\!]\rho\sigma,\mathcal{C}[\![c_2]\!]\rho\sigma) \\
\mathcal{C}[\![x\texttt{:=}e]\!] & = & \lambda_{\perp,e}\rho\sigma.\text{updateS}(\text{locofE}(x,\rho),\mathcal{E}[\![e]\!]\rho\sigma,\sigma) \\
\mathcal{C}[\![c_1\,\texttt{;}\,c_2]\!] & = & \mathcal{C}[\![c_2]\!] \circ \mathcal{C}[\![c_1]\!] \\
\mathcal{C}[\![\texttt{write}\ e]\!] & = & \lambda_{\perp,e}\rho\sigma.\text{outputS}(\sigma,\mathcal{E}[\![e]\!]\rho\sigma) \\
\mathcal{C}[\![\texttt{read}\ x]\!] & = & \lambda_{\perp,e}\rho\sigma.\gamma \\
 & & \text{where} \\
 & & \gamma = \begin{cases} \texttt{input-error} \\ \qquad \text{if inputS}(\sigma) = \texttt{input-error} \\ \text{updateS}(\text{locofE}(x,\rho),n,\sigma') \\ \qquad \text{if inputS}(\sigma) = \langle n,\sigma'\rangle \end{cases} \\
 & & \\
\mathcal{P}[\![c]\!] & = & \lambda l.\gamma
\end{array}
$$

where

$$\gamma = \begin{cases} \texttt{input-error} & \text{if } \mathcal{C}[\![c]\!]\rho_0\sigma_0 = \texttt{input-error} \\ \bot & \text{if } \mathcal{C}[\![c]\!]\rho_0\sigma_0 = \bot \\ \mathrm{trd}(\mathcal{C}[\![c]\!]\rho_0\sigma_0) & \text{otherwise} \end{cases}$$

$$\begin{aligned} \rho_0 &= \langle \lambda x.0, 0 \rangle \\ \sigma_0 &= \langle \lambda x.0, l, [] \rangle \\ \mathrm{trd}(\langle x, y, z \rangle) &= z \end{aligned}$$

We have used the following operations on the state:

$$\begin{aligned} \mathrm{valofS} &\in \mathrm{Loc} \times \mathrm{Store} \to \mathrm{Int} \\ \mathrm{updateS} &\in \mathrm{Loc} \times \mathrm{Int} \times \mathrm{Store} \to \mathrm{Store} \\ \mathrm{updateE} &\in \mathrm{Id} \times \mathrm{Loc} \times \mathrm{Env} \to \mathrm{Env} \\ \mathrm{locofE} &\in \mathrm{Id} \times \mathrm{Env} \to \mathrm{Loc} \end{aligned}$$

with the following definitions

$$\begin{aligned} \mathrm{valofS}(l, \langle m, i, o \rangle) &= m(l) \\ \mathrm{updateS}(l, n, \langle m, i, o \rangle) &= \langle m[l \mapsto n], i, o \rangle \\ \mathrm{updateE}(i, l, \langle e, h \rangle) &= \langle e[i \mapsto 1], h \rangle \\ \mathrm{locof}(x, \langle e, h \rangle) &= e(x) \end{aligned}$$

4.3.4 Procedures

The extensions we have made to our program language have forced us to introduce more and more complicated domains but not caused us any real mathematical problems. We could even go further and introduce procedures

$$\begin{aligned} d &::= \cdots \mid \texttt{proc}\ p(x)\ c \\ c &::= \cdots \mid p(e) \end{aligned}$$

where $d \in \mathrm{Decl}$, $p \in \mathrm{Id}$, $x \in \mathrm{Id}$, $c \in \mathrm{Com}$ and $e \in \mathrm{Exp}$

In order to give meaning to these constructions we have to extend the environment to contain bindings of identifiers to procedure meanings. Furthermore, we move the information about the highest used location from the environment to the store.

$$\mathrm{Env} = \mathrm{Id} \to (\mathrm{Loc} + (\mathrm{Int} \to \mathrm{Store}_{\perp,e} \to \mathrm{Store}_{\perp,e}))$$

We also have to change

$$\begin{aligned} \mathrm{Store} &= (\mathrm{Loc} \to \mathrm{Int}) \times \mathrm{List}(\mathrm{Int}) \times \mathrm{List}(\mathrm{Int}) \times \mathrm{Loc} \\ \mathcal{D} &\in \mathrm{Decl} \to \mathrm{Env} \to \mathrm{Store} \to (\mathrm{Env} \times \mathrm{Store}) \end{aligned}$$

$$
\begin{array}{rcl}
\mathcal{D}[\![\texttt{var } x]\!] & = & \lambda\rho.\lambda_{\perp,e}\sigma.\langle\rho[x \mapsto h+1], \langle m, i, o, h+1\rangle\rangle \\
\mathcal{D}[\![\texttt{proc } p(x)\, c]\!] & = & \lambda\rho.\lambda_{\perp,e}\sigma.\langle\rho[p \mapsto inr(\gamma)], \sigma\rangle \\
 & & \text{where } \gamma \;=\; \lambda n.\lambda_{\perp,e}\langle m, i, o, h\rangle. \\
 & & \qquad \mathcal{C}[\![c]\!]\rho[x \mapsto inl(h+1)] \\
 & & \qquad\qquad (\text{update}(h+1, n, \langle m, i, o, h+1\rangle)) \\
\mathcal{C}[\![p(e)]\!] & = & \lambda_{\perp,e}\rho\sigma.\underbrace{(outr(\rho(p)))}_{\text{semantics of } p}(\mathcal{E}[\![e]\!]\rho\sigma)\sigma
\end{array}
$$

We have successively introduced more and more constructions in our programming language. It has been almost no mathematical problems and we now have an almost real programming language. So what type of program constructions can cause problems?

Recursive definitions There are two different ways of solving the problem. One solution is to let the semantics of a recursive function (procedure) be the *text* of the function. Another is to say that it is the least fix point to the recursive functional which is the meaning. In order to do this properly we need a theory in which there exists unique (least) fixed points to recursive functionals.

Procedure variables Assume we want to have procedure variables and the possibility to assign procedures to variables in our programming language. For example something like

```
var p :  procedure; y :  integer

p := procedure(x:integer);
begin y := x+12 end;

y:=1; p(15);

p := procedure(y:integer);
begin y := y-x end;
...
```

Now we must be able to store procedures. The semantics of a procedure is a function of type

$$\text{Int} \to \text{Store} \to \text{Store}$$

If we want to have such objects in the store, the mathematical object for the store has to satisfy the following equation

$$\text{Store} \;=\; (\text{Loc} \to (\text{Int} + (\text{Int} \to (\text{Store} \to \text{Store})))) \times \text{List(Int)} \times \text{List(Int)}$$

We need a set which contains functions on itself as elements. This is not possible if we use ordinary functions and sets as we have done up to now.

