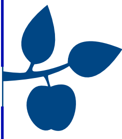


Course Notes DM509

Programming Languages

Peter Schneider-Kamp
petersk@imada.sdu.dk

based on lecture notes by
Jürgen Giesl
giesl@cs.rwth-aachen.de

Department of Mathematics and Computer Science
University of Southern Denmark, Odense

Contents

1 Introduction **3**

2 Logic Programming **5**
2.1 Basic Idea . . . 5
2.2 Prolog Structures . . . 11
2.3 Prolog Evaluation . . . 14
2.4 Arithmetics . . . 18
2.5 Lists . . . 21
2.6 Operators . . . 22
2.7 Cut and Negation-as-Failure . . . 24
2.8 Input/Output . . . 32
2.9 Constraint Programming . . . 35

3 Functional Programming **37**
3.1 Basic Language Constructs of HASKELL . . . 39
3.1.1 Declarations . . . 39
3.1.2 Expressions . . . 52
3.1.3 Patterns . . . 56
3.1.4 Types . . . 60
3.2 Higher-order Functions . . . 65
3.3 Programming using Lazy Evaluation . . . 74
3.4 Input/Output using Monads . . . 78

Chapter 1

Introduction

Today, there are literally thousands of programming languages of which hundreds are in daily use by programmers all over the world. While a computer scientist cannot obtain intimate knowledge of all these languages, there are good reason why he or she should have knowledge of at least some specimens from different families:

- Knowing about different concepts used in programming languages allows to better express one's own ideas when developing software.
- There is never a programming language which is best for all tasks. Background knowledge about different programming languages and concepts is needed to choose the most adequate language for a concrete project.
- Knowing different programming languages and concepts makes it so much easier to learn further programming languages in the future.
- Computer scientists often have to develop new (domain specific) programming languages. Here, knowledge about existing concepts and languages is invaluable.
- In recent times, useful concepts native one family of languages have been successfully ported to other families of languages. Understanding these concepts in their original context aids enormously in understanding these developments.

There are many ways to distinguish and group programming languages by. In this course we do not consider machine or assembly languages but only so-called *high-level* languages. Maybe the most fundamental division that can be made is the one into *imperative* and *declarative* programming languages.

A program written in an *imperative* language basically consists of a sequence of statements manipulating the values of variables in memory together with some special control structures or statements to control which parts of the program are executed. Thus, imperative languages are closely linked to the computer architecture of the John von Neumann machine model. Essentially, imperative programs are a comfortable way to tell the computer exactly *how* to solve a problem and can easily be translated into machine code.

In contrast to this approach, a program written in a *declarative* language consists of a specification telling the computer *what* to do and leaves the details of how to actually

perform the computation to the interpreter or compiler. Thus, declarative programming languages are problem-oriented instead of machine-oriented. Declarative languages are usually further subdivided into *functional* and *logic* programming languages.

In *functional* programming languages, a program is realized as a function potentially being defined with the help of other functions. The execution of a functional program is than the computation of the value of that function for the given inputs.[1] As an example for functional programming we will consider the language Haskell in Chapter 3.

In contrast to that, a *logic* program defines a number of relations. These relations express how different entities are related. In the execution of such a program, the definitions of the relations are used to answer and solve queries. As an example of a logic programming language we will consider the language Prolog in Chapter 2.

In theory, all Turing-complete languages have the same expressive power. That is, every program can be written in any of the conventional programming languages. In practice, different programming languages are of differing adequacy depending on the area of application.

Imperative languages like C are typically used for writing efficient low-level code where the programmer needs to take over the management of memory. Examples are operating system kernels, device drivers, and rendering engines. In other imperative languages like Java, memory is managed automatically by the compiler or the runtime system. This allows to develop programs faster while avoiding a major source of runtime errors. On the other hand, such programs typically require more time and memory than hand-optimized C programs that manage memory on their own.

Functional languages like Haskell are often used for rapid prototyping or developing safety-critical applications. Thanks to efficient compilers such as GHC, Haskell has increasingly been adopted in areas traditionally dominated by imperative languages. Examples for this development are the window manager XMonad or the revision control system Darcs.

Logic languages like Prolog are typically used in the area of artificial intelligence. Examples are expert systems, natural language processing, and deductive databases. Recently, logic languages are also applied for reasoning tasks in the context of the semantic web.

In the rest of this course we will introduce the reader to the principles of logic programming using Prolog and to the concepts of functional programming using Haskell.

For both languages there is a plethora of programming environments. In this course we will make use of Hugs 98 for Haskell and GNU Prolog for Prolog. Virtually all programs from this course should run on any Haskell or Prolog implementation, though.

[1] In fact, an imperative program implicitly also describes a function changing the values of variables.

Chapter 2

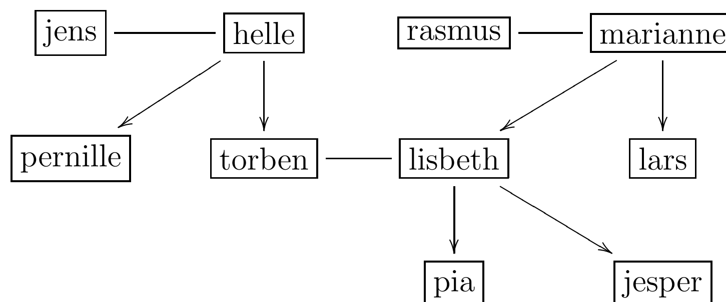
Logic Programming

In Section 2.1 we present the basic idea behind logic programming using a restricted setting corresponding to a subset of Datalog. In Section 2.2 we see that in principle all data Prolog is represented by terms and execution is performed by unification.

Nevertheless, for certain data structures (especially numbers and lists), Prolog offers special syntax and support to improve the efficiency and readability of programs. The treatment of arithmetics and of lists in Prolog is presented in Sections 2.4 and 2.5. In Section 2.6 we introduce the syntax of operators (i.e, functors with infix, prefix or postfix notation). In Section 2.7 we introduce the non-logical *cut* predicate, that enables us to control backtracking and thereby to implement negation. Section 2.8 explains the handling of input and output in Prolog.

2.1 Basic Idea

The main idea of logic programming is that the programmer only describes the logical relations of a problem to be solved, i.e., the database of knowledge about the problem that is available. This description should not require any knowledge over technical details of the computer. Let us consider the following database of knowledge about familiy relation ships as represented in a "familiy tree":



Here, the arrows connect mothers to their children while horizontal lines connect married persons. For the sake of simplicity, we assume there are no divorces, no illegitimate children, and no gay marriages. In the following, we call a database of knowledge a *knowledge base*.

Facts and Queries

To represent a knowledge base in Prolog, we use the language of *predicate logic*. In fact, the name "Prolog" is short for "Programming in Logic".

In Prolog, the knowledge base consists of a number of logical formulas (so-called *clauses*). More precisely, there are two kinds of clauses in a program: *facts*, that express relations between objects, and *rules*, that allow to infer new facts.

In general one should prefer rules over facts to minimize the changes of the program needed to reflect changes to the knowledge base. To describe family relationships under our restrictive assumptions, it is enough to know who is male and who is female, who is married to whom, and who is mother of whom.

The concrete information represented in the preceding graph can be represented as the following list of facts:

```
female(helle).
female(pernille).
female(marianne).
female(lisbeth).
female(pia).

male(jens).
male(torben).
male(rasmus).
male(lars).
male(jesper).

married(jens, helle).
married(rasmus, marianne).
married(torben, lisbeth).

motherOf(helle, pernille).
motherOf(helle, torben).
motherOf(marianne, lisbeth).
motherOf(marianne, lars).
motherOf(lisbeth, jesper).
motherOf(lisbeth, pia).
```

A fact always consists of the name of the relation (the so-called *predicate symbol*) and a list of the objects being in relation. All clauses end in a dot. In Prolog, objects (like `helle`) and relations (like `motherOf`) begin with a lowercase letter. The number of arguments is called the *arity* of the predicate symbol. A predicate can be seen as a function whose result is either "true" oder "false". In our example, we see that relations have a direction. If an object is in relation with another object, the reverse does not necessarily hold. Consider for example the fact `motherOf(helle, pernille)`. The reverse, i.e., `motherOf(pernille, helle)`, does not hold. Line comments in Prolog begin with `%` while multi-line comments are enclosed in `/*` and `*/`.

As mentioned before, a logic program is executed by the user posing *queries* to the knowledge base. Thus, logic languages are "dialogue-oriented" programming languages. A possible question would for example be "Is `rasmus male`?". In Prolog, queries start with "`?-`" followed by the facts to be proven. The individual facts are separated by commas and the whole query is ended by a dot. Multiple facts in a query means that the conjunction of all these facts has to be proven.

Thus, in Prolog our example query could be written in the following way:

```
?- male(rasmus).
```

Now, Prolog computes the answer by performing a logical proof that this fact follows from the knowledge base. The task of the computer is, thus, to infer a solution to the query from the knowledge base. In other words, "computing" in Prolog means "proving". The base technologies needed for these automated proofs are "*unification*" and "*resolution*". In the preceding example, the computer would answer `yes`.

In contrast, for the query

```
?- married(rasmus, helle).
```

we obtain the answer `no`. The reason is the so-called "closed world assumption", i.e., the assumption that our knowledge base contains all relevant knowledge. If a certain fact does not follow from our knowledge, we consider it to be "false". The fact `married(rasmus, helle)` does not follow from our knowledge base and, thus, has to be considered false.

To be able to pose queries to a logic program, the program has, of course, to be loaded first. [1] After loading, we can use the knowledge base defined in the program.

Programs with Variables

The knowledge base may also contain *variables*. In Prolog, variables start with an uppercase letter or an underscore. For example we might add the following fact to our knowledge base:

```
human(X).
```

Variables in the knowledge base represent *all* possible objects. Thus, this fact means "All objects are humans.". If we now ask the query

```
?- human(rasmus).
```

we obtain the expected answer `yes`. But also the query "`?- human(5).`" would lead to the answer `yes`.

Identical variables in the same fact signify equal objects. Thus, the fact "`likes(X,Y).`" means "Everyone like everyone.". On the other hand, "`likes(X,X).`" means "Everyone only likes herself or himself.". Identical variables in two different facts (and clauses) do not entail any restriction, though.

[1] GNU Prolog can be started on the machines in the terminal room using the `gprolog` command. To load a program from a file `family.pl`, enter the query "`consult(family).`" (or shorter "`[family].`").

Queries with Variables

It also possible to let the program compute solutions. To this end one uses queries with variables. Let us for example consider the following query:

```
?- motherOf(X, lisbeth).
```

This corresponds to the question "Who is the mother of `lisbeth`?" or, more formally, "Is there an assignment of the variable `X`, such that `X` is the mother of `lisbeth`?". Now, the computer does not only answer with `es`, but it also looks for a satisfying assignment of `X`. Thus, the answer is `X = marianne`. All variables in facts are *universally quantified* while variables in queries are *existentially quantified*.

As another example consider the following query:

```
?- motherOf(marianne, Y).
```

This corresponds to the question "Who are the kids of marianne?" or, more formally, "What are the possible assignments of the variable `Y`, such that `Y` is a child of `marianne`?". The computer answers with "`Y = lisbeth`". By pressing "Return", one obtains a simple "`yes`", as the question has been answered positively.

But obviously this is not the only solution. If we want the computer to compute further solutions, we have to press the semicolon. Then, the computer looks for further solutions and we obtain "`Y = lars`". If we press semicolon again, we obtain "`no`", because `marianne` has no further children.

We observe that `motherOf` is no function mapping inputs to outputs, but a relation. What we consider to be the input and what to be the output is not determined by the program, but only by the query. Thus, we can use a single program to compute all children of a mother given the mother or to compute the mother of a child given the child. Of course, we could also pose the query "`?- motherOf(X,Y).`" corresponding to the question "What pairs of mother and children do we know about?". In the end, it is the user of the program who determines which values to provide in a query and which values the program should compute.

To execute such queries, Prolog scans the knowledge base from beginning to end, returning the first answer that fits. *In other words, the clauses of the knowledge base are used from top to bottom.*

As a further example, let us consider the following query:

```
?- human(X).
```

Every possible instantiation of the variable `X` would be a solution. The computer computes the "most general" solution. In this case, it will answer "`yes`" as the relation holds for every instantiation of `X`. In general, the computer always finds the *most general* answer to a query, such that the answer is valid for every possible instantiation of the variables.

Combining Queries

As mentioned before, we can use the comma to combine queries as a conjunction. As an example, consider the following query:

```
?- married(rasmus,F), motherOf(F,lisbeth).
```

Here, the question is "Is there a woman `F` that is married to `rasmus` and at the same time is the mother of `lisbeth`?". For this combination to make sense, the variable `F` has to be instantiated in the same way in all parts of the query.

Prolog proceeds by first finding a solution for the goal "`married(rasmus,F)`". This solution instantiates `F` to `marianne`. Then, with this solution it tries to solve "`motherOf(F,lisbeth)`". In other words, it tries to prove "`motherOf(marianne,lisbeth)`". If that does not work, the computer goes back and tries to find a different instantiation for `F` that also satisfies "`married(rasmus,F)`". *Goals in a query are processed from left to right.*

As another example, let us consider the question who is the grandmother of `jesper` from his mother's side:

```
?-  motherOf(Granny,Mom), motherOf(Mom,jesper).
```

Here, the first solution for the first goal is `Granny = helle, Mom = pernille`. With this solution, we try to prove "`motherOf(pernille,jesper)`". As we cannot prove this, we have to backtrack until we find the solution `Granny = marianne, Mom = lisbeth`. If we had swapped the two goals `motherOf(Oma,Mama)` and `motherOf(Mama,jesper)`, the solution would have been found without backtracking.

Rules

Besides *facts*, the knowledge base can also contain *rules*. Rules are needed to infer new facts from given facts. As an example, let us consider the relation of fathers and their children. We might, of course, define this relation for all the objects in our knowledge base explicitly by adding appropriate facts. But it is much shorter and easier to understand if we use a general rule for this relation. [2] The following rule states: "A person `V` is the father of a child `K`, if he is married to a woman `F` and this woman is the mother of the child `K`.".

```
fatherOf(V,K) :- married(V,F), motherOf(F,K).
```

Here, the sign "`:-`" means "if" and rules formulate "if - then" relationships. Whenever the premise on the right-hand side of the rule holds, the statement on the left-hand side holds, too. The left-hand side is called the *head* of a rule and the right-hand side is called the *body*. The premises in the body are separated by commas and the whole rule is, of course, ended by a dot. The meaning of a rule `p :- q,r.` is: If `q` and `r` hold, then also `p` holds.