Self application The same kind of problem arises in for example untyped λ-calculus (and LISP) where it is possible to apply a function on itself. This leads to the problem of solving type equations like

$$\Lambda \;=\; \Lambda \to \Lambda$$

Assume that the semantic domain of a λ term is Λ. But a term $\lambda x.x$ is also a function that maps λ terms to λ terms, i.e. whose domain is $\Lambda \to \Lambda$.

This leads us to the next chapter where we will introduce a theory where equations such as the ones we have just encountered have solutions.

Exercise 25 *When the meaning of the syntactical categories in a programming language is the "ordinary" (natural) meaning we say that we give a* **standard semantics**. *We can get a* **non-standard semantics** *if we let the semantics of an expression be its type. Define such a non-standard denotational semantics for the language*

$$\begin{array}{rcl} e & ::= & n \mid t \mid x \mid e \texttt{ + } e \mid e \texttt{ * } e \\ & \mid & \texttt{if } e \texttt{ then } e \texttt{ else } e \mid e \texttt{ or } e \mid \texttt{not } e \\ & \mid & \texttt{< } e, e \texttt{ > } \mid \texttt{let (} x \texttt{ , } y \texttt{) = } e \texttt{ in } e \end{array}$$

where $e \in$ Expr, $n \in$ IntConst, $t \in$ BoolConst *and* $x \in$ Var

Exercise 26 *Define a semantical equivalence for commands and prove that*

$$\texttt{while } b \texttt{ do } c \approx_C \texttt{if } b \texttt{ then } c \texttt{ ; while } b \texttt{ do } c \texttt{ else skip}$$

Exercise 27 *Extend two of the languages (the simplest and the most complicated) with a command*

$$c ::= \cdots \mid \texttt{swap } x_1\ x_2$$

where $c \in$ Com *and* $x \in$ Var

which swaps the values of two variables. Furthermore show that

$$\texttt{swap } x\ x \approx_C \texttt{skip}$$

Chapter 5

A Brief Introduction to Domain Theory

We start by trying to motivate the notions which are basic in domain theory. One of the basic principles of computers in general is that they work on finite objects and in some theories, for example basic Computability theory (Recursion theory), one therefore assumes that a computing machine can only manipulate a predefined domain of objects, for example the natural numbers. All other objects such as characters, lists and programs which one wants to compute with, have to be coded in terms of natural numbers. With this view it is natural to say that a program computes a partial function from natural numbers to natural numbers and that a computation intuitively proceeds as follows

1. The input is supplied to the machine.
2. The machine starts the computation.
3. If it stops, the result is presented.

This model is today a little bit "old fashioned" when most programs are evaluated interactively. We would like to have a model of computation which take into account that also non terminating programs could give a result and that also non terminating programs could be interesting and useful. Take for example a program that generates the decimals of π — it is not just a non terminating program which is equal to every other non terminating program. To achieve this we must leave the strictly finite world and assume that a computer may also compute infinite objects. But of course not all infinite objects, only simple ones — the machine can after all only handle finite approximations of the infinite objects. A suitable theory for this view of computers and computing is what Dana Scott has called *continuous mathematics* — all objects are finite or could be approximated by finite approximations. So we replace the old fashioned finite model by

> A computation gives successively better and better approximations of the output. In this process better and better approximations of the input is needed.

Notice that this is a strict generalization of the finite model in which *all information* of the input is given immediately and the program then in one step either produces no information or complete information about the output.

Consider the evaluation of a Pascal(C, Ada, ...) program in an interactive environment:

1. When we start the evaluation we have no information about the output.
2. After a while we have perhaps got a piece of the output, we have a better approximation of the output (we know how it starts).
3. Then the program does not give a better approximation until it gets a better approximation of the input. When the program started it had no information at all about the input.
4. By providing a better approximation of the input, the program can compute yet another piece of the output and therefore can give a better approximation of the output.

$\vdots$

The kinds of infinite objects we may compute have to be determined by their finite approximations and be limits of sequences of finite approximations. We can also see that a program computes a *monotone function* – a better approximation of the input gives a better approximation of the output.

Let us summarize the requirements for semantic domains:

- Every domain is a partial order $\langle D, \sqsubseteq \rangle$ where the relation $\sqsubseteq$ orders the elements according to their information content.
- All programs compute monotone and continuous functions.
- Every domain has a countable base of finite elements.

Before we can introduce domains more formally, we have to make some definitions.

Definition 5 *A* **partial order** *is a pair $\langle D, \sqsubseteq \rangle$, where D is a set and $\sqsubseteq$ is a binary relation on D such that*

1. $x \sqsubseteq x$ *for all* $x \in D$ *(reflexive)*
2. *if* $x \sqsubseteq y$ *and* $y \sqsubseteq x$ *then* $x = y$ *(antisymmetric)*
3. *if* $x \sqsubseteq y$ *and* $y \sqsubseteq z$ *then* $x \sqsubseteq z$ *(transitive)*

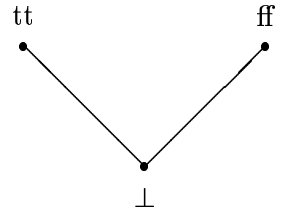
If $\langle D, \sqsubseteq \rangle$ is a partial order and A is a subset of D, then an element $x \in D$ is an *upper bound* of A if $a \sqsubseteq x$ for all $a \in A$. An upper bound is called a *least upper bound* if it is less than all other upper bounds (notation $\bigsqcup X$).

Definition 6 *An (ω)chain is a sequence $[x_i]_{i \geq 0}$ such that $x_i \sqsubseteq x_{i+1}$ for all $i \geq 0$.*

Definition 7 *A partial order $\langle D, \sqsubseteq \rangle$ is a* **complete partial order (CPO)** *if*

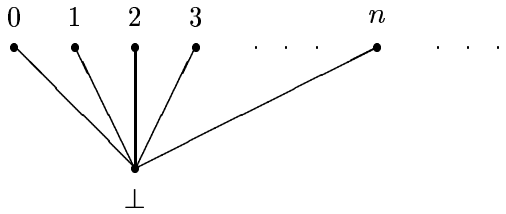
1. *there is a least element $\bot$ in D*
2. *every chain $[x_i]_{i \geq 0}$ has a least upper bound in $\mathcal{D}$*

A *domain* is a CPO with its elements ordered according to their information content. The least element $\bot$ means *no information*. The simplest way to construct a CPO is just to add a least element $\bot$ to an arbitrary set. The result is a *flat domain*.