During execution (i.e., proving) in Prolog, rules are applied backwards. To show that the left-hand side of a rule holds, one has to show that the right-hand side holds. This proof method is called *backward chaining*.

As an example, consider the following query:

```
?- fatherOf(rasmus,lisbeth).
```

[2] By using rules we can also describe infinite sets of objects and our knowledge base is more likely to stay consistent if we add or remove facts.

To prove this goal, due to the rule for `fatherOf`, we have to find a value for `F`, such that `married(rasmus,F), motherOf(F,lisbeth)` holds. This exactly corresponds to the query `?- married(rasmus,F), motherOf(F,lisbeth).` Thus, Prolog answers "`yes`". Analogously, for the query

```
?- fatherOf(rasmus,Y).
```

we obtain the answers `Y = lisbeth` and `Y = lars`.

Multiple Rules per Predicate

Until now we have only defined rules, where the head holds, when the *conjunction* of the premises holds. Now, we also consider the case where the head follows from the *disjunction* of two premises. To this end, we use multiple rules for the same predicate symbol. While these rules can be placed anywhere in the program, it is good style to group such rules (just as we did with the facts).

As an example, let us consider a predicate `parent` where `parent(X,Y)` should hold, if `X` is the mother or the father of `Y`. The rules for this predicate are:

```
parent(X,Y) :- motherOf(X,Y).
parent(X,Y) :- fatherOf(X,Y).
```

If we now pose the query

```
?- parent(X, lisbeth).
```

we obtain the two answers `X = marianne` and `X = rasmus`. We observe that the order of the clauses influences the search for and, thus, the order of the solutions.

Recursive Rules

Recursion ist an important concept for programming in Prolog. For example, we define a predicate `ancestor`. Here, an object `V` is an ancestor of `X`, if `V` is a parent of `X` or there is a `Y`, such that `V` is parent of `Y` (i.e., `V` has a child `Y`) and `Y` is an ancestor of `X`. The translation of this rule to Prolog yields:

```
ancestor(V,X) :- parent(V,X).
ancestor(V,X) :- parent(V,Y), ancestor(Y,X).
```

Now, the query

```
?- ancestor(X, jesper).
```

means "Who are the ancestors of `jesper`?". Here, Prolog finds the following answers:

```
X = lisbeth;
X = torben;
X = helle;
X = marianne;
X = jens;
X = rasmus
```

Characteristics of Logic Programs

We observe the following characteristics of logic programs:

1. Pure logic programs contain no control structures for controlling program execution. Programs are just a collection facts and rules that are processed from top to bottom and from left to right.

2. Logic programming emerged from automated proving, and, indeed, when executing a logic program, we try to prove a query. In addition to proving whether a query holds, we also compute solutions for the variables occurring in the query. Which arguments are used for input and which are used for output is not determined beforehand.

3. Logic programs are often used in applications from the area of artificial intelligence. For example, they are very adequate to implement expert systems, where the rules of the program are formed by the knowledge of the expert. Further main areas of application are deductive databases and rapid prototyping.

2.2 Prolog Structures

So far we have only considered a restricted subset of Prolog programs consisting of facts and rules built over predicate symbols with a list of variables and constants for their arguments. The objects of the relations were, thus, limited to be either variables or constants. We now introduce the concept of terms which are the unified representation of all elements in Prolog programs. In fact, we will see that even programs are just terms. This comes in handy for example when one considers programs as data.

Terms in Prolog

Terms are trees where every leaf node is labelled with a constant or a variable and all inner nodes are labelled with a constant. If an inner node has n children $t_1, \ldots, t_n$ and is labelled by a constant f, we say that f is a functor of arity n and $f(t_1, \ldots, t_n)$ is a structure. [3]

As an example, let us consider the fact "`human(X)`". When representing this fact as a term, the root node is labelled by the functor `human` of arity 1 and the only child is a node labelled by the variable `X`.

Using terms, we can for example represent natural numbers by terms using the unary functor `s` for the successor function and the constant `0` for zero. To represent the number 3 we would then use the structure `s(s(s(0)))`.

Prolog allows to overload functors, i.e, to use functors with different arities to build terms. Consider for example the following rule:

```
p(a,p) :- p(c(a,b,b,a)).
```

Here, the left-most occurrence of `p` is a binary functor representing a predicate. One of its arguments is a constant `p` which is completely independent from the binary predicate `p`. Finally, the right-hand side of the rule contains a unary functor `p` representing another

[3] A constant is actually nothing but a functor of arity 0.

predicate distinct from the one represented by the binary functor `p`. To identify functors in the context of possible overloading, we usually give their arity after the name and a slash. Thus the left-most binary functor `p` would be identified as `p/2`, its constant argument `p` as `p/0`, and the unary functor `p` on the right-hand side of the rule would be `p/1`. Furthermore, the rule contains the functors `a/0`, `b/0`, and `c/4`.

Unification

In logic programming, facts and rules are applied by unifying a fact or the head of a rule with the leftmost goal of our query. Here, two terms t_1 and t_2 unify if they can be instantiated to the same term u.

To make this notion more formal, we introduce the concept of a *substitution*. A substitution σ is a function mapping variables to terms such that $\sigma(X) \neq X$ for only finitely many variables X. Thus, we can represent a substitution by a finite set of pairs of variables and terms. We extend substitutions to work on terms by defining $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$. Instead of $\sigma(t)$ we often write $t\sigma$.

Now, two terms t_1 and t_2 unfiy if, and only if, there is a substitution σ such that $t_1\sigma = t_2\sigma$. In this case, the substitution σ is called a *unifier* of t_1 and t_2. A unifier σ of two terms t_1 and t_2 is called a *most general unifier* (mgu) if for all unifiers μ of t_1 and t_2, there is a substitution δ such that $t_1\sigma\delta = t_1\mu$. If two terms unfiy, the most general unifier is unique up to renaming of the variables.

As an example, consider the terms `add(X,s(Y),Z)` and `add(s(0),s(s(0)),U)`. The substitution μ = {`X/s(0), Y/s(0), Z/0, U/0`} is a unifier of these two terms. But it is not a most general unifier as it needlessly instantiates `Z` and `U` to `0`. The most general unifier is σ = {`X/s(0), Y/s(0), Z/U`} and indeed, for δ = {`Z/0, U/0`} we have `add(X,s(Y),Z)`$\sigma\delta$ =`add(s(0),s(s(0)),U)`δ =`add(s(0),s(s(0)),0)`=`add(X,s(Y),Z)`μ.

There are many different algorithms to compute the most general unifier of two terms. We will use one which is based on simplifying a set of equations until we obtain a substitution. If we want to unify two terms t_1 and t_2, we start with the initial set $\{t_1 \stackrel{?}{=} t_2\}$.

There are four rules to simplify these sets:

Delete
$\{t \stackrel{?}{=} t\} \uplus S \quad \Rightarrow \quad S$

Decompose
$\{f(t_1, \ldots, t_n) \stackrel{?}{=} f(u_1, \ldots, u_n)\} \uplus S \quad \Rightarrow \quad \{t_1 \stackrel{?}{=} u_1, \ldots, t_n \stackrel{?}{=} u_n\} \cup S$

Orient
$\{t \stackrel{?}{=} X\} \uplus S \quad \Rightarrow \quad \{X \stackrel{?}{=} t\} \cup S$ if t is not a variable

Eliminate
$\{X \stackrel{?}{=} t\} \uplus S \quad \Rightarrow \quad \{X \stackrel{?}{=} t\} \cup S\{X/t\}$ if X occurs in S, but does not occur in t

When we obtain a substitution, i.e., a set of equations such that the left-hand sides of

the equations are distinct variables and the right-hand sides are terms not containing any of these variables, we say that the set is in *solved form*.

As an example, consider again the two terms `add(X,s(Y),Z)` and `add(s(0),s(s(0)),U)`. We start with the initial set $\{\texttt{add}(\texttt{X},\texttt{s}(\texttt{Y}),\texttt{Z}) \stackrel{?}{=} \texttt{add}(\texttt{s}(\texttt{0}),\texttt{s}(\texttt{s}(\texttt{0})),\texttt{U})\}$. By applying Decompose we obtain $\{\texttt{X} \stackrel{?}{=} \texttt{s}(\texttt{0}), \texttt{s}(\texttt{Y}) \stackrel{?}{=} \texttt{s}(\texttt{s}(\texttt{0})), \texttt{Z} \stackrel{?}{=} \texttt{U})\}$. By applying Decompose again, we obtain $\{\texttt{X} \stackrel{?}{=} \texttt{s}(\texttt{0}), \texttt{Y} \stackrel{?}{=} \texttt{s}(\texttt{0}), \texttt{Z} \stackrel{?}{=} \texttt{U}\}$. Thus, we have already reached solved form.

Now, the following example demonstrates the use of all four rules. We start with the two terms $\texttt{g}(\texttt{X},\texttt{g}(\texttt{X},\texttt{X}))$ and $\texttt{g}(\texttt{f}(\texttt{a}),\texttt{g}(\texttt{X},\texttt{Y}))$. We obtain the following sequence of simplifications: $\{\texttt{g}(\texttt{X},\texttt{g}(\texttt{X},\texttt{X})) \stackrel{?}{=} \texttt{g}(\texttt{f}(\texttt{a}),\texttt{g}(\texttt{X},\texttt{Y}))\} \stackrel{\text{Decompose}}{\Rightarrow} \{\texttt{X} \stackrel{?}{=} \texttt{f}(\texttt{a}), \texttt{g}(\texttt{X},\texttt{X}) \stackrel{?}{=} \texttt{g}(\texttt{X},\texttt{Y})\} \stackrel{\text{Eliminate}}{\Rightarrow} \{\texttt{X} \stackrel{?}{=} \texttt{f}(\texttt{a}), \texttt{g}(\texttt{f}(\texttt{a}),\texttt{f}(\texttt{a})) \stackrel{?}{=} \texttt{g}(\texttt{f}(\texttt{a}),\texttt{Y})\} \stackrel{\text{Decompose}}{\Rightarrow} \{\texttt{X} \stackrel{?}{=} \texttt{f}(\texttt{a}), \texttt{f}(\texttt{a}) \stackrel{?}{=} \texttt{f}(\texttt{a}), \texttt{f}(\texttt{a}) \stackrel{?}{=} \texttt{Y}\} \stackrel{\text{Delete}}{\Rightarrow} \{\texttt{X} \stackrel{?}{=} \texttt{f}(\texttt{a}), \texttt{f}(\texttt{a}) \stackrel{?}{=} \texttt{Y}\} \stackrel{\text{Orient}}{\Rightarrow} \{\texttt{X} \stackrel{?}{=} \texttt{f}(\texttt{a}), \texttt{Y} \stackrel{?}{=} \texttt{f}(\texttt{a})\}$

In these cases, the algorithm found the most general unifier of the two terms. In general, if we can simplify a set of equations into solved form, we obtain a most general unifier of the initial set of equations. In the following we learn why this is the case.

When applying these four rules as long as possible, we always obtain a set of equations that cannot be simplified anymore. In other words, the unification algorithm always terminates.

To see this, consider the mapping from a set of equations S to a triple (n_1, n_2, n_3) where

- n_1 is the number of variables in S that are not solved,
- n_2 is the number of all occurrences of variables and functions in S, and
- n_3 is the number of equations $t \stackrel{?}{=} X$ in S where t is not a variable.

We call a variable X *solved* if it only occurs once in S in an equation $X \stackrel{?}{=} t$.

For example, the problem $\{\texttt{add}(\texttt{X},\texttt{s}(\texttt{Y}),\texttt{Z}) \stackrel{?}{=} \texttt{add}(\texttt{s}(\texttt{0}),\texttt{s}(\texttt{s}(\texttt{0})),\texttt{U})\}$ is mapped to the triple $(3, 12, 0)$, the problem $\{\texttt{X} \stackrel{?}{=} \texttt{s}(\texttt{0}), \texttt{s}(\texttt{Y}) \stackrel{?}{=} \texttt{s}(\texttt{s}(\texttt{0})), \texttt{Z} \stackrel{?}{=} \texttt{U})\}$ to $(1, 10, 0)$, and, finally, $\{\texttt{X} \stackrel{?}{=} \texttt{s}(\texttt{0}), \texttt{Y} \stackrel{?}{=} \texttt{s}(\texttt{0}), \texttt{Z} \stackrel{?}{=} \texttt{U}\}$ to $(0, 8, 0)$.

We can show that these triples decrease with respect to the lexicographic extension of the $>$ relation on natural numbers for every simplification rule:

	n_1	n_2	n_3
Delete	$\geq$	$>$	
Decompose	$\geq$	$>$	
Orient	$\geq$	$=$	$>$
Eliminate	$>$		

Next we show that the application of the four simplification rules does not change the set of unifiers of the respective unification problem, i.e, we show that all unifiers of S are also unifiers of S' if we have $S \Rightarrow S'$. When using Delete, Decompose, or Orient to go from a set S to a set S', the set of unifiers of S is obviously the same as the set of unifiers of S'. Now,let $\theta = \{X/t\}$ where X does not occur in t and, thus, $\{X \stackrel{?}{=} t\} \uplus S \stackrel{\text{Eliminate}}{\Rightarrow} \{X \stackrel{?}{=} t\} \cup S\theta$. Then, for any unifier σ of $\{X \stackrel{?}{=} t\} \uplus S$, from

$X\sigma = t\sigma$ and X does not occur in t we obtain $\sigma = \theta\sigma$ Then, clearly $\theta\sigma$ is also a unifier of $\{X \stackrel{?}{=} t\} \uplus S$ if, and only if, σ is a unifier of $\{X \stackrel{?}{=} t\} \cup S\theta$.