The flat domain $\{\mathrm{ff}, \mathrm{tt}\}_\bot = \mathcal{T}$

In a flat domain there are just *two levels of information*, no information and complete information. The element $\bot$ denotes no information and, in our example, tt and ff denotes different but complete information. Another important domain which we can get in the same way is $\mathbf{N}_\bot$:



The flat domain $\mathbf{N}_\bot = \mathcal{N}$.

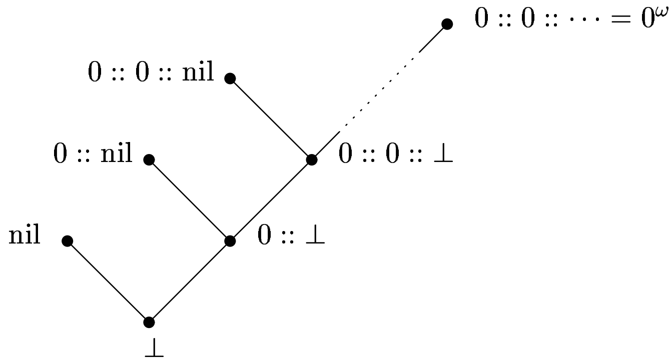
If $x \sqsubseteq y$ in a flat domain, then either $x = \bot$ or $x = y$. In most programming languages all primitive types are flat domains, for example `Char`, `Integer`, `Boolean`, ... If a Pascal program is supposed to give an integer as its result, only two situations can occur:

1. We do not get any information at all, that is nothing is printed.
2. We get all information, a number is printed.

It is impossible to imagine a situation where we get partial information about the output, for example that the number is greater than four but less than ten. A more complicated situation occurs if we have a program that is supposed to give a list of integers as result. Then the following situations can occur

1. No output – no information about the result
2. We have got the output `[1,2` and the program has not terminated – we have partial information about the result, we know that it begins with $1, 2$.
3. We have got more of the output, `[1,2,3,4`, and the program has not terminated – we still have only partial information about the result, but more information than in the previous case.
4. We have got the complete output `[1, 2, 3, 4, 5]` and the program has terminated – we have complete information about the result.

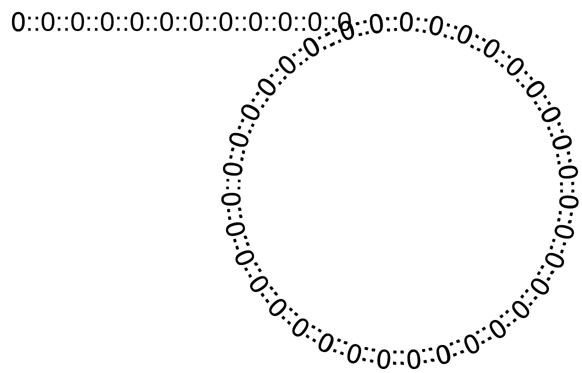
How should the domain look like to be suitable to model the result of such a program? Assume, for simplicity, that the result is just a list of zeroes (instead of arbitrary integers). Since $\perp$ should be the least element and lists must be ordered according to the prefix ordering, we get the following domain



The domain of lists of zeroes

The element 0^ω is the infinite list of zeroes and could perhaps be defined as follows (from J. Stoy and M. C. Escher [38]):

0^w =



The definition of 0^ω

Notice also that $0 :: \perp$ means the list that starts with a zero followed by a list about which we have no information. The next step of information that we can get is either that the rest of the list is empty (= nil) or that it starts with a zero.

The domain of lists of Booleans or lists of integers is just a straight forward generalization of the list of zeroes. The domain of lazy lists is a suitable semantic domain for an interactive Pascal program, but not suitable to denote lists in eager (strict, call-by-value) programming languages such as ML. We can see that there are three reasonable possibilities to choose a domain for lists in a programming language.

flat lists: the `cons`-operator is strict in both its arguments.

$$\begin{cases} \texttt{cons}(\perp, x) = \perp \\ \texttt{cons}(x, \perp) = \perp \end{cases}$$

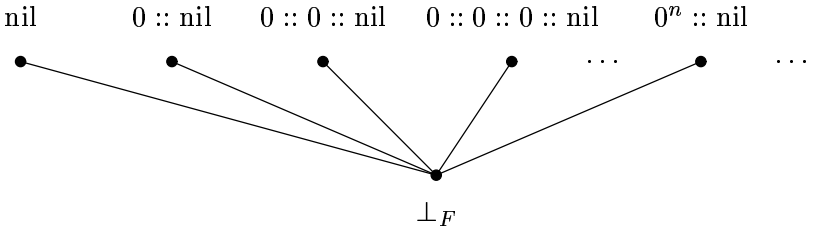
lazy lists: the cons-operator is strict in its first argument, but lazy in its second.

$$\exists x.\mathtt{cons}(x, \bot) \neq \bot$$

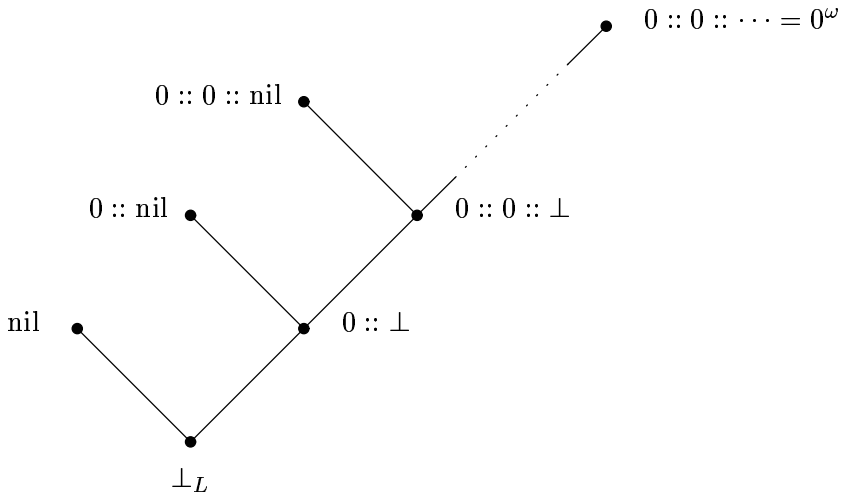
fully lazy lists: the cons-operator is lazy in both its argument

$$\begin{cases} \exists x.\mathtt{cons}(x, \bot) \neq \bot \\ \exists x.\mathtt{cons}(\bot, x) \neq \bot \end{cases}$$

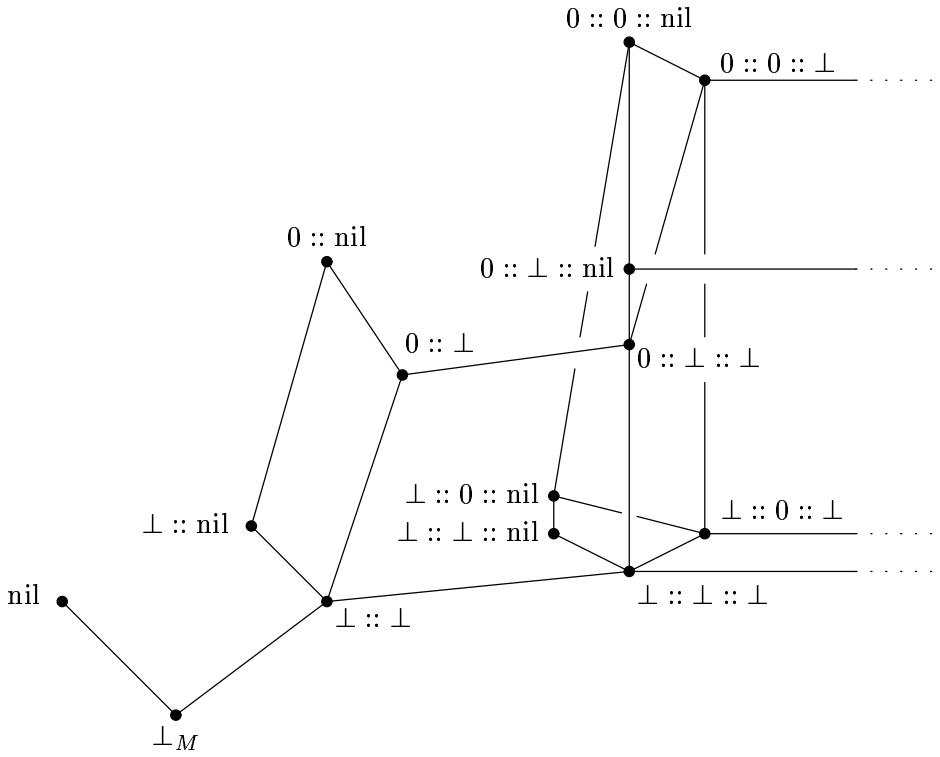
These domains three domains can be drawn as follows.



The flat domain of lists of zeroes



The lazy domain of lists of zeroes



The maximal lazy domain of lists of zeroes

Two programs that would give different results in the domains are

```
let fun from i = cons(i, from(i+1))
in  hd (from 1)
end
```

and

```
let fun length[]          = 0
      | length(cons(x,y)) = 1 + length(y)

    fun f(i) = f(i+1)

in  length (map f [1,2,3,4,5])
end
```

Exercise 28 *Give an example of a partial order:*

1. *without a least element; and one*
2. *with a chain without a least upper bound.*

Exercise 29 *Draw pictures for the three domains of lists of Booleans.*

Exercise 30 *What results do the two programs above give for the three different domains of lists.*

Exercise 31 *The difference between the different domins of lists of zeroes could be understood in terms of saying that an element containing $\bot$ in the simple domain is split into several elements in the more complicated domain. Indicate which elements in the lazy domains that $\bot$ in the flat domain has been split into. Do the same for $\bot$ in the lazy domain and the elements in the maximal flat domain.*

5.1 Simple domain constructors

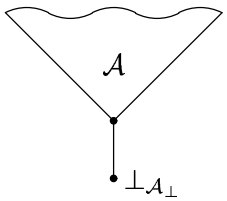
In the previous section we saw how one could construct domains by explicitly describing the elements of the domain and their order. A more systematic way is to define domain constructors that construct new domains from simpler ones. We will give some examples which correspond to the usual type constructors in programming languages.

5.1.1 Lifting

We have already seen the first domain constructor. However we have used it a bit strange since it has not taken a domain as argument, but an ordinary set. The result has been a flat domain created from the argument. We call this operation *lifting* and use the notation $A_\bot$ for the result of applying the lifting operator to the set A. It is easy to extend the operator also to ordinary domains. If we have a domain $\mathcal{A}$, we can **lift** it to $\mathcal{A}_\bot$ by putting in a new least element below the others.

$$\mathcal{A}_\bot \;=\; \langle \{\mathrm{up}(a) \mid a \in \mathcal{A}\} \cup \{\bot_{A_\bot}\},\ \sqsubseteq_{A_\bot} \rangle$$

$$\text{where}\quad \bot_{A_\bot} \sqsubseteq \mathrm{up}(x), \qquad x \in \mathcal{A}$$
$$\mathrm{up}(a) \sqsubseteq_{A_\bot} \mathrm{up}(a') \Longleftrightarrow a \sqsubseteq_A a'$$



The domain $\mathcal{A}_\bot$

5.1.2 Product

If we have two domains we can construct their product by combining the elements to pairs and use the pointwise ordering on the pairs.

Definition 8 *If $\mathcal{A} = \langle A,\ \sqsubseteq_A \rangle$ and $\mathcal{B} = \langle B,\ \sqsubseteq_B \rangle$ then we define the* **product domain** *$\mathcal{A} \times \mathcal{B}$ by*

$$\mathcal{A} \times \mathcal{B} = \langle A \times B,\ \sqsubseteq_{A \times B} \rangle$$

where

$$\langle a, b\rangle \sqsubseteq_{A\times B} \langle a', b'\rangle \quad \textit{if and only if} \quad a \sqsubseteq_A a' \textit{ and } b \sqsubseteq_B b'$$

The least element in the domain $\mathcal{A} \times \mathcal{B}$ is $\bot_{\mathcal{A}\times\mathcal{B}} = \langle \bot_{\mathcal{A}}, \bot_{\mathcal{B}}\rangle$ by the definition of the order of the elements.

It is quite easy to see that $\mathcal{A} \times \mathcal{B}$ is a CPO if $\mathcal{A}$ and $\mathcal{B}$ are CPOs. To show this, we must show that there is a least element and that every chain has a least upper bound in the domain.

1. $\bot_{\mathcal{A}\times\mathcal{B}}$ is the least element since $\bot_{\mathcal{A}}$ is less than every element in A and $\bot_{\mathcal{B}}$ is less than every element in B.