Now, we also need to show that whenever we cannot apply any more simplification rules and have not reached a solved form, the terms do not unify. There are two cases to consider. Consider for example the terms `f(a)` and `f(b)`. By one application of the DECOMPOSE rule we obtain $\{\mathtt{a} \stackrel{?}{=} \mathtt{b}\}$. This is cannot be simplified any further, but is clearly not in solved form. Obviously, there is no substitution that can make `a` and `b` equal. We call this failure a *clash-failure*. For another kind of failure, consider the terms `X` and `f(X)`. The problem $\{\mathtt{X} \stackrel{?}{=} \mathtt{f(X)}\}$ cannot be simplified, but is not in solved form, because `X` occurs in `f(X)`. Assume there was a substitution σ that is a unifier of `X` and `f(X)`. Then it would replace X by some term t. But then $X\sigma = t \neq f(t) = f(X)\sigma$. Thus, these two terms do not unify. We call this kind of failure an *occur-failure*.

We are left to show that if we obtain a set of equations S' in solved form from a set S, it actually represents a most general unifier of S' and, therefore, of S. Let σ be the substitution corresponding to $S' = \{X_1/t_1, \ldots, X_n/t_n\}$. As S' is in solved form, no X_i occurs in any of the t_j. Hence, $X_i\sigma = t_i = t_i\sigma$ for all i and, consequently, σ is a unifier of S'. Let μ be another unifier of S. Then for all X_i, we have $X_i\mu = t_i\mu = X_i\sigma\mu$, and for $X \notin \{X_1, \ldots, X_n\}$, we have $X\mu = X\sigma\mu$ because $X\sigma = X$. Thus, σ is a most general unifier of S' and S. This concludes our proof of the correctness of the unification algorithm.

2.3 Prolog Evaluation

In this section we formally describe how evaluation in Prolog works. To this end we introduce predicate logic and the logic proof method of resolution. We then specialize these concepts for the evaluation mechanism used in Prolog.

Predicate Logic

The basic building blocks of predicate logic are atoms, i.e., predicate symbols applied to a list of arguments that are terms. Formulas in predicate logic are built from such atoms $\mathcal{A}t$, Boolean operators, and existential and universal quantifiers over variables. Thus, the set $\mathcal{F}$ of predicate logic formulas is the smallest set such that:

- $\mathcal{A}t \subseteq \mathcal{F}$,
- if $\varphi \in \mathcal{F}$, then $\neg\varphi \in \mathcal{F}$
- if $\varphi_1, \varphi_2 \in \mathcal{F}$, then $(\varphi_1 \wedge \varphi_2), (\varphi_1 \vee \varphi_2), (\varphi_1 \rightarrow \varphi_2), (\varphi_1 \leftrightarrow \varphi_2) \in \mathcal{F}$
- if $\varphi \in \mathcal{F}$ and X is a variable, then $(\forall X.\varphi), (\exists X.\varphi) \in \mathcal{F}$

Conversion to Clausal Form

We call a formula just consisting of one atom or a negated atom a *literal*. A formula is in *conjunctive normal form (CNF)* if it is a conjunction of disjunctions of literals. For example, consider the formula $p \wedge (\neg p \vee q) \wedge (\neg q \vee \neg p)$ which is in CNF. Here, p can be

seen as a disjunction of just one literal. We often denote a formula in CNF by a set of sets of literal. For the formula $p \wedge (\neg p \vee q) \wedge (\neg q \vee \neg p)$, we write $\{\{p\}, \{\neg p, q\}, \{\neg q, \neg p\}\}$.

Any formula can be transformed into an equisatisfiable formula where all variables are universally quantified and the quantifier-free part is in CNF. We call this form *clausal form* and we will obtain it by a number of transformations. As an example, we consider the formula $(\forall X.p(X)) \vee \neg(\forall X.(q(X) \rightarrow p(X)))$.

We start by replacing all formulas $(\varphi_1 \leftrightarrow \varphi_2)$ by $(\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \leftarrow \varphi_1)$. Then we replace all formulas $(\varphi_1 \rightarrow \varphi_2)$ by $(\neg\varphi_1 \vee \varphi_2)$. This transformation leaves us with only $\neg, \vee, \wedge$ as the Boolean operators. In our example, we obtain $(\forall X.p(X)) \vee \neg(\forall X.(\neg q(X) \vee p(X)))$.

Next, we transform the formula into negation normal form. In this form, negations occur only in the form of negated literals. We obtain this form by repeatedly applying the following transformation steps:

- replace $\neg(\varphi_1 \vee \varphi_2)$ by $(\neg\varphi_1 \wedge \neg\varphi_2)$ (de Morgan's first law)
- replace $\neg(\varphi_1 \wedge \varphi_2)$ by $(\neg\varphi_1 \vee \neg\varphi_2)$ (de Morgan's second law)
- replace $\neg(\forall X.\phi)$ by $(\exists X.\neg\phi)$
- replace $\neg(\exists X.\phi)$ by $(\forall X.\neg\phi)$
- replace $\neg\neg\phi$ by ϕ

In our example, we obtain $(\forall X.p(X)) \vee (\exists X.(q(X) \wedge \neg p(X)))$.

By renaming the quantified variables appropriately, we can move all quantifiers to the outside. This form is called prenex normal form. In our example, we first rename the existentially qualified X to Y and move the quantifiers to the outside. We obtain $(\forall X.(\exists Y.(p(X) \vee (q(Y) \wedge \neg p(Y)))))$.

To get rid of existential quantifiers, we replace all existentially qualified variables by terms with a new functor and all universally quantified variables as arguments. This process is called *skolemization* and the resulting form is called the *Skolem normal normal*. In our example, we replace Y by $f(X)$ and obtain $(\forall X.(p(X) \vee (q(f(X)) \wedge \neg p(f(X)))))$.

With all variables universally quantified from the outside, we consider only the quantifier-free part of the formula. In our example, this is $(p(X) \vee (q(f(X)) \wedge \neg p(f(X))))$.

Finally, we transform the quantifier-free part into conjunctive normal form by repeatedly applying the following transformation steps:

- replace $(\varphi_1 \vee (\varphi_2 \wedge \varphi_3))$ by $((\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3))$ (distribution of $\vee$ over $\wedge$)
- replace $((\varphi_1 \wedge \varphi_2) \vee \varphi_3)$ by $((\varphi_1 \vee \varphi_3) \wedge (\varphi_2 \vee \varphi_3))$ (distribution of $\vee$ over $\wedge$)

In our example, we obtain $((p(X) \vee q(f(X))) \wedge (p(X) \vee \neg p(f(X))))$.

Resolution

We can now express logic programs using first order formulas. The logic program from Section 2.1 can be expressed as follows:

$$
\begin{array}{lrcl}
 & & & \mathsf{female}(\mathsf{helle}) \\
 & & & \mathsf{female}(\mathsf{pernille}) \\
 & & & \mathsf{female}(\mathsf{marianne}) \\
 & & & \mathsf{female}(\mathsf{lisbeth}) \\
 & & & \mathsf{female}(\mathsf{jesper}) \\
 & & & \\
 & & & \mathsf{male}(\mathsf{jens}) \\
 & & & \mathsf{male}(\mathsf{torben}) \\
 & & & \mathsf{male}(\mathsf{rasmus}) \\
 & & & \mathsf{male}(\mathsf{lars}) \\
 & & & \mathsf{male}(\mathsf{pia}) \\
 & & & \\
 & & & \mathsf{married}(\mathsf{jens}, \mathsf{helle}) \\
 & & & \mathsf{married}(\mathsf{rasmus}, \mathsf{marianne}) \\
 & & & \mathsf{married}(\mathsf{torben}, \mathsf{lisbeth}) \\
 & & & \\
 & & & \mathsf{motherOf}(\mathsf{helle}, \mathsf{pernille}) \\
 & & & \mathsf{motherOf}(\mathsf{helle}, \mathsf{torben}) \\
 & & & \mathsf{motherOf}(\mathsf{marianne}, \mathsf{lisbeth}) \\
 & & & \mathsf{motherOf}(\mathsf{marianne}, \mathsf{lars}) \\
 & & & \mathsf{motherOf}(\mathsf{lisbeth}, \mathsf{jesper}) \\
 & & & \mathsf{motherOf}(\mathsf{lisbeth}, \mathsf{pia}) \\
 & & & \\
\forall X. & & & \mathsf{human}(X) \\
 & & & \\
\forall V, F, K. & \mathsf{married}(V, F) \wedge \mathsf{motherOf}(F, K) & \rightarrow & \mathsf{fatherOf}(V, K) \\
 & & & \\
\forall X, Y. & \mathsf{motherOf}(X, Y) & \rightarrow & \mathsf{parent}(X, Y) \\
\forall X, Y. & \mathsf{fatherOf}(X, Y) & \rightarrow & \mathsf{parent}(X, Y) \\
 & & & \\
\forall V, X. & \mathsf{parent}(V, X) & \rightarrow & \mathsf{ancestor}(V, X) \\
\forall V, Y, X. & \mathsf{parent}(V, Y) \wedge \mathsf{ancestor}(Y, X) & \rightarrow & \mathsf{ancestor}(V, X)
\end{array}
$$

To answer the query "`?- motherOf(X,lisbeth).`", we need to show that the formula $(\exists X.\mathsf{motherOf}(X, \mathsf{lisbeth}))$ follows from the conjunction of all the formulas that make up the program. For this query, it will be enough to show that $(\exists X.\mathsf{motherOf}(X, \mathsf{lisbeth}))$ follows from $\mathsf{motherOf}(\mathsf{marianne}, \mathsf{lisbeth})$.

Instead of showing that φ follows from $\varphi_1 \wedge \ldots \wedge \varphi_n$, we can show that $\neg\varphi \wedge \varphi_1 \wedge \ldots \wedge \varphi_n$ is unsatisfiable. In our example, we have to show that $\neg(\exists X.\mathsf{motherOf}(X, \mathsf{lisbeth})) \wedge \mathsf{motherOf}(\mathsf{marianne}, \mathsf{lisbeth})$ is unsatisfiable.

Using the preceding transformation to clausal form, this is equivalent to showing unsatisfiability of $(\forall X.(\neg\mathsf{motherOf}(X, \mathsf{lisbeth}) \wedge \mathsf{motherOf}(\mathsf{marianne}, \mathsf{lisbeth})))$ or in quantifier-free

set notation $\{\{\neg\mathsf{motherOf}(X, \mathsf{lisbeth}\}, \{\mathsf{motherOf}(\mathsf{marianne}, \mathsf{lisbeth})\}\}$.

For any literal p, we define the *negated literal* $\overline{p}$ as $\neg p$ if p is an atom and q if $p = \neg q$ for some atom q. Thus, in our example we have $\overline{\neg\mathsf{motherOf}(X, \mathsf{lisbeth})} = \mathsf{motherOf}(X, \mathsf{lisbeth})$ and $\overline{\mathsf{motherOf}(\mathsf{marianne}, \mathsf{lisbeth})} = \neg\mathsf{motherOf}(\mathsf{marianne}, \mathsf{lisbeth})$.

For two clauses $A = \{\alpha_1, \ldots, \alpha_n\}$ and $B = \{\beta_1, \ldots, \beta_n\}$ in set notation, the clause $R = \{\alpha_{i+1}\sigma, \ldots, \alpha_n\sigma, \beta_{j+1}\sigma, \ldots, \beta_n\sigma\}$ is the *resolvent* of A and B if there is a most general unifier σ of the unification problem $\{\overline{\alpha_1}, \ldots, \overline{\alpha_i}, \beta_1, \ldots \beta_j\}$ for $i, j \geq 1$. In our example, for $i = j = 1$ we have $\{X/\mathsf{marianne}\}$ as the most general unifier of $\overline{\neg\mathsf{motherOf}(X, \mathsf{lisbeth})} = \mathsf{motherOf}(X, \mathsf{lisbeth})$ and $\mathsf{motherOf}(\mathsf{marianne}, \mathsf{lisbeth})$. The resolvent is $\emptyset$ in this case, i.e., the empty disjunction. Thus, we have shown unsatisfiability of our original formula and the truth of the query "`?- motherOf(X,lisbeth).`" with the answer `X = lisbeth`.

This general *resolution* for predicate logic is sound and complete, but unfortunately not very practical as the search space of which clauses to resolve is immense. When restricted to *Horn clauses* (i.e., clauses with at most one positive literal), we can restrict ourselves to *binary input resolution*. Here, we always use the resolvent of the last step together with a clause corresponding to the original clauses of the program (input resolution). We also restrict $i = j = 1$ (binary resolution). If we additionally consider the order of literals in our clause from left to right and the order of clauses of the program from top to bottom, we obtain *SLD Resolution*.

For a more involved example, consider the query "`?- fatherOf(X,torben).`" or as a formula $(\exists X.\mathsf{fatherOf}(X, \mathsf{torben}))$. For the sake of simplicity, we will restrict the program to the facts and rules needed to obtain the first (and only) answer to the question. Thus, we have to prove unsatisfiability of $\neg(\exists X.\mathsf{fatherOf}(X, \mathsf{torben})) \wedge \mathsf{married}(\mathsf{jens}, \mathsf{helle}) \wedge \mathsf{motherOf}(\mathsf{helle}, \mathsf{torben}) \wedge (\forall V, F, K.\mathsf{married}(V, F) \wedge \mathsf{motherOf}(F, K) \rightarrow \mathsf{fatherOf}(V, K))$. Thus, we show unsatisfiability of the clausal form $\{\{\neg\mathsf{fatherOf}(X, \mathsf{torben})\}, \{\mathsf{married}(\mathsf{jens}, \mathsf{helle})\}, \{\mathsf{motherOf}(\mathsf{helle}, \mathsf{torben})\}, \{\neg\mathsf{married}(V, F), \neg\mathsf{motherOf}(F, K), \mathsf{fatherOf}(V, K)\}\}$.

We start by resolving the clause corresponding the query $\{\neg\mathsf{fatherOf}(X, \mathsf{torben})\}$ and the program clause $\{\neg\mathsf{married}(V, F), \neg\mathsf{motherOf}(F, K), \mathsf{fatherOf}(V, K)\}\}$ using $\{V/X, K/\mathsf{torben}\}$. We obtain as a resolvent the clause $\{\neg\mathsf{married}(X, F), \neg\mathsf{motherOf}(F, \mathsf{torben})\}$.

Next, we resolve this new clause with the program clause $\{\mathsf{married}(\mathsf{jens}, \mathsf{helle})\}$ using $\{X/\mathsf{jens}, F/\mathsf{helle}\}$ and obtain the resolvent $\{\neg\mathsf{motherOf}(\mathsf{helle}, \mathsf{torben})\}$. Finally, we resolve this new clause with $\{\mathsf{motherOf}(\mathsf{helle}, \mathsf{torben})\}$ and obtain the empty clause $\emptyset$. Thus, we have proved our query with the answer `X = jens`.