2. every chain

   $$c_{ab} = \langle a_1, b_1\rangle \sqsubseteq_{A\times B} \langle a_1, b_2\rangle \sqsubseteq_{A\times B} \cdots$$

   has a lub $\langle \sqcup c_a, \sqcup c_b\rangle$ since

   $$c_a = a_1 \sqsubseteq_A a_2 \sqsubseteq_A \cdots$$

   is a chain in $\mathcal{A}$ with lub $\sqcup c_a$ and

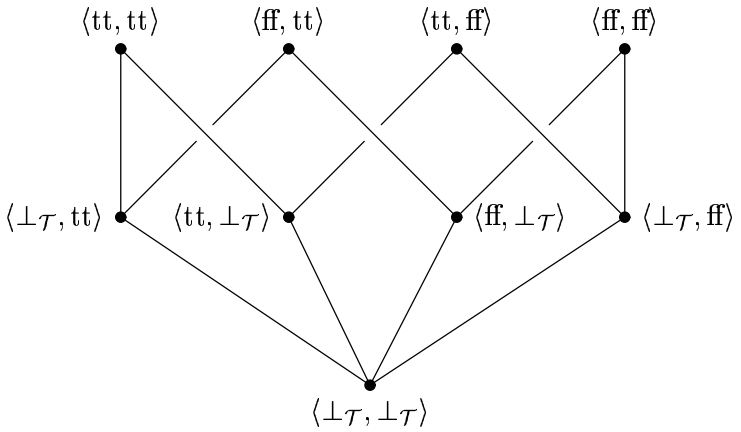
   $$c_b = b_1 \sqsubseteq_B b_2 \sqsubseteq_B \cdots$$

   is a chain in $\mathcal{B}$ with lub $\sqcup c_b$.

The product domain constructor is suitable for describing the meaning of the type of cartesian products with a pairing operation having the following properties:

$$\begin{aligned} \langle \bot_A, b\rangle &\neq \bot_{\mathcal{A}\times\mathcal{B}} \text{ when } b \neq \bot_B \\ \langle a, b\rangle &\neq \bot_{\mathcal{A}\times\mathcal{B}} \text{ when } a \neq \bot_A \text{ and } b \neq \bot_B \\ \langle \bot_A, \bot_B\rangle &= \bot_{\mathcal{A}\times\mathcal{B}} \end{aligned}$$

Example: As an example of a product domain, we give the order for the domain $\mathcal{T} \times \mathcal{T}$, that is the product of Booleans.



The product domain $\mathcal{T} \times \mathcal{T}$

□

It is possible to define other product domains with other properties. In another product all pairs with one component equal to $\perp$ are identified.

$$\begin{array}{lcll} \langle \perp_{\mathcal{A}}, y \rangle & = & \perp_{\mathcal{A}\times\mathcal{B}} & \text{for all } y{\in}\mathcal{B} \\ \langle x, \perp_{\mathcal{B}} \rangle & = & \perp_{\mathcal{A}\times\mathcal{B}} & \text{for all } x{\in}\mathcal{A} \end{array}$$

We call this the *smashed product* and denote it by $\mathcal{A} \otimes \mathcal{B}$. This kind of product is suitable for strict languages with a strict pairing constructor. It is also possible to define a *separated product* where

$$\begin{array}{rcl} \langle \perp_A, \perp_B \rangle & \neq & \perp_{\mathcal{A}\times\mathcal{B}} \\ \perp_{\mathcal{A}\times\mathcal{B}} & \sqsubseteq & \langle \perp_A, \perp_B \rangle \end{array}$$

This product could be used to model languages with an operation where you can determine something to be a pair even if both components are nonterminating. Consider for example the expression

```
let (x,y) = (loop,loop)
in 23
```

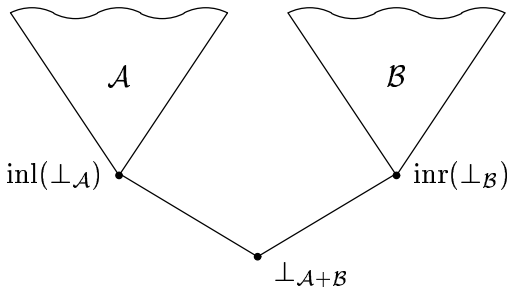
This program may give the result 23 if the pattern matching is lazy enough. Compare with the expression

```
let (x,y) = loop
in 23
```

If you get different behaviour from these programs it means that $\langle \perp, \perp \rangle$ is not the least element in the domain. The separated product is isomorphic to $(\mathcal{A} \times \mathcal{B})_\perp$, that is the lifted ordinary product.

5.1.3 Disjoint Union (Sum)

If we have two domains $\mathcal{A}$ and $\mathcal{B}$ then we can form the *(separated) sum*, $\mathcal{A} + \mathcal{B}$, by joining $\mathcal{A}$ and $\mathcal{B}$ with a new least element $\perp_{\mathcal{A}+\mathcal{B}}$.



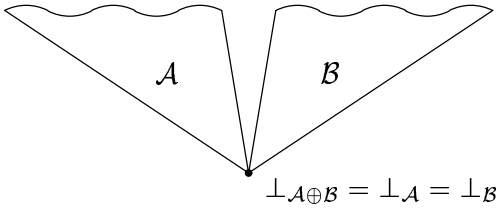
The domain $\mathcal{A} + \mathcal{B}$

In other words:

$$\begin{array}{rcl} \mathcal{A}+\mathcal{B} & = & \langle A+B \sqcup \{\bot_{\mathcal{A}+\mathcal{B}}\},\ \sqsubseteq_{\mathcal{A}+\mathcal{B}}\rangle \\ & \text{where} & \text{inl}(a) \sqsubseteq_{\mathcal{A}+\mathcal{B}} \text{inl}(a') \text{ iff } a \sqsubseteq_{\mathcal{A}} a' \\ & & \text{inr}(b) \sqsubseteq_{\mathcal{A}+\mathcal{B}} \text{inr}(b') \text{ iff } b \sqsubseteq_{\mathcal{B}} b' \\ & \text{and} & \bot_{\mathcal{A}+\mathcal{B}} \sqsubseteq_{\mathcal{A}+\mathcal{B}} x \text{for all } x \in A+B \end{array}$$

The separated sum is suitable to model a disjoint union with lazy constructors (inl, inr) in a programming language.

A variant of the separated sum is the *coalesced sum* $\mathcal{A} \oplus \mathcal{B}$ where the elements $\bot_{\mathcal{A}}$ and $\bot_{\mathcal{B}}$ are identified and equal with the least element in the domain. This sum is suitable to model a disjoint union with strict constructors.



The coalesced sum $\mathcal{A} \oplus \mathcal{B}$

It is also possible to construct half separated sum domains modeling one lazy and one strict constructor.

Exercise 32 *Show that $\mathcal{A}+\mathcal{B}$ is a CPO.*

Exercise 33 *Define $\mathcal{A} \oplus \mathcal{B}$ and show that it is a CPO.*

5.2 Functions

The function domains are the really important part of domain theory. We have indicated that it is impossible to solve equations such as

$$\text{Store} \simeq \cdots \text{Store} \to \text{Store}$$

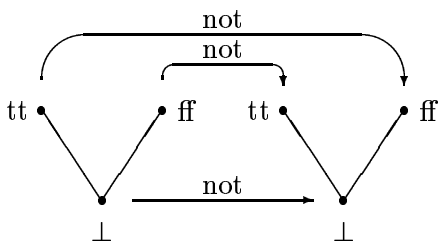
if Store should be a set and $\to$ is the set of ordinary set theoretic functions. The problem is simply that there are "too many functions" in the function set. To construct function domains for which we can solve equations as the one above, we must reduce the number of functions without, of course, removing any "interesting" functions.

5.2.1 Monotone and continuous functions

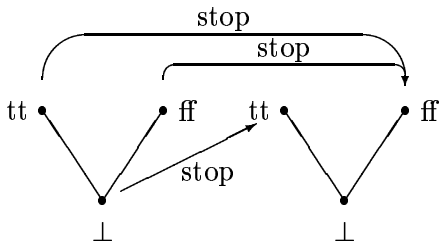
If $\mathcal{A}$ and $\mathcal{B}$ are two domains and $f \in A \to B$ then f is *monotone* if for $x \in \mathcal{A}$ and $y \in \mathcal{B}$

$$x \sqsubseteq_{\mathcal{A}} y \quad \text{implies that} \quad f(x) \sqsubseteq_{\mathcal{B}} f(y)$$

Every computable function is monotone. If we give a better approximation of the argument, we get a better (or at least not a worse) approximation of the result. Two examples:



the monotone function: not



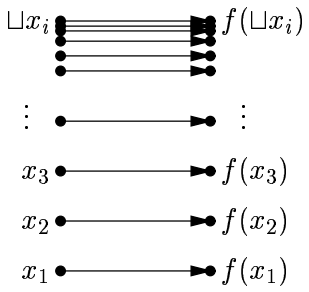
the non monotone function: stop

So if we forget all non monotone function we have not removed any computationally interesting functions, but unfortunately this is not enough. We must also remove the discontinuous functions in order to reduce the size of the function domain so that the number of functions is manageable.

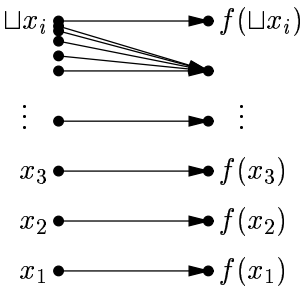
Definition 9 *A monotone function $f \in A \to B$ is* **continuous** *if for all chains $\langle x_i \rangle$ in A*

$$\underbrace{f(\bigsqcup[x_i])}_{lub\ in\ A} = \underbrace{\bigsqcup[f(x_i)]}_{lub\ in\ B}$$

Continuity means something more than monotonicity only for "interesting chains" (chains which do not contain their own lub — infinite chains). For a discontinuous function g we can not get an arbitrary good approximation to $g(\bigsqcup x_i)$ by choosing a sufficiently good approximation to the argument.



A continous function



A discontinous function

Example: Let us give two examples of discontinuous functions. Two functions which gives one answer for all approximations to an element but another answer for the limit.

1. $\text{isinfinite} \in \text{List(N)} \to \text{Bool}$

$$\text{isinfinite}(l) = \begin{cases} \text{true} & \text{if } l \text{ is infinite} \\ \bot & \text{otherwise} \end{cases}$$

2. $g \in (\text{N} \to \text{N}) \to \text{Bool}$

$$g(f) = \begin{cases} \text{true} & \text{if } f(n) \neq \bot \text{ for all } n \\ \bot & \text{otherwise} \end{cases}$$

Intuition

- The approximation of an infinite list is a finite initial segment.
- The approximation of a function is a finite function, i.e. a function defined for a finite domain.

The result would be $\bot$ for all approximations but true for the limit. So the functions are not continuous. □

5.2.2 Function domains

Definition 10 *If $\mathcal{A}$ and $\mathcal{B}$ are domains then $\mathcal{A} \to \mathcal{B}$ is a domain*

$$\mathcal{A} \to \mathcal{B} \;=\; \langle \{ f \in A \to B \;\mid\; f \text{ monotone } \& f \text{ continuous } \},\ \sqsubseteq_{\mathcal{A}\to\mathcal{B}} \rangle$$

$$\text{where } f \sqsubseteq_{\mathcal{A}\to\mathcal{B}} g \text{ if and only if } f(x) \sqsubseteq_B g(x) \text{ for all } x \in A$$

A function is greater than another if the result for every argument is greater. The least element in $\mathcal{A} \to \mathcal{B}$ is the function that gives $\bot_{\mathcal{B}}$ as result for every argument. Notation

$$\bot_{\mathcal{A}\to\mathcal{B}} \;=\; \lambda x.\bot_{\mathcal{B}}$$

or, if we see a function as a set $\{\langle x_i, f(x_i)\rangle\}$ and exclude all pairs $\langle x_i, \bot_{\mathcal{B}}\rangle$

$$\bot_{\mathcal{A}\to\mathcal{B}} \;=\; \{\}$$

One of the nice properties of continuous functions is that these functions always have a least fixed point.

Theorem 11 *Let $\mathcal{D}$ be a domain and f an element in $\mathcal{D} \to \mathcal{D}$. Then f has a least fixed point,* $\mathbf{fix}(f)$. *Furthermore*

$$\mathbf{fix}(f) \;=\; \bigsqcup_{i=0}^{\infty} f^i(\bot)$$

$$\left(\;=\; \bigsqcup_{i=0}^{\infty} \langle f^i(\bot)\rangle_{i>0}\right)$$

It is also possible to show that if $\mathcal{D}$ is a domain, then

$$\mathbf{fix} \;=\; \lambda f.\bigsqcup_{i=0}^{\infty} f^i(\bot) \;\in\; (D \to D) \to D$$

is continuous and therefore belong to the domain $(\mathcal{D} \to \mathcal{D}) \to \mathcal{D}$. This means that we have a way of giving meaning to recursively defined functions in programming languages! One of the problems we had earlier.