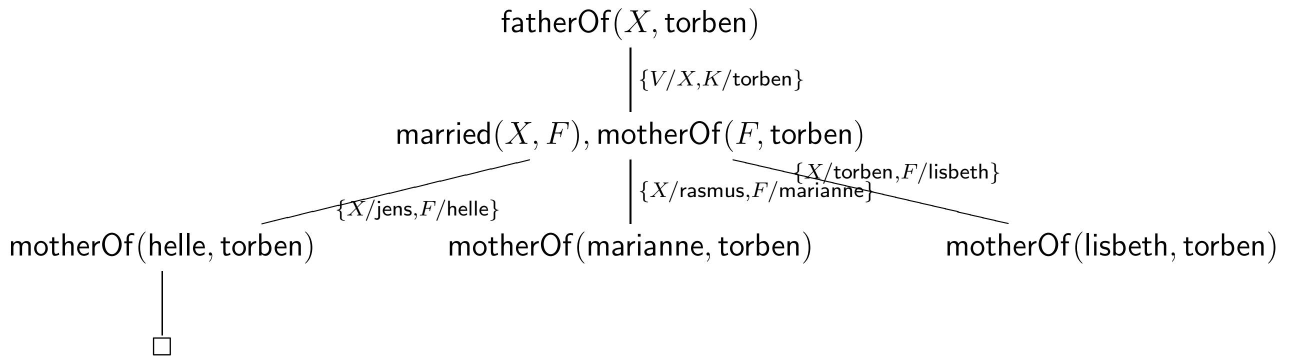
SLD Tree

Typically we do not know exactly which clauses of the program are needed to prove a given query, i.e., with which clauses to perform resolution. To visualize the process of trying clauses from top to bottom and literals in the query from left to right, we introduce *SLD trees*.

Each node contains the resolvent of the previous node and a program clause. The order of the children is important and corresponds to the top to bottom order of the clauses in the program.

For the preceding program and the query "`?- fatherOf(X,torben)`" we obtain the

following SLD tree:



The only solution is `X = jens` following the left-most branch of the tree. The other two leaves represent failed attempts. We call these failed attempts *finite failures*.

2.4 Arithmetics

To compute with natural numbers, we can represent them as terms using `s` and `0` as in the previous section. The following logic program can be used to compute addition:

```
add(X,0,X).
add(X,s(Y),s(Z)) :- add(X,Y,Z).
```

Here, "`add(X,Y,Z)`" corresponds to the statement "`X + Y = Z`". If one now poses a query, where the first two arguments of `add` are given, this program computes addition. For the query "`?- add(s(0),s(s(0)),X).`" we obtain the answer `X = s(s(s(0)))`.

As mentioned before, in a logic program it is not predetermined which arguments have to be given and which are to be computed. Thus, this program can also be used for subtraction by giving the second and the third argument and asking for the first. To compuer "3-2", we pose the query "`?- add(X,s(s(0)),s(s(s(0)))).`" and obtain the answer `X = s(0)`.

We can also use this programn to computer all pairs of natural numbers for which the sum is a given number. For example, if we pose the query "`?- add(X,Y,s(s(s(0)))).`" we obtain the solutions `X = s(s(s(0))), Y = 0`, `X = s(s(0)), Y = s(0)`, `X = s(0), Y = s(s(0))`, and, finally, `X = 0, Y = s(s(s(0)))`. The query "`?- add(X,s(s(0)),Z)`" also returns a meaningful result (all pairs of numbers such that the second is two bigger than the first, i.e., `X = U, Z = s(s(U))` for a fresh variable `U`. But for the query "`?- add(s(0),Y,Z)`" we obtain infinitely many answers.

Disadvantages of representing natural numbers as terms using `s` and `0` are that there are no pre-defined arithmetical operations and that the terms can become very large. Together, this leads to inefficient and hard to read programs. For this reason, Prolog allows the usual syntax for integers and floating point numbers and provides the basic arihtmetical operations in the form of built-in predicates.

An *arithmetical expression* is a term that only contains numbers, variables, and binary infix functions such as `+`, `-`, `*`, `//` (integer division), `**` (power), etc. as well as the unary function `-` (negation). These expressions are normal Prolog terms and are handled the

same way as all other terms, i.e., using unification. If we have a program with the fact “`equal(X,X).`”, posing the query “`?- equal(3,1+2).`” yields the answer `no`. For the query “`?- equal(X,1+2).`” we obtain the solution `X = 1+2`.

While arithmetical expressions are in general treated by normal syntactic unification, there are special built-in predicates that interpret these terms as integer values based on the pre-defined semantics of the functions `+`, `-`, `*`, `//`, `**`, etc.

For the *comparison* of arithmetical expressions Prolog uses the binary infix predicates `<`, `>`, `=<`, `>=`, `=:=`, `=\=`. Here, the last operators represent equality and disequality. A query “`?-` t_1 *op* t_2.” succeeds, if t_1 and t_2 are fully-instantiated (i.e., variable-free) arithmetical expressions at the time of execution and the values obtained by the interpretation of the expressions are in relation according to *op*. Thus, these predicates force an evaluation of their arguments. If t_1 or t_2 are not fully-instantiated arithmetical expression, the program aborts with an error message. The following queries yield the following resutls:

- “`?- 1 < 2.`” or “`?- -2 < -1.`” or “`?- 1*1 < 1+1.`” yield the result `yes`.
- “`?- 2 < 1.`” or “`?- 6//3 < 5-4.`” yield the result `no`.
- “`?- a < 1.`” or “`?- X < 1.`” yield a program error.

The demand that during execution all variables have to be instantiated means that these built-in predicates cannot be used to instantiate variables by unification. A query like “`?- X =:= 2.`” does not lead to the answer `X = 2`, but yields a runtime error. For this reason there is a further pre-defined predicate `is`. A query “`?-` t_1 `is` t_2.” succeeds, if t_2 is a fully-instantiated arithmetical expression during execution and its value z_2 unifies with t_1. If t_2 is not a fully-instantiated arithmetical expression, the query does not fail but yields a runtime error. Thus, the following queries yield the following results:

- “`?- 2 is 1+1.`” or “`?- 2 is 2.`” yield the result `yes`.
- “`?- 1+1 is 2.`” or “`?- 1+1 is 1+1.`” or “`?- X+1 is 1+1.`” yield the result `no`.
- “`?- X is 2.`” or “`?- X is 1+1.`” yield the answer `X = 2`.
- “`?- X is 3+4, Y is X+1.`” yields the answer `X = 7, Y = 8` (as during execution the variable `X` in “`Y is X+1`” is instantiated by 7).
- “`?- X is X.`”, “`?- 2 is X.`”, “`?- X is a.`”, or “`?- Y is X+1, X is 3+4.`” yield a runtime error.

Furthermore, there is a pre-defined predicate `=` for the equality of arbitrary terms. This symbol is treated as if it was defined by the fact “`X = X.`”. It is not restricted to arithmetical expressions. In contrast to the special built-in predicates for arithmetical expressions, there is no evaluation of the pre-defined functions `+`, `-`, `*`, `//`, `**`, etc. The same holds for user-defined predicates. Thus, the following queries yield the following results:

- “`?- a = a.`” or “`?- 2 = 2.`” or “`?- 1+1 = 1+1.`” or “?- X = X.” yield the result `yes`.

- "`?- 2 = 1+1.`" or "`?- 1+1 = 2.`" yield the result `no`.
- "`?- X+1 = 1+1.`" or "`?- 1 = X.`" yield the answer `X = 1`.
- "`?- X = 1+1.`" yields the answer `X = 1+1`.
- "`?- 1+X = Y+1.`" yields the answer `X = 1, Y = 1`.
- "`?- X = 3+4, Y is X+1.`" yields the answer `X = 3+4, Y = 8`.

There is also a built-in predicate `==/2` for syntactic equality, i.e., that is true when both arguments are the same terms. Consequently, we have four notions of equality in Prolog:

- *equality of values* "t_1 `=:=` t_2", where t_1 and t_2 are evaluated and the values are compared.
- *assignment of values* "t_1 `is` t_2", where t_2 is evaluated and its value is unified with t_1.
- *equality of terms by unification* "t_1 `=` t_2", where we do not evaluate but unify t_1 and t_2.
- *syntactic equality of terms* "t_1 `==` t_2", where we check if t_1 and t_2 are the same terms.

The following example shows how the program for addition from the beginning of this section can be written using pre-defined arithmetic expressions. (Of course, we could alternativelyjust use the program "`add(X,Y,Z) :- Z is X+Y.`".)

```
add(X,0,X).
add(X,Y,Z) :- Y > 0, Y1 is Y-1, add(X,Y1,Z1), Z is Z1+1.
```

As expected, the query "`?- add(1,2,X).`" yields the answer `X = 3`. The advantage of this this program is more efficiency and more readability. A disadvantage is that we cannot use different directions of computation anymore. The query "`?- add(X,2,3).`" yields a runtime error as `Z1` in the query "`Z is Z1+1`" is not fully-instantiated during runtime.

The alternative program

```
add(X,0,X).
add(X,Y+1,Z+1) :- add(X,Y,Z).
```

would not work as expected. The query "`?- add(1,2,X).`" yields `no`, because `2` unifies neither with `0` nor with `Y+1`. The query "`?- add(1,0+1,X).`" yields the answer `X = 1+1`.

As further typical examples we show how to write predicates for the factorial function and for the greatest common divisor of two natural numbers.

```
fact(0,1).
fact(X,Y) :- X > 0, X1 is X-1, fact(X1,Y1), Y is X*Y1.

gcd(X,0,X).
gcd(0,X,X).
gcd(X,Y,Z) :- X =< Y, X > 0, Y1 is Y-X, gcd(X,Y1,Z).
gcd(X,Y,Z) :- Y  < X, Y > 0, X1 is X-Y, gcd(X1,Y,Z).
```

The query "`?- fact(3,X).`" yields the answer `X = 6` and the query "`?- gcd(28,36,X).`" yields the answer `X = 4`.

To check the types of terms, there are a number of pre-defined predicates in Prolog:

- `var/1`, which is true if its argument is a variable
- `number/1` which is true if its argument is instantiated by a number (integer or floating point, does not evaluate its argument)
- `atom/1` which is true if its argument is instantiated by a non-numerical constant
- `list/1` which is true if its argument is instantiated by a list (see next section)

2.5 Lists

To represent lists as terms in Prolog, one typically uses a constant for the empty list and a binary functor for insertion of an element at the beginning of the list. If we call these symbols `nil/0` and `cons/2`, we obtain the following logic program for computing the length of a list (pre-defined predicate `length/2`):

```
len(nil,0).
len(cons(X,Xs),Y) :- len(Xs,Y1), Y is Y1+1.
```

The query "`?- len(cons(7,cons(3,nil)), X).`" yields the answer `X = 2`.

If instead of `nil` we use the constant `[]/0` and instead of `cons` the functor `./2`, there is built-in support for more readable notations. As before, we can write the algorithm for computing the length of a list as follows:

```
len([],0).
len(.(X,Xs),Y) :- len(Xs,Y1), Y is Y1+1.
```

But we can also use the following notations:

- `.(`t_1`, `t_2`)` = `[`t_1`|`t_2`]`
- `.(`t_1`, [])` = `[`t_1`]`
- `.(`t_1`, .(`t_2`, .(`t_3`, `t`)))` = `[`t_1`, `t_2`, `t_3`|`t`]`
- `.(`t_1`, .(`t_2`, .(`t_3`, [])))` = `[`t_1`,`t_2`,`t_3`]` = `[`t_1`,`t_2`|[`t_3`|[]]]` = `[`t_1`|[`t_2`,`t_3`|[]]]` etc.

The terms described by these notations are *identical* to the ones using only "`./2`" and "`[]/0`". For example, consider the following queries:

- "`?- [1,2] = [1|[2]].`" or "`?- [1,2] = .(1,[2])`" or "`?- [1,2,3] = [1|[2,3|[]]].`" or "`?- .(1,.(2,[3])) = [1,2,3]`" or "`.(1,2) = [1|2]`" yield `yes`.
- "`?- .(1,X) = [1,2,3]`" yields the answer `X = [2,3]`.
- "`?- [X,[1|X]] = [[2],Y].`" yields the answer `X = [2], Y = [1,2]`.

We observe that "`.`" really is an arbitrary binary function symbol and can, thus, also be used to represent binary trees. For example, the term `.(.(1,2),.(3,4))` represents a binary tree, that could also be written as `[[1|2]|[3|4]]`.

The following example program checks if an object is contained in a list.

```
member(X,[X|Xs]).
member(X,[Y|Ys]) :- member(X,Ys).
```

The query "`?- member(X,[[a,b],1,[]]).`" yields the answer substitutions `X = [a,b]`, `X = 1`, and `X = []`. The query "`?- member(b,X).`" finds all lists, that contain `b`. We obtain the infinitely many solutions `X = [b|Xs]` (alle lists where the first element equals `b`), `X = [Y,b|Xs]` (all lists where the second element equals `b`), etc.

A further popular predicate is `app` which can be used to concatenate lists (pre-defined predicate `append/3`):

```
app([],Ys,Ys).
app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).
```

The query "`?- app([1,2],[3,4,5],Xs).`" yields the answer `Xs = [1,2,3,4,5]`. The query "`?- app(Xs,Ys,[1,2,3]).`" yields the answers `Xs = [], Ys = [1,2,3]`, `Xs = [1], Ys = [2,3]`, `Xs = [1,2], Ys = [3]`, and `Xs = [1,2,3], Ys = []`. The query "`?- app(Xs,[],Zs).`" has again infinitely many solutions.

2.6 Operators

The standard notation for Prolog terms uses a prefix notation where a functor is headed by a list of arguments in parentheses. For example, we write `p(X,f(a))` given the functors `p/2`, `f/1`, and `a/0`.

Instead of this we can also use binary functors in infix notation and unary functors in prefix or postfix notation - and without parentheses. To this end we have declare these functors to be operators. The advantage of these possibilities is a user-friendly syntax with better readability. In this way we can do some kind of "Programming in Natural Language".