Exercise 34 *Assume we have a function on flat domains. What is the intuition of the function order? Draw the order between the functions in* $\mathrm{Bool}_\bot \to \mathrm{Bool}_\bot$

Exercise 35 *Prove the fixpoint theorem (theorem 11).*

Exercise 36 *Consider the domain, $\mathcal{I}$, with three elements called $\bot$, $|$ and $\top$ and with the following order*

1. *Which mathematical functions exists from $\mathcal{I}$ to $\mathcal{I}$?*
2. *Which are monotone?*
3. *Which are continuous?*
4. *Are all monotone functions on a finite domain continuous?*

5.2.3 Recursively defined functions

If f is recursively defined as a fixed point to the functional F_f then

$$f = \mathbf{fix}(F_f) \qquad \text{or rather } \mathcal{D}[\![f]\!] = \mathbf{fix}(D[\![F_f]\!])$$
$$= \bigsqcup_{i=0}^{\infty} H^i(\bot)$$

Example: Let us as an example take

$$f = \text{fac}$$
$$F_{\text{fac}} = \lambda\texttt{f.}\lambda\texttt{n.if n=0 then 1 else n * f(n-1)}$$

What about the functionals H^i:

$$\begin{aligned}
H^0(\bot) &= \bot_{\mathcal{N}\to\mathcal{N}} \\
&= \{\} \quad \text{the completely undefined function} \\
H^1(\bot) &= \lambda\texttt{n.if n=0 then 1 else n*}H^0(\bot)\texttt{(n-1)} \\
&= \{\langle 0,1\rangle\} \\
H^2(\bot) &= \lambda\texttt{n.if n=0 then 1 else n*}H^1(\bot)\texttt{(n-1)} \\
&= \{\langle 0,1\rangle, \langle 1,2\rangle\} \\
&\vdots \\
H^n(\bot) &= \lambda\texttt{n.if n=0 then 1 else n*}H^{n-1}(\bot)\texttt{(n-1)} \\
&= \{\langle 0,1\rangle, \langle 1,2\rangle, \ldots \langle n,n!\rangle\}
\end{aligned}$$

We can see that fac is the limit of the chain of H^i:s

$$\text{fac} = \bigsqcup_{i=0}^{\infty} H^i(\bot)$$

□

Using **fix**, we can now also give a much simpler meaning to the while statement.

$$\begin{array}{lcll} \mathcal{C}[\![\texttt{while } b \texttt{ do } c]\!] & = & \mathbf{fix}(H) & \\ & \text{where} & H & \in & (\mathrm{S} \to \mathrm{S}) \to (\mathrm{S} \to \mathrm{S}) \\ & & & = & \lambda\omega \in \mathrm{S} \to \mathrm{S}.\lambda\sigma \in \mathrm{S}.\mathrm{Cond}(\mathcal{B}[\![b]\!]\sigma, \omega(\mathcal{C}[\![c]\!]\sigma), \sigma) \end{array}$$

Exercise 37 *Give an operational intuition of* $H^i(\bot)$

Exercise 38 *Prove that* `while` b `do` $c \equiv_C$ `if` b `then` c`;while` b `do` c `endif`

5.3 Domain equations

One of the reasons why we introduced domain theory was that we wanted to be able to solve domain equations such as

$$\mathcal{D} \simeq \mathcal{N} + (\mathcal{D} \to \mathcal{D})$$

So how do we construct a solution if $\mathcal{D}$ and $\mathcal{N}$ are domains and $+$ and $\to$ are interpreted as domain constructors rather than set constructors. It is unfortunately a little bit too complicated to get into details of this (for more details see for example [36] or [38]). In brief, there are two methods of constructing a solution of a recursive domain equation

1. Start with a simple domain (a bad approximation of the solution) then successively construct better and better approximations. The solution is "the limit" of this sequence and is called the *inverse limit* construction.

2. Start with a domain which is large enough and then show how the primitive domains should be interpreted. Furthermore, it is shown how the domain constructors should be interpreted in this *universal domain*. Scott has used $\mathbf{P}\omega$ (the powerset of the set of natural numbers with setinclusion as ordering principle) as a universal domain [37].

Let us take a very simple example and construct a solution according to the first method. The domain equation is

$$Alist \simeq \mathcal{O} + (A \times Alist)$$

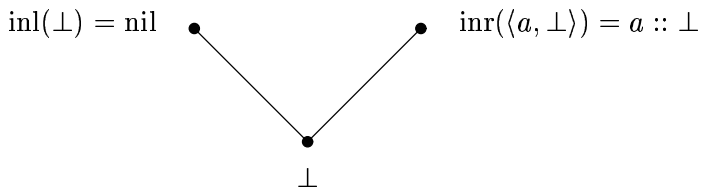
Start with the simplest possible domain as the first approximation. Take

$$Alist_0 = \mathcal{O} = \{\bot\} \quad \text{(the domain with just one element)}$$

The approximation $Alist_0$.

Continue to construct a better approximation by using the first approximation in the equation and iterate once.

$$Alist_1 = \mathcal{O} + (A \times Alist_0)$$



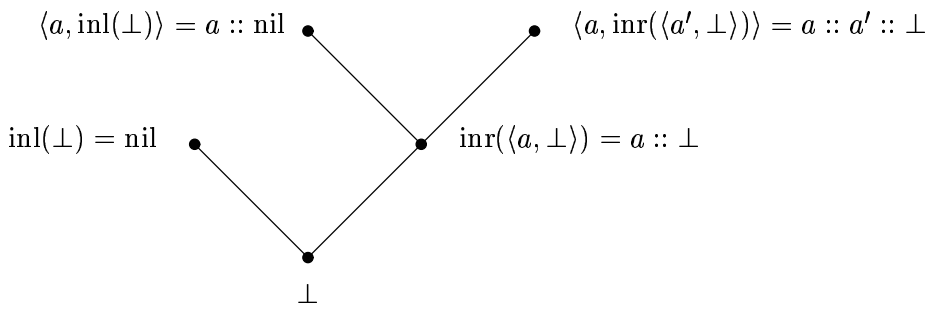
The approximation $Alist_1$.

If we proceed one more step, we get

$$\begin{aligned} Alist_2 &= \mathcal{O} + \underbrace{(A \times Alist_1)}_{A \times (\mathcal{O} + (A \times Alist_0))} \end{aligned}$$

We have four different kinds of elements in this domain.