For example, the functor `+` is already pre-defined as an operator. Thus, we are allowed write a term as `2+3`. This is converted by Prolog into the term `+(2,3)` (just as `[1,2]` is converted to `.(1,,(2,[]))`). The query "`?- 2+3 = +(2,3).`" therefore yields `yes`.

To define operators, we use so-called *directives* of the form

`:- op(`*precedence*`, `*type*`, `*name(s)*`).`

This means we have a clause with an empty head and the built-in predicate `op/3`. The operators declared in a directive can be used after their declaration. Directives are queries which are executed while the program is loaded. For the operators `+`, `-`, and `*` the following pre-defined directives are executed:

```
:- op(500,yfx,[+,-]).
:- op(400,yfx,*).
```

The last argument of `op` always contains the symbol or the list of symbols that are being declared as operators. The *precedence* is needed to express how strong a functor binds. For example, `*` binds stronger than `+` and this is expressed by `*` having a smaller precedence. (A *smaller* precedence corresponds to a *stronger* binding.)

The *type* determines the order of operators and arguments. Here, `f` represents the operator while `y` and `x` represent the arguments. For infix functors there are the types `xfx`, `yfx`, and `xfy`. For prefix functors there are the types `fx` and `fy`. Finally, for postfix functors there are the types `xf`, and `yf`.

Here, the precedence of `x`–arguments has to be *strictly smaller* than the precedence of the operator `f`. The precedence of `y`–arguments has to be *less or equal* than the precedenz of `f`. The precedence of an argument is the precedence of the leading operator and the precedence of functors and of arguments in parentheses is 0).

For the type `yfx`, arguments with the same precedence as the operator may only occur left of the operator. This means that `1+2+3` has to be read as `(1+2)+3`. Thus, the query "`?- 1+2+3 = (1+2)+3.`" yields the answer `yes` and the query "`?- 1+2+3 = 1+(2+3).`" yields `no`. Such operators are called *left-associative*. This also means that `5-4-3` has to be read as `(5-4)-3`. The query "`?- X is 5-4-3`" therefore yields the answer `X = -2`. Analogously, the type `xfy` declares *right-associative* operators while `xfx` declares operatoren without associativity.

The term `1+2*3+4` has to be read as `(1+(2*3))+4`. The reason is that each operator may only have arguments of equal or smaller precedence. As `*` has smaller precedence than `+`, the two `+`-terms cannot be the arguments of `*`.

Operator may, of course, also be overloaded. Thus, there is additionally the unary operator `-`.

```
:- op(200,fy,-).
```

The expression `-2-3` thus represents `(-2)-3`.[4]

The following example shows how to define one's own operators to implement a simple form of natural language processing. We use the verb "`was`" in infix notation. It should not have associativity, as sentences like "`lone was young was beautiful`" do not make sense. Fruthermore, we use the word "`of`" in infix notation where this should be right-associative. Thus, the term "`secretary of son of john`" represents "`secretary of (son of john)`". To make `of` bind stronger than `was`, the operator `of` should have a lower precedence. Then the term "`laura was secretary of john`" represents "`laura was (secretary of john)`". Finally, we also introduce the word "`the`'. This is a prefix operator without associativity, as "`the secretary the son`" makes no sense. The precedence of "`the`" should be smaller than the precedence of "`of`". Then, the term "`the secretary of the son`" represents "`(the secretary) of (the son)`".

Now we can define the following Prolog program:

```
:- op(300,xfx,was).
:- op(250,xfy,of).
:- op(200,fx,the).
```

[4]Be careful with overloading operators and relying on priorities, though. For example, the term `- - 2` represents `-(-,2)` and not `-(-2)`.

```
laura was the secretary of the head of the department.
```

The fact "`laura was the secretary of the head of the department.`" represents the following fact using the functors `was`, `of`, and `the`:

```
was(laura,of(the(secretary),of(the(head),the(department))))
```

We can now pose the following queries:

```
?-  Who was the secretary of the head of the department.
Who = laura

?- laura was What.
What = the secretary of the head of the department

?- Who was the secretary of the head of What.
Who = laura
What = the department
```

2.7 Cut and Negation-as-Failure

The backtracking behaviour of Prolog is very powerful, but so far we lack the possibility to control it. Thus, we now introduce the *cut* operator `!/0`. Using this special built-in operator, we can forbid backtracking in situations where it is undesirable. The cut operator also allows to define meta predicates such as the *negation-as-failure* operator `\+/1`.

The Cut Operator

Prolog automatically performs backtracking if it reaches a leaf in the SLD tree that represents a finite failure, i.e., a leaf that is not the empty clause $\Box$. While this behaviour is often advantageous, there are many cases where we want to avoid backtracking:

- Backtracking consumes time and also space, as all nodes with all possible choices have to be stored during the proof.
- Backtracking and, thus, exploration of all branches of the SLD tree can lead to undesired non-termination in case that some branches are infinite.
- With a way of controlling backtracking, we can implement negation.

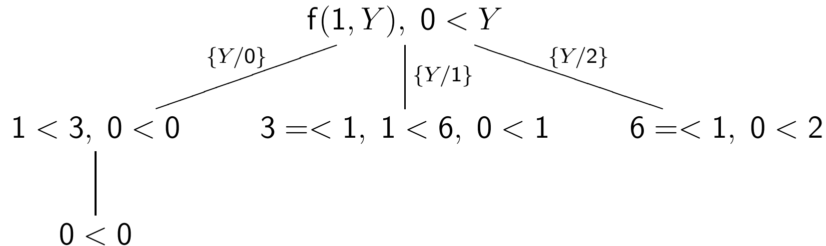
Let us first consider the following simple function f where

$$f(x) = \begin{cases} 0, \text{falls } x < 3 \\ 1, \text{falls } 3 \leq x < 6 \\ 2, \text{falls } 6 \leq x \end{cases}$$

The following Prolog program computes this function f:

```
f(X,0) :- X<3.
f(X,1) :- 3=<X, X<6.
f(X,2) :- 6=<X.
```

We now pose the query "`?- f(1,Y), 0<Y.`". This leads to the following SLD tree.



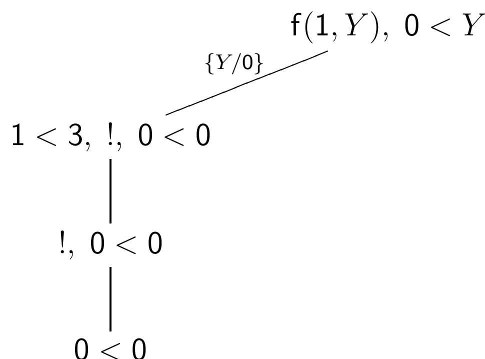
The query does not succeed and the result is `no`. We recall that SLD resolution tries the clauses from top to bottom. In our SLD tree, this means that Prolog first visits the leftmost path of the tree. Here, the the proof of the first goal `X<3` succeds, because `X` is instantiated to `1`. But the second goal `0<Y` fails as `Y` is instantiated to `0`. After this finite failure, Prolog performs backtracking and explores the two other paths corresponding to the two other program clauses.

As a human we realize that the the conditions "`X<3`", "`3=<X, X<6`", and "`6=<X`" of the three `f`–clauses mutually exclude each other. As soon as the proof of one of these three conditions succeeds, we can completely disregard the other `f`–clauses. As for our query already the condition `X<3` of the first `f`–clause can be proven, we do not need to perform backtracking to consider the two other `f`–clauses – it is already clear that their conditions cannot hold. For this reason, we would like to *cut* the middle and the right path of the SLD tree.

To perform this cut, we use the so-called *cut* operator represented by the functor "`!/0`" This constant may occur on the right-hand side of rules and in queries. Its proof always succeeds without instantiating any variables, but its effect is to cut alternative paths in the SLD tree. To illustrate this behaviour, we modify our program as follows:

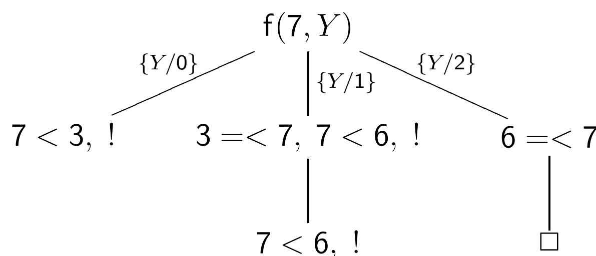
```
f(X,0) :- X<3, !.
f(X,1) :- 3=<X, X<6, !.
f(X,2) :- 6=<X.
```

The effect of the cut in the first `f`–clause is to disallow backtracking after the first `f`–clause has been used for a query `f(...)` and the goal `X<3` has successfully been proven. This means that Prolog is not allowed to backtrack anymore in order to try alternative proofs for `X<3` or for our initial query `f(...)`. Thus, proof attempts using the third and the second `f`–clause are cut away. The resulting SLD tree for our query "f(1,Y), 0¡Y." now looks as follows:



Cuts like the one used in the program above are so-called *green* cuts. They only affect efficiency but not the results of the program. If one drops green cuts, we still obtain the identical solutions.

The cuts in the preceding program have the effect that Prolog never tries to apply further `f`–clauses after a successful proof with the condition of the first or second `f`–clause. This allows a further improvement of the program to increase efficiency even more. To understand the idea of this improvement, consider the SLD–tree for the query "`?- f(7,Y).`".



After the goal `X<3` in the left-most path has failed as `X` is instantiated by `7`, in the middle path obtained by using the second `f`–clause we have to prove the corresponding negated goal `3=<X`. But this turns out to be unnecessary, because whenever the proof of the goal `X<3` fails, the proof of the goal `3=<X` must succeed. Similarly, the proof of the goal `X<6` fails when `X` is instantiated by `7` in the middle path. But then it is unnecessary to try to prove the corresponding negated goal `6<=X` in the right-most path, because the proof is known to succeed. When using cut operators, we can simplify the original program to the following three simple clauses:

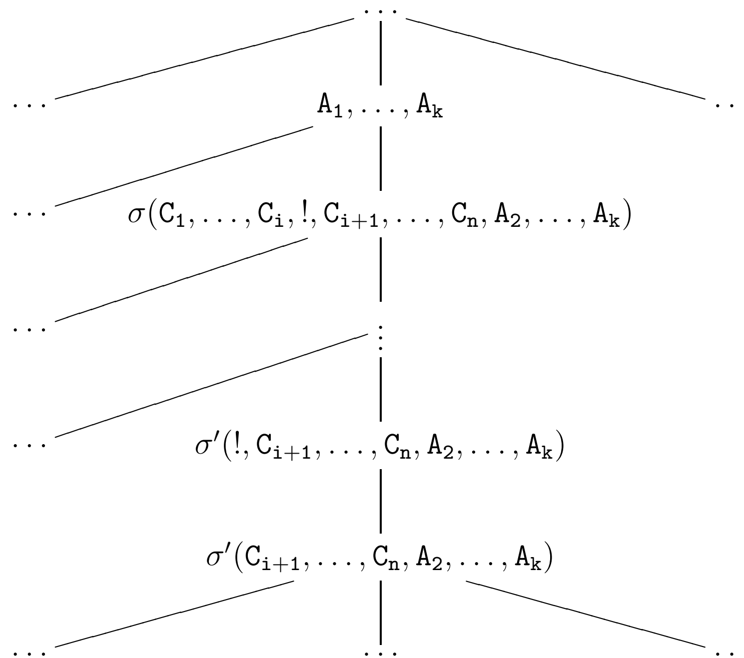
```
f(X,0) :- X<3, !.
f(X,1) :- X<6, !.
f(X,2).
```

These cuts are now so-called *red* cuts, i.e., they affect not only the efficiency of the program but also its results. If we remove these cuts, we obtain different solutions. For example, the query "`?- f(1,Y)`" would not only have the answer `Y = 0` but also the answers `Y = 1` and `Y = 2`.

We also have to consider that when introducing cuts we usually have a certain use of the predicate (with input and output positions) in mind. In other words, we have certain types

of queries that we think will use the predicate. If we do not consider the second position as an output position (by posing a query with an instantiation of the second argument), we observe undesired effects. For example, the query `?- f(0,2)` yields the result `yes`, although the desired function should yield $f(0) = 0$. Likewise, the query "`?- p(X).`" for the program consisting of the clauses "`p(0) :- !.`" and "`p(1) :- !.`" yields just the answer `X = 0`, although the answer `X = 1` should have been possible, too.

We now explain precisely what executing a cut means. If a query "`?-` $A_1,\ldots,A_k$" is resolved with a program clause "`B :-` $C_1,\ldots,C_i$`, !,` $C_{i+1},\ldots,C_n$" and later we have proved the corresponding instantiated subgoals $C_1, \ldots, C_i$, we obtain the following SLD tree where the effect of the cut is already visible:



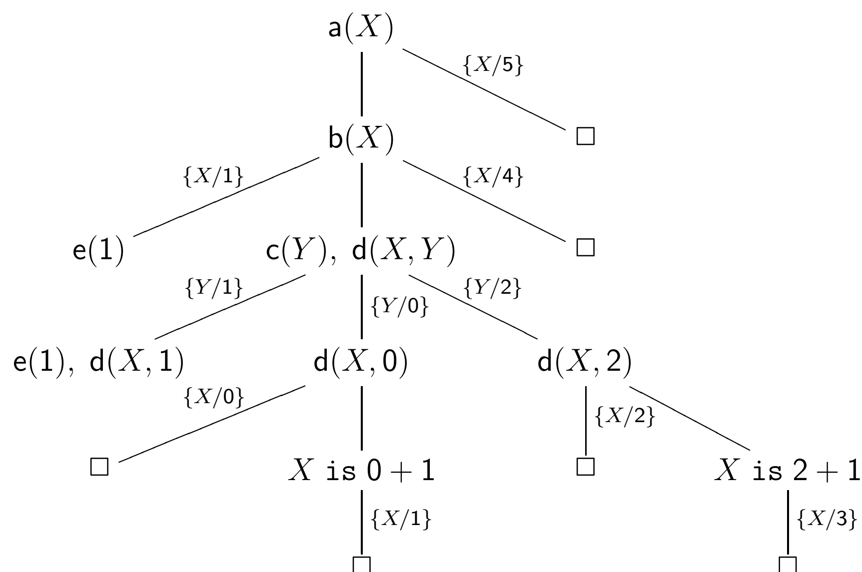
Thus, the cut means that for all nodes between and including the node marked by "$A_1,\ldots,A_k$" and the node marked by "$\sigma'(!,C_{i+1},\ldots,C_n,A_2,\ldots,A_k)$" we do not consider any alternatives on the right-hand side. In contrast, for all nodes above and below these nodes, alternatives are considered. If the proof attempt fails before reaching a cut, i.e., if the instantiated subgoals $C_1, \ldots, C_i$ cannot be proven, Prolog tries all alternatives.

You can see this in the following example:

```
a(X) :- b(X).
a(5).

b(1) :- e(1).
b(X) :- c(Y), d(X,Y).
b(4).

c(1) :- e(1).
c(0).
c(2).
```

```
d(X,X).
d(X,Y) :- X is Y+1.

e(0).
```

For the query "`?- a(X).`" we obtain the following SLD tree:



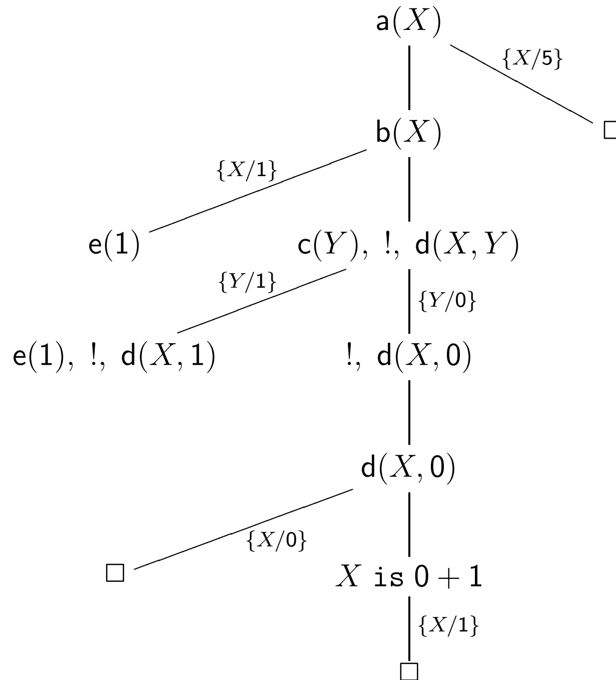
When asking for all solutions, we thus obtain the following answers: `X = 0`, `X = 1`, `X = 2`, `X = 3`, `X = 4`, and `X = 5`.

We now change the second `b`–clause by introducing a cut:

```
b(X) :- c(Y), !, d(X,Y).
```

The effect is that for `a` and `d` we continue to consider alternatives. But we do not consider

alternative for b and c after we have reached the cut. We obtain the following SLD tree:



When asking for all solutions, we obtain the following answers: `X = 0`, `X = 1`, and `X = 5`.

Now, as an example for a real-world use of the cut we consider our greatest common divisor program from Section 2.4:

```
gcd(X,0,X).
gcd(0,X,X).
gcd(X,Y,Z) :- X =< Y, X > 0, Y1 is Y-X, gcd(X,Y1,Z).
gcd(X,Y,Z) :- Y  < X, Y > 0, X1 is X-Y, gcd(X1,Y,Z).
```

If one of the first two clauses unifies with the query, we should not perform further proof attempts with the other gcd–clauses. Thus, we introduce a cut for each of these clauses. This ensure that the two lower clauses are only used if X and Y are both greater than 0 (assuming we are only interested in queries with natural numbers). Now, we can remove the literals `X > 0` and `Y > 0` from the two gcd–rules. Finally, we introduce a cut in the third clause just after the literal `X =< Y`. This ensures that we only reach the last clause if we have `Y < X`. Thus, we can drop that literal from the last clause:

```
gcd(X,0,X) :- !.
gcd(0,X,X) :- !.
gcd(X,Y,Z) :- X =< Y, !, Y1 is Y-X, gcd(X,Y1,Z).
gcd(X,Y,Z) :- X1 is X-Y, gcd(X1,Y,Z).
```

If we are concerned about negative numbers (and the resulting infinite SLD tree for e.g. the query “`gcd(1,-1,Z)`”), we can add two clauses before the first clause two check for negative values:

```
gcd(X,Y,Z) :- X < 0, !, X1 is -X, gcd(X1,Y,Z).
gcd(X,Y,Z) :- Y < 0, !, Y1 is -Y, gcd(X,Y1,Z).
```

Finally, we present a natural example for the use of the cut when programming with lists. The predicate `remove(X,Xs,Ys)` should be provable, if the list `Ys` can be constructed from the list `Xs` by removing all occurrences of `X` from `Xs`.

```
remove(X,[],[]).
remove(X,[X|Xs],Ys) :- !,remove(X,Xs,Ys).
remove(X,[Y|Xs],[Y|Ys]) :- remove(X,Xs,Ys).
```

The query "`?- remove(1,[0,1,2,1],Ys).`" yields the only answer `Ys = [0,2]`. Without the cut there are also the answers `Ys = [0,2,1]`, `Ys = [0,1,2]`, and `Ys = [0,1,2,1]`.

Meta Variables and Negation-as-Failure

Prolog allows the use of *meta variables*. These are variables that may contain arbitrary Prolog terms. In particular, it may contain complete goals. Actually, there is no special treatment for these variables. They are instantiated by unification just like any other variables. We call predicates that work on goals *meta predicates*.

As a simple example consider the following program:

```
p(a).
a.
```

Here, `a/0` is a functor representing a 0-ary predicate. Thus, the predicate `p/1` takes as an argument a goal. If we pose the query "`?- p(X), X.`", the variable `X` is a meta variable that will be instantiated by the goal "`a`". This query succeeds with the unique answer `X = a`. Meta variables always have to be instantiated before they can be used for resolution. The query "`?- p(X), X, Y.`" thus leads to a program error.

A further example for the use of meta variables is the following program:

```
or(X,Y) :- X.
or(X,Y) :- Y.
```

This predicate is also available in Prolog as the pre-defined operator "`;`'. Here, "`;`" is an infix operator declared by the directive `:- op(1100,xfy,;)`. The query "`?- X = 4 ; X = 5.`" returns the answers `X = 4` and `X = 5`.

The buit-in operator "`,`" for conjunction has precedence 1000, i.e., it binds stronger than "`;`". Thus, for the query "`?- p(X,Y).`" given the program with the clause "`p(X,Y) :- X = 1, Y = 1; X = 2, Y = 2.`" the answers are `X = 1, Y = 1` and `X = 2, Y = 2`.

By using the cut we can in particular program meta predicates that negate other existing predicates or that combine such predicates in a different way. The following program implements an operation similar to the popular if-statement "`if A then B else C`".

```
if(A,B,C) :- A, !, B.
if(A,B,C) :- C.
```

The cut is needed in this situation to avoid further proofs when `A` is proved and `B` fails. Otherwise, `if(A,B,C)` would be provable whenever `C` is. In Prolog, such a predicate is pre-defined. Instead of `if(A,B,C)` we can also write "`A -> B ; C`".

A further important predicate is *negation*. Up to now, for a given logic program we can only prove positive statements, i.e., existentially quantified statements of the form $A_1 \wedge \ldots \wedge A_k$. Our goal is to also be able to prove goals containing negated literals $\neg A$. When implementing negation in Prolog, a number of assumptions are made:

- All true statements about the world can be derived from the logic program (*Closed World Assumption*). Thus, if a statement A cannot be proven, then it does not hold and, consequently, the statement $\neg A$ holds.

- If a statement cannot be derived from the logic program, we detect this in finite time, i.e., by a finite tree with finite failures at the leaves.

In this way we interpret negation as a finite failure (*negation-as-failure*). To prove $\neg A$, we instead try to prove A. If the SLD tree for A is finite and there are no successes, we have proven $\neg A$. Note that the variables in $\neg A$ are universally quantified. We can implement negation-as-failure using the cut operator:

```
not(A) :- A, !, fail.
not(A).
```

Here, `fail/0` is a pre-defined predicate that always fails. It can for example be defined by the two clauses "`fail :- p(a).`" and "`p(b).`" for a fresh predicate `p/1`. The cut is needed in this situation as otherwise, `not(A)` could always be proven by using the fact "`not(A)`". The predicate `not/1` is pre-defined in Prolog and can also be written using the prefix operator `\+` .

Note that `not/1` is only a special case of our `if/3` predicate. Thus, we can also define `not` in the following way:

```
not(A) :- if(A,fail,true).
```

Here, `true/0` is a predefined predicate that always succeeds. It can be defined by the fact "`true.`".

As an example for the use of negation-as-failure, consider the definition of inqueality:

```
X \= Y :- not(X = Y).
```

As expected, the query "`?- 1 \= 2`" yields the result `yes` while the query "`?- 1 \= 1`" yields `no`. The query "`?-X = 2, 1 \= X`" yields `yes`, but "`?- 1 \= X, X = 2`" yields `no`. The reason is that negation turns the existential quantifier into a universal quantifier. The query `1\= X` thus corresponds to the question if *all* `X` are different from `1`. This is, of course, not the case. In the previous query, `X` was already instantiated to `2` and, thus, the query could be proven.

Negation in Prolog does not necessarily correspond to the intuitive meaning of negation. We cannot detect every failure as infinite failing SLD trees are not recognized as failing. The problem is that it is not decidable whether the empty clause can be derived using SLD resolution. To see this, consider the following program:

```
even(0).
even(X) :- X1 is X-2, even(X1).
```

Clearly, for this program, the query "?- even(1)" cannot be proven. But the proof tree is infinite. Thus, neither the query "?- even(1)." nor the query "?- not(even(1))." terminate.

In the version

```
even(0).
even(X) :- X >= 2, X1 is X-2, even(X1).
```

we have that ?- even(t) terminates for every number. But as the query "?- even(-2)." is not provable, we can now prove "?- not(even(-2)).". The reason here is that the Closed World Assumption does not hold as we have not modelled the complete application area by our logic program.

An alternative version that works correctly for all integers is the following:

```
even(0) :- !.
even(X) :- X > 0, !, X1 is X-1, not(even(X1)).
even(X) :- X1 is X+1, not(even(X1)).
```

2.8 Input/Output

Up to now the only way to pass *inputs* to a program was the use of queries. The only way to obtain *output* was from the answer substitution returned and the results "yes" and "no". But Prolog also offers built-in predicates that have input and output as side effects. These predicates leave the pure context of logic programming, but are immensely useful when writing real software.

Using the built-in predicate write/1 we can write the representation of a term to the currently selected output stream. This is usually the console of the user. Thus, the query "?- write(t)." succeeds for every term t and outputs t as a side-effect. For example, for the query "?- X is 2+3, write(X)." as as side-effect the output 5 is printed and the answer is X = 5. For the query "?- write('I am a constant!').", the constant "I am a constant!" is printed as a side-effect.

For the following program

```
mult(X,Y) :- Result is X*Y, write(X*Y), write(' = '), write(Result).
```

we obtain the following result:

```
?- mult(3,4).
3*4 = 12.
```

Here, we have to keep in mind that backtracking cannot undo the side-effects of a write literal. For the program

```
q(a).
q(b).
p :- q(X), write(X), X = b.
```

we, thus, obtain:

```
?- p.
ab
```

A further pre-defined predicate is `nl/0` (short for **n**ew**l**ine), that results in a linefeed in the current output stream. Consider e.g. the behaviour of the following query:

```
?- write(a),nl,write(b),nl,write(c).
a
b
c
```

Of course, there are many other predicates for output and formatting output available in Prolog.

For input, there is a pre-defined predicate `read/1`. Here, the query "`?- read(t)`" reads a term `s` from the current input streams and tries to unify `t` and `s`. If the terms `t` and `s` do not unify, the goal `read(t)` fails. To mark the end of the term `s`, this has to end with a ".".

Let us consider the following program:

```
sqr(X,Y) :- Y is X*X.

sqr :- nl,
       write('Please enter a number or "stop": '),
       read(X),
       proc(X).

proc(stop) :- !.
proc(X) :- sqr(X,Y),
           write('The square of '),
           write(X),
           write(' is '),
           write(Y),
           sqr.
```

To execute the program, we pose the query "`?- sqr.`". One possible run of the program could then be:

```
?- sqr.

Please enter a number or "stop": 3.
The square of 3 is 9
Please enter a number or "stop": -4.
The square of -4 is 16
Please enter a number or "stop": stop.

yes
```

It is, of course, also possible to use input and output with files. To this end, we have to set the current input and output stream to the files that we want to read from and write to, respectively. This can be achieved by using the predicates `see/1` and `tell/1`. The query "`?- see(t).`" sets the input stream to a file with the name `t`. Analogously, the query "`?- tell(t).`" sets the output stream. The predicates `seen/0` and `told/0` close the input and output stream, respectively. The current streams are then set back to `user`.

We can now modify the preceding program as follows:

```
sqr(X,Y) :- Y is X*X.

start  :- nl,
          write('Please enter the name of the input file: '),
          read(Inputfile),
          write('Please enter the name of the output file: '),
          read(Outputfile),
          see(Inputfile),
          tell(Outputfile),
          sqr,
          seen,
          told.

sqr :- read(X),
       proc(X).

proc(end_of_file) :- !.
proc(X) :- sqr(X,Y),
           write('The square of '),
           write(X),
           write(' is '),
           write(Y),
           nl,
           sqr.
```

The program now reads terms from the input file. When the end of the file is reached, `read(X)` returns the answer `X = end_of_file`. If the file `input` contains the following content

```
3.  -4.
```

the following program run is possible:

```
?- start.

Please enter the name of the input file: input.
Please enter the name of the output file: output.

yes
```

Then, the file `output` contains:

```
The square of 3 is 9
The square of -4 is 16
```

2.9 Constraint Programming

The built-in backtracking of Prolog is a powerful tool for solving search problems. Consider for example the famous "SEND + MORE = MONEY" puzzle. In this puzzle we are looking for instantiation of the variables S, E, N, D, M, O, R, and Y with digits from $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ such that all variables get a unique value, the numbers SEND, MORE, and MONEY do not start with a zero and the equation SEND + MORE = MONEY holds.

A natural approach to solve this problem in Prolog is to generate all possible instantiations of the variables and test if these fulfill the requirements of a solution. The following program uses this approach:

```
smm(X) :-
  X = [S,E,N,D,M,O,R,Y],
  Digits = [0,1,2,3,4,5,6,7,8,9],
  assign(X,Digits),
  M > 0,
  S > 0,
            1000*S + 100*E + 10*N + D +
            1000*M + 100*O + 10*R + E =:=
  10000*M + 1000*O + 100*N + 10*E + Y.

select(X,[X|Xs],Xs).
select(X,[Y|Xs],[Y|Ys]) :- select(X,Xs,Ys).

assign([],L).
assign([D|Ds], L) :- select(D, L, M), assign(Ds, M).
```

Given the query "`smm(X)`", we obtain the only answer `X = [9,5,6,7,1,0,8,2]` after some seconds. The reason why it takes so long to compute the answer is that there are $\frac{10!}{2!}$ possible instantiations of the variables in `X` to consider.

To obtain a more efficient implementation, we would like to already use the conditions on the variables while looking for possible instantiations. Unfortunately, we can only perform tests on a variable once it has been instantiated. To remedy this situation, we introduce the concept of *constraint programming*.

Instead of using the built-in arithmetic operators `>`, `>=`, `=:=`, `=\=`, etc. we use the built-in constraint operators `#>`, `#>=`, `#=`, `#\=`, etc.

Now, the big difference is that we can introduce the constraints first and start the search only when the problem is fully constrained. Additionally, there is built-in support for many routine tasks in constraint programming:

- fd_domain($[\mathtt{X_1}, \ldots, \mathtt{X_n}]$, Min, Max):
  With the help of this predicate we can fix the possible values of variables (i.e., the domain) to $\mathtt{Min} \leq \mathtt{X_i} \leq \mathtt{Max}$.

- fd_all_different($[\mathtt{X_1}, \ldots, \mathtt{X_n}]$):
  This predicate specifies that the vairables $\mathtt{X_1}, \ldots, \mathtt{X_n}$ have to be instantiated by unique values. In other words, for all $1 \leq i, j \leq n$ with $i \neq j$ we have $\mathtt{X_i}$ #\= $\mathtt{X_j}$.

- fd_labeling($[\mathtt{X_1}, \ldots, \mathtt{X_n}]$):
  When all constraints are set up, the search is started using this predicate. The argument is the list of constraint variables $\mathtt{X_1}, \ldots, \mathtt{X_n}$ for which we want to obtain concrete values. Typically, here one uses a list containing all constraint variables.

We obtain the following constraint logic program:

```
smm(X) :-
  X = [S,E,N,D,M,O,R,Y],
  fd_domain(X,0,9),
  fd_all_different(X),
  M #> 0,
  S #> 0,
            1000*S + 100*E + 10*N + D +
            1000*M + 100*O + 10*R + E #=
  10000*M + 1000*O + 100*N + 10*E + Y,
  fd_labeling(X).
```

Given the query "smm(X)", thanks to the constraints we now obtain the only answer X = [9,5,6,7,1,0,8,2] almost instantaneously.

In this example, we used constraints over finite integer domains (fd domain). There are many other domains that are supported by at least some Prolog implementations, most notably the domains of rational or real numbers.

Chapter 3

Functional Programming

The following example demonstrates one main difference between imperative and functional programming languages. We use Java as the imperative language and Haskell as the functional language. To illustrate these programming paradigms, we consider the algorithm for computing the length of a list. The input of the algorithm is a list, e.g. `[15,70,36]`. The output is the length of this list (here: 3). An imperative algorithm that solves this task can easily be given:

```
class Element {

  Data    value;
  Element next;
}
```

```
class List {
  Element head;

  static int len (List l) {
    int n = 0;

    while (l.head != null) {
      l.head = l.head.next;
      n = n + 1;
    }
    return n;
  }
}
```

We observe the following about this program:

- The program consists of single statements that are processed one after the other. As mentioned in the introduction, there are also control structure (conditional execution, loops, etc.).
- The execution of a statement changes the value of variables in the memory. Thus, almost any execution can have side effects. For example, during the execution of `len` both the value of `n` and of `l` are changed. While the former is only of interest inside the method `len`, the latter also has consequences outside of `len`. During the execution of `len(l)` the value of the object that `l` references is changed. Thus we do not only obtain an `int` value as the result, but as a *side effect* the list `l` is modified. After the execution of `len(l)`, we have `l.head = null`), i.e., the list has been emptied while computing its length.

- The programmer has to think about how to implement and manage non-primitive data types (such as lists, trees etc.) in memory. For example, we can assume that the side effect in `len` is not intended by the programmer. To avoid this undesired behavior, the programmer has to anticipate desired and undesired side effects. Such side effects and the explicite management of memory in imperative programs often lead to errors that are hard to localize.

After having illustrated the concepts of imperative programming, we now consider declarative programming again. As mentioned before, in contrast to the detailed description on how to perform the computation that is common to imperative programs, the idea behind declarative programming is to specify what the program should compute and leave the details of how to compute it to the compiler or the interpreter. Our goal is gain to compute the length of a list. The following description clarifies what we mean by the length `len` of a list `l`:

(A) If the list `l` is empty, then `len(l) = 0`.

(B) If the list `l` is not empty and "`xs`" is the list `l` without its first element, then `len(l) = 1 + len(xs)`.

In the previous chapter, we already learned how to use relations to represent such a function. Here, we could use a binary predicate `len/2` that is provable whenever the second argument is the length of the list given as the first argument. When posing a query with a list as the first argument and a variable as the second argument, Prolog instantiates the variable by the length of a list. The following program uses the built-in support for integers to implement `len/2`:

```
len([],0).
len([X|Xs],M) :- len(Xs,N), M is N+1.
```

In the following we write `x:xs` for the list that is the result of inserting the element (or the value) `x` at the beginning of the list `xs`. Thus, we have `15:[70,36] = [15,70,36]`. Every non-empty list can be represented as `x:xs`, where `x` is the first element of the list and `xs` is the remaining list without its first element. If we denote the empty list by `[]`, we have `15:70:36:[] = [15,70,36]` (where insertion using "`:`" is right-associative, i.e., we have `15:70:36:[] = 15:(70:(36:[]))`.

Now we can directly translate the above specification of the function `len` into a functional program. The implementation in the functional programming language HASKELL would be as follows:

```
len :: [data] -> Int
len []     = 0
len (x:xs) = 1 + len xs
```

Just like a logic programming is a set of clauses, a functional programming is a set of declarations. The three declarations for `len` bind the variable `len` to a value. Here, this value is the function that computes the length of a list. The line `len :: [data] -> Int`

is a *type declaration*, stating that the function `len` is bound to expects a list as input and returns an integer as output. Here, `data` is a *type variable* representing the type of the elements of the list. The data structure for lists is pre-defined in in HASKELL. We will later see how we can define such structures on our own.

The next two lines contain the defining equations for `len`. The first equation states what the result of the function `len` is, if case `len` is applied to the empty list `[]`. (In HASKELL, instead of writing `f(23)` for the application of the function `f` to the value `23`, we just write `f 23`. Thus, the space between `f` and `23` is the application operator and it is left associative. The second equation can be applied in case the argument of `len` is a non-empty list. Then the argument has the form `x:xs`. In this case, we continue by computing the value of the expression `1 + len xs` where the variable `xs` is bound to the rest of the list.

We observe that `len` is defined *recursively*, i.e., for computing the value of `len(x:xs)` wie have to compute the value of `len` applied to another argument (here: `xs`).

The execution of a functional program consists of evaluating expressions using the defining equations of the function. In our example we just define the function `len`. To illustrate the workings of that aglorithm, let us consider the evaluation of the expression `len [15,70,36]`. When executing the algorithm, we first check if the argument is the empty list, i.e., if `[]` can be instantiated to be equal to the argument `[15, 70, 36]`. This process is called *pattern matching*. In general, a term s matches a term t if, and only if, there is a substitution σ such that $s\sigma = t$.[1]

In this example, there is no substitution σ such that $[]\sigma = [15, 70, 36]$. Thus, we try to apply the second equation. This is possible in our example by instantiating the first list element `x` by the value `15` and the rest of the list `xs` by the list `[70,36]`. To evaluate `len [15,70,36]`, we now have to determine the value of the expression `1 + len [70,36]`. As the new argument `[70,36]` is again a non-empty list with `x` instantiated by `70` and `xs` by `[36]`, we obtain another application of the function `len` with the argument `[36]`. Iterating this once more, we obtain an application of `len` to the argument `[]`. Here, we can finally use the first defining equation and obtain the value `0`. In total, we now obtain the expression `1 + 1 + 1 + 0` which evaluates to the value `3` using the pre-defined function for addition.

3.1 Basic Language Constructs of HASKELL

In this section we introduce the basic language constructs of HASKELL (declarations, expressions, patterns, and types).

3.1.1 Declarations

A program in HASKELL is a sequence of declarations. The order of declarations only matters if more than one defining equations is applicable to one expression, i.e., if the definitions overlap for some function. In this case, equations are tried in the order that they appear

[1]Thus, in functional programm, only the variables of the left-hand side of the defining equation are instantiated. In fact, expressions that are to be evaluated cannot contain variables. In contrast, in logic programming, both clauses and queries may contain variables and both the variables in the head of the clause and the variables in the query are instantiated.

in the sequence, i.e., from top to bottom in the source code. The declarations have to be left-aligned, i.e., they start in the same column. We will later learn the reason for this requirement when considering local declarations that occur on indented positions.

In the simplest case, a *declaration* is a description of a function. Functions are bound to a variable and are characterized by their domain, their range, and a mapping from values of the domain to values of the range. The domain and the range are determined by a *type declaration* and the mapping is determined by the *function declarations*. The syntax of declarations can be described by the following context-free grammar.

$$\underline{\mathsf{decl}} \rightarrow \underline{\mathsf{typedecl}} \mid \underline{\mathsf{fundecl}}$$

In the following we mark nonterminal symbols by underlining. Furthermore, we start with a subset of the syntax of HASKELL and extend the grammar rules successively.[2]

There are two kinds of texts that are considered to be *comments* by HASKELL: all texts enclosed between `{-` and `-}` as well as all text from `--` and the end of the line.

Type Declarations

In HASKELL functions are defined by binding a variable to a function. For example, we might use the variable named `square` for the function that computes the square of a given number. A declaration binds variable (like `square`) to a value (like the function that computes the squares of numbers). Now, we can give the following type declaration for `square`:

```
square ::  Int -> Int
```

The first `Int` describes the domain and the second `Int` describes the range of `square`. The type `Int` is pre-defined in Haskell. The declaration $\underline{\mathsf{var}}$ `::` $\underline{\mathsf{type}}$ means that the the variable $\underline{\mathsf{var}}$ has the type $\underline{\mathsf{type}}$. By using the *type constructor* "`->`" we can define function types. In our example, `Int -> Int` is the type of functions that map integers to integers. As another example, `[Int]` describes the type of lists of integers. In general, for each type `a` there is a type `[a]` of lists with elements of type `a`.

We obtain the following grammar rule for the syntax of type declarations. Here, a type declaration determines the type of one or more variables.

$$\underline{\mathsf{typedecl}} \rightarrow \underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n :: \underline{\mathsf{type}}, \text{ where} n \geq 1$$

It is not necessary to specify type declarations. If they are missing, the interpreter or compile automatically computes them. Type declarations are nevertheless very useful for understanding a program and should therefore usually be given explicitly. These declarations are then checked by the interpretero or compile.

Variable names $\underline{\mathsf{var}}$ are arbitrary sequences of letters and digits (strings) starting with a small letter (e.g. `square`).

[2] We will not consider all possbile HASKELL programs. The complete grammar is available in the Haskell-Report available from: `http://haskell.org/haskellwiki/Language_and_library_specification`

Function Declarations

After the type declaration we find the defining equations, i.e., the mapping from domain values to range values. For example. the function declaration for `square` could be written as follows:

```
square x = x * x
```

The left-hand side of a defining equation consists of the name of the function (or more precisely the name of the variable the function is bound to) and a description of its arguments. The right-hand side defines the results of the function. The types of the arguments and of the result must correspond to the type of the function as declared in the type declaration. Here, `square` may only get arguments of type `Int` and it must also return expression of type `int`.

Basic arithmetic operations such as `+`, `*`, `-`, etc. as well as comparison operations such as `==` (for equality), `>=`, etc. are pre-defined in HASKELL. Likewise, the data structure `Bool` with the values `True` and `False` and the functions `not`, `&&` , and `||` are pre-defined. Heavily used pre-defined function and data structures are typically defined in a library. The function above are defined in a standrd library (the so-called "Prelude"), which is loaded whenever HASKELL starts. In general, function declarations are constructed as follows:

$$\begin{array}{lcl} \underline{\text{fundecl}} & \rightarrow & \underline{\text{funlhs}}\ \underline{\text{rhs}} \\ \underline{\text{funlhs}} & \rightarrow & \underline{\text{var}}\ \underline{\text{pat}} \\ \underline{\text{rhs}} & \rightarrow & = \underline{\text{exp}} \end{array}$$

Here, var represents the name of the variable the function is bound to (like `square`) and pat represents the argument on the left-hand side of the defining equation (like e.g. `x`). How such arguments may look like in the general case will be discussed later. The right-hand side of a defining equation is an arbitrary expression exp (like e.g. `x * x`).

Execution of a Functional Program

The execution of a functional program consists of the evaluation of expressions. This happens in a fashion similar to the query prompt in Prolog: The user provides an expression and the computer evaluates this expression.[3] More precisely, the user sees a prompt > and enters an expression (e.g. `42` or `6 * 7` or `len [15,70,36]`). Then the result is displayed (`42` in the first two cases and `3` in the last). If one enters `42`, the result will be the value `42`. If one enters `6 * 7`, the result is also the value `42` as the operation `*` is pre-defined. User-defined functions are used in the same way for evaluating expressions. When entering the expression `square 11`, we obtain the result `121`. The same result is obtained for the input expression `square (12 - 1)`. The binding priority of function application is highest, i.e., for the expression `square 12 - 1` we obtain the value `143`.

An expression is evaluated by *term rewriting* in two steps:

(1) The computer looks for a subexpression that corresponds to the left-hand side of a defining equation. Here, the variables of the right-hand side have to be replaced by adequate expressions. We call such a subexpression a *redex* (short for "reducible expression").

[3]In `hugs` a program is loaded from `file.hs` using the command `:l file(.hs)`

(2) The redex is replaced through the right-hand side of the defining equation. Here, the variables in the right-hand side have to be instantiated in the same way as in (1).

The evaluation steps are repeated until no more replacements are possible.

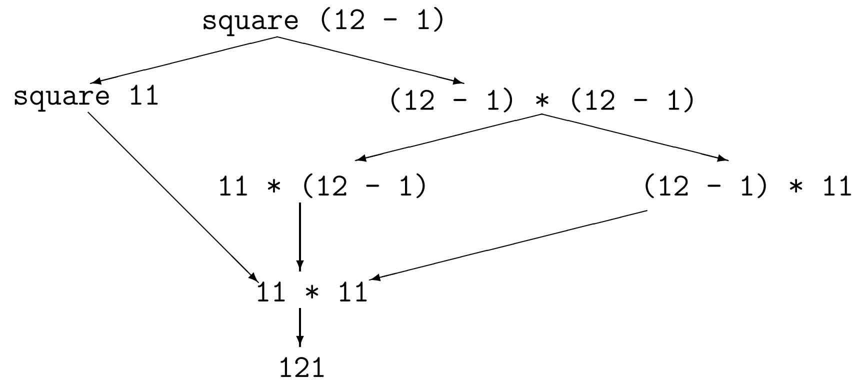


Figure 3.1: Evaluation of an Expression

In Figure 3.1 we see all possibilities for evaluating the expression `square (12 - 1)`. Each path through the diagram corresponds to a possible sequence of evaluation steps. An *evaluation strategy* is an algorithm for selecting the next redex.

In particular, we differentiate between *strict* and *non-strict* evaluation. When using strict evaluation, we always choose the redex that occurs as left and as deep in the expression as possbile. This corresponds to the leftmost path through the diagram in Figure 3.1. This strategy is called the *leftmost innermost* or *call-by-value* strategy and the resulting form of evaluation is often called *eager evaluation*.

When using non-strict evaluation, we choose the redex that occurs as left and as shallow in the expression as possible. The arguments of functions are now in general unevaluated expressions. This corresponds to the middle path through the diagram in Figure 3.1. This strategy is called *leftmost outermost* or *call-by-name* .

Both stratgegies have advantages and disadvantages. When using non-strict evaluation, we only evaluate subexpressions whose value contributes to the final result. This is not the case when using strict evaluation. On the other hand, the non-strict strategy often has to evaluate the same value more than once, although this is not required when using a strict strategy. To see this, consider for example the subexpression `12 - 1`.

HASKELL uses the principle of so-called *lazy evaluation* that tries to combin the advantages of both strategies. Here, we use non-strict evaluation, but duplicated subexpressions are not evaluated more than once, if they result from the same original expression. In the preceding example, the subexpression $\mathtt{12} - \mathtt{1}$ would be *shared* e.g. by using two pointers that reference the same memory cell. Thus, the subexpression is only evaluated once. Identical subexpressions that occur by chance are not (necessarily) recognized and are, therefore, potentially evaluated more than once.

When comparing the evaluation strategies, we obtain the following important result: Whenever evaluation with some evaluation strategy terminates, then also non-strict evaluation terminates (while strict evaluation might well fail to terminate). Furthermore, for all

strategies, if the computation stops, then the result is always the same, i.e., it is independent of the strategy used. Thus, strategies only influence the termination behaviour, but not the result. For example, let us consider the following functions:

```
three :: Int -> Int          non_term :: Int -> Int
three x = 3                  non_term x = non_term (x+1)
```

The evaluation of the function `non_term` does not terminate for an argument. The strict evaluation of the expression `three (non_term 0)` would consequently fail to terminate, too. In HASKELL however, this expression would be evaluated to the result `3`. Further advantages of the non-strict strategy will be discussed in Section 3.3.

Conditional Defining Equations

We would, of course, also like to use functions that take more than one argument as well as conditional defining equations. Too see this, let us consider the function `maxi` with the following type declaration.

```
maxi ::  (Int, Int) -> Int
```

Here, `(Int, Int)` denotes the cartesian product iof the types `Int` and `Int`, i.e., it corresponds to the mathematical notation `Int` × `Int`. The type `(Int, Int) -> Int` is consequently the type of functions mapping pairs of integers to integers. The function declaration of `maxi` could be as follows.

```
maxi(x, y) | x >= y     = x
           | otherwise  = y
```

The expression on the right-hand side of a defining equation can in this way be restricted by a condition (i.e., an expression of type `Bool`). For evaluation we use the first equation for which the condition holds. Note that this case distinction does not have to cover all cases. The expression `otherwise` is actually a pre-defined function that always returns the Boolean constant `True`. Thus, the grammar rule for the construction of right-hand sides rhs of defining equations has to be changed as folllows:

$$\begin{array}{lcl} \underline{\mathsf{rhs}} & \rightarrow & = \underline{\mathsf{exp}} \mid \underline{\mathsf{condrhs}}_1 \ldots \ \underline{\mathsf{condrhs}}_n, \text{ where } n \geq 1 \\ \underline{\mathsf{condrhs}} & \rightarrow & \mid \underline{\mathsf{exp}} = \underline{\mathsf{exp}} \end{array}$$

Currying

To reduce the number of parentheses in expressions (and thereby increase readability), we often replace tuples of arguments by a series of arguments. This technique is named after the logician *Haskell B. Curry*, whose first name was already used for the name of the programming language. For illustration of this technique we first consider a conventional definition of the function `plus`.

```
plus :: (Int, Int) -> Int
plus (x, y) = x + y
```

Instead of this definition, we could also use the following one:

```
plus :: Int -> (Int -> Int)
plus x y = x + y
```

The process of transforming the first definition of `plus` into the second is called *currying*. For the type `Int -> (Int -> Int)` we could also just write `Int -> Int -> Int`, because we use the convention that the function type constructor `->` associates to the right. Function application however associates to the left, i.e., the expression `plus 2 3` corresponds to `(plus 2) 3`. Thus, an expression like `square square 3` is not typed correctly.

With the curried definition, `plus` gets two arguments in succession. More precisely, `plus` is now a function that gets an integer `x` as input. The result is the function `plus x`. This is a function from `Int` to `Int`, where `(plus x) y` computes the addition of `x`, and `y`.

Such function can also be called with just one argument. We call this *partial application*. The function `plus 1` is for example the successor function, that increments integers by `1`. Likewise, `plus 0` is the identity function on integers. In addition to saving on the number of parentheses, the possibility of applying functions to a lesser number of arguments is the second advantage of currying. All in all, the grammar rules for left-hand sides of defining equations have to be changed as follows:

$$\underline{\mathsf{funlhs}} \;\rightarrow\; \underline{\mathsf{var}}\; \underline{\mathsf{pat}}_1 \;\ldots\; \underline{\mathsf{pat}}_n, \quad \text{where } n \geq 1$$

Defining Functions using Pattern Matching

The arguments of the left-hand side of a defining equation do not have to be variables but may contain arbitrary *patterns* that serve as a pattern for the expected values of the argument. Let us consider the function `and`, that computes the conjunction of Boolean values.

```
and :: Bool -> Bool -> Bool
and True  y = y
and False y = False
```

In particular, we now have more than one function declaration (i.e., defining equations) for one and the same function symbol.

Here, `True` and `False` are pre-defined data constructors of the data type `Bool`, i.e., they are used to construct objects of that data type. Constructors in HASKELL always start with an upper-case letter.

Given a function call `and` $\underline{\mathsf{exp}}_1$ $\underline{\mathsf{exp}}_2$, to determine which defining equation to apply, we go through the equations from top to bottom and test which patterns match the current arguments $\underline{\mathsf{exp}}_1$ and $\underline{\mathsf{exp}}_2$ of our function `and`. The question can be answered by determining if there is a substitution that instantiates the patterns by concrete expressions such that the instantiated patterns are identical to $\underline{\mathsf{exp}}_1$ and $\underline{\mathsf{exp}}_2$. In this case we also say that the pattern $\underline{\mathsf{pat}}_i$ matches the expression $\underline{\mathsf{exp}}_i$. Then, the whole expression is evaluated by replacing the instance of the left-hand side by the corresponding instance of the right-hand side. For example, `and True True` evaluates to `True` because using the substitution [`y`/`True`] the

patterns `True` and `y` of the first defining equations are identical to the current arguments `True` and `True`.

As patterns are evaluated from top to bottom, the preceding definition of `and` is equivalent to the following alternative declaration.

```
and :: Bool -> Bool -> Bool
and True y = y
and x    y = False
```

If we define a function

```
unclear :: Int -> Bool
unclear x = not (unclear x)
```

whose evaluation does not terminate, then the evaluation of the expression `and False (unclear 0)` terminates nevertheless. The reason is that for performing the pattern matching, we do not need to evaluate the subexpression `unclear 0`. In contrast, `and (unclear 0) False` or `and True (unclear 0)` do not terminate.

The pattern matching used in the definition of the function `and` works as a value of type `Bool` can only be constructed from the data constructors `True` or `False`. Thus, Boolean values are constructed according to the following rule.[4]

$$\underline{\texttt{Bool}} \;\rightarrow\; \texttt{True} \mid \texttt{False}$$

Pattern matching is also possible for other data types. To show how pattern matching can be used for lists, we again consider the algorithm `len` for computing the length of a list.

```
len :: [data]   -> Int
len []       = 0
len (x : xs) = 1 + len xs
```

The pre-defined data structure for lists has the data constructors `[]` and `:`, such that lists can be constructed as follows:

$$\underline{\texttt{[data]}} \;\rightarrow\; \texttt{[]} \mid \underline{\texttt{data}} : \underline{\texttt{[data]}}$$

Here, `[]` represents the empty list and the (infix) constructor "`:`" is used to construct non-empty lists. As mentioned before, the expression `x:xs` represents the lists `xs` where we insert the element `x` at the beginning. Here, the element `x` has a type `data` and `xs` is a list of elements of the same type `data`. (Thus, the grammar defines how lists of type `[data]` are constructed.)

When evaluating the expression `len [15,70,36]`, we first transform the expression into the equivalent notation using "`:`". Thus, the argument of `len` is actually `15:(70:(36:[]))`. Now, we beginn the pattern matching process with the first defining equation. The first data constructor `[]` does not match the current argument that was constructed using the

[4]These grammar rules for <u>`Bool`</u>, <u>`[data]`</u>, and <u>`Int`</u> just serve to illustrate pattern matching and are not a part of the HASKELL language definition.

data constructor ":". But the pattern of the second defining equations matches this value using the substitution [`x/15, xs/70:(36:[])`]. Thus, the first step of the evaluation of this expression yields the new expression `1 + len (70:(36:[]))`. Continuing this process, we finally obtain the result `0`.

Similarly, we could define the following algorithm:

```
second :: [Int]   -> Int
second []           = 0
second (x : [])     = 0
second (x : y : xs) = y
```

We could also use the nicer list notation in the pattern of the second defining equation and replace this by the defining equation `second [x] = 0`.

Pattern matching is also possible for values of type `Int`. This is demonstrated by the following example for computing the factorial.

```
fac :: Int  -> Int
fac 0     = 1
fac (x+1) = (x+1) * fac x
```

During pattern matching, natural numbers are treated as if they were of the form `0` or `0 + 1 + ...+ 1`. For non-negative integers we thus obtain the following rule.

$$\underline{\texttt{Int}} \;\rightarrow\; 0 \,|\, \underline{\texttt{Int}} + 1$$

When computing the value of the expression `fac 2` we first represent 2 as `0 + 1 + 1`. By instantiating `x` with `0 + 1`, we can apply the second defining equation of `fac` and we obtain `(0 + 1 + 1) * (fac (0 + 1)) = 2 * (fac (0 + 1))`. Now we have to instantiate `x` with `0` to apply the second defining equation again. This yields `2 * ((0 + 1) * (fac 0)) = 2 * (1 * (fac 0))`. By applying the first equation we finally obtain `2 * (1 * 1) = 2 * 1 = 2`.

Finally, it should be remarked that HASKELL does not demand completeness of the defining equations. The given definition of `fac` ist for example undefined for negative numbers. This could be remedied by introducing a third equation:

$$\mathsf{fac}\,x = 1$$

When we use pattern matching with a pattern `x + 2`, instead of `x + 1 + 1` we have to write integers as `x + 2` etc. Thus, the algorithm for computing the rounded half of integers can be written as follows:

```
half :: Int  -> Int
half 0     = 0
half 1     = 0
half (x+2) = 1 + half x
```

Pattern Declarations

Declarations cannot only be used to define functions. We can also use declarations to determine other values:

```
pin :: Float
pin = 3.14159

suc :: Int -> Int
suc = plus 1

x0, y0 :: Int
(x0, y0) = (1,2)

x1, y1 :: Int
[x1,y1] = [1,2]

x2 :: Int
y2 :: [Int]
x2:y2 = [1,2]
```

Here, `Float` is the pre-defined type for floating point numbers. Note that pattern declarations like `[x1,y1] = []` or `[x1,y1] = [1,2,3]` lead to program errors.

In general, we may assign an expression to an arbitrary pattern. The only exception are patterns of the form var `+` integer. In the simplest case, a pattern is a variable. Otherwise, it is an expression as e.g. `(x0, y0)`, such that during an assignment of a value `(1,2)` to this pattern, we can uniquely determine which value the individual variables are assigned. In contrast to function declarations, there can only be one pattern declaration for each variable.

Thus, we extend the possibilites for declarations decl with pattern declarations as follows:

$$\begin{array}{rcl} \underline{\mathsf{decl}} & \rightarrow & \underline{\mathsf{typedecl}} \mid \underline{\mathsf{fundecl}} \mid \underline{\mathsf{patdecl}} \\ \underline{\mathsf{patdecl}} & \rightarrow & \underline{\mathsf{pat}}\ \underline{\mathsf{rhs}} \end{array}$$

Local Declarations

Local declarations are used to create a block of declarations inside a declaration. To this end, we can state a number of local declarations in each right-hand side of a function or pattern declaration after the keyword `where` . These declarations are local in the sense that they only refer to the right-hand side where they are declared. Note that this right-hand side may consist of more than one conditional right-hand side. Here, declarations from outside that use the same variable names are shadowed by the local declarations.

The grammar rules for fundecl and patdecl have to be changed accordingly. Here, square brackets in the grammar signify that the expressions enclosed by these brackets are optional.

$$\begin{array}{lcl} \underline{\text{fundecl}} & \rightarrow & \underline{\text{funlhs}}\ \underline{\text{rhs}}\ [\texttt{where}\ \underline{\text{decls}}] \\ \underline{\text{patdecl