1. If the element is in $\mathcal{O}$, the first summand, then it is a representative of the empty list nil exactly as in the first approximation above.
2. If the element is in $(A \times Alist_0)$, we have three situations dependent on the second component.
   (a) It could be $\bot_{\mathcal{O}+(A\times Alist_0)}$, and the complete element $\langle a, \bot\rangle$ represents the list $a :: \bot$.
   (b) If the second component is $\mathrm{inl}(\bot) \in \mathcal{O} + (A \times Alist_0)$ then $\langle a, \mathrm{inl}(\bot)\rangle$ represents the list $a ::$ nil.
   (c) Finally if the second component is $\mathrm{inr}(\langle a', \bot\rangle) \in \mathcal{O} + (A \times Alist_0)$ with $\langle a', \bot\rangle \in A \times Alist_0$, then $\langle a, \mathrm{inr}(\langle a', \bot\rangle)\rangle$ represents the list $a :: a' :: \bot$



The approximation $Alist_2$.

By continuing and constructing the sequence of domains $Alist_i$ we can form

$$Alist = \bigsqcup_{i=0}^{\infty} Alist_i = \bigsqcup \left\{ \bot, \bot, \bot, \bot, \bot, \ldots \right\}$$

We then get the solution to the domain equation. Notice that we have not written out all elements in the domain, a denotes an arbitrary element in A

Bibliography

[1] Annika Aasa. Recursive Descent Parsing of User Defined Distfix Operators. Licentiate Thesis, Chalmers University of Technology and University of Göteborg, Sweden, May 1989.

[2] Annika Aasa. Precedences in Specifications and Implementations of Programming Languages. In J. Maluszynski and M. Wirsing, editors, *Proceedings of Third International Symposium on Programming Language Implementation and Logic Programming, Lecture Notes in Computer Science*, volume 528, pages 183–194. Springer-Verlag, August 1991.

[3] Annika Aasa, Kent Petersson, and Dan Synek. Concrete Syntax for Data Objects in Functional Languages. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 96–105, Snowbird, Utah, 1988.

[4] A. V. Aho, J. D. Ullman, and R. Sethi. *Compilers: Principles, Techniques, Tools.* Addison-Wesley Publishing Company, Reading, Mass., 1986.

[5] L. Augustsson and T. Johnsson. The Chalmers Lazy-ML Compiler. *The Computer Journal*, 32(2):127–141, 1989.

[6] L. Augustsson and T. Johnsson. *Lazy ML User's Manual.* Programming Methodology Group, Department of Computer Sciences, Chalmers, S–412 96 Göteborg, Sweden, 1993. Distributed with the LML compiler.

[7] Roland Backhouse. *Syntax of Programming Languages, Theory and Practice.* Prentice Hall, 1979.

[8] H. P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics.* North-Holland, 1984.

[9] Andrzej Blikle and Andrzej Tarlecki. Naive denotational semantics. In R. E. A. Mason, editor, *Information Processing 83, IFIP*, pages 345–355. Elsevier, 1983.

[10] Luca Cardelli. Basic Polymorphic Typechecking. *Science of Computer Programming*, 8:147–172, 1987.

[11] J. C. Cleaveland and R. C. Uzgalis. *Grammars for Programming Languages.* Elsevier North-Holland, 1977.

[12] Luis Damas and Robin Milner. Principal type-schemes for functional languages. In POPL, pages 207 – 212, 1982.

[13] Thierry Despeyroux. Executable specifications of static semantics. In *Proc. Symposium on Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 215 – 233, 1984.

[14] P. Deussen. A decidability criterion for van Wijngaarden grammars. *Acta Informatica*, 5(4):353–375, 1975.

[15] Jay Earley. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

[16] M. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, 1979.

[17] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, 1979.

[18] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[19] T. Johnsson. Efficient Compilation of Lazy Evaluation. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 58–69, Montreal, 1984.

[20] T. Johnsson. Attribute Grammars as a Functional Programming Paradigm. In *Proceedings 1987 Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 274, Portland, Oregon, U.S.A., 1987. Springer Verlag.

[21] Gilles Kahn. Natural semantics. In *Proc. Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer Verlag, 1987.

[22] D. E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Correction in vol. 5, 1, pp 95–96, 1971.

[23] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8:607 – 639, 1965.

[24] Peter Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[25] Peter Landin. A correspondance between ALGOL 60 and Church's lambda notation. *Communications of the ACM*, 8:89–101, 158–165, 1965.

[26] Per Martin-Löf. Constructive Mathematics and Computer Programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153–175. North-Holland, 1982.

[27] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

[28] J. McCarthy. Towards a mathematical science of computation. In Proceedings of the IFIP Congress 1962, pages 21 – 28. North-Holland Publishing Company, 1962.

[29] R. Milner. Standard ML Proposal. *Polymorphism: The ML/LCF/Hope Newsletter*, 1(3), January 1984.

[30] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and Systems Sciences*, 17:348–375, 1978.

[31] Robin Milner. *A Calculcus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[32] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

[33] Kent Petersson. Semantik för programspråk. Föreläsningsanteckningar. Technical report, Institutionen för Informationsbehandling, Chalmers, Göteborg, Sverige, 1985.

[34] Kent Petersson. *Beräkningsbarhet för dataloger. Från λ till P.* Bokförlaget Aquila, 1987.

[35] Gordon Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Aarhus University, September 1981.

[36] David Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, 1986.

[37] D. Scott. Data types as lattices. *SIAM Journal of Computing*, 5:522 – 587, 1976.

[38] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey approach to programming language theory.* MIT Press, 1977.

[39] Christopher Strachey. Towards a formal semantics. In T. B. Steel, editor, *Formal Language Description Languages*, pages 198–220. North-Holland, 1966.

[40] R. D. Tennent. The denotational semantics of programming languages. *Communications of the ACM*, 19(8):437 – 453, 1976.

[41] Göran Uddeborg. Operational Semantics and its Implementation in Prolog. Technical Report 86-01, Department of Computer Science, Chalmers, 1984.

[42] A van Wijngaarden, et al (eds.). Revised report on the algorithmetic language ALGOL 68. ACM *Sigplan Notices*, 12(5):1 – 70, May 1977.

[43] Peter Wegner. The Vienna Definition Language. *ACM Computing Surveys*, pages 5 – 63, 1972.

[44] Niklaus Wirth and C. A. R. Hoare. A Contribution to the Developemnt of ALGOL. *Communications of the ACM*, 9(6):413 – 431, September 1966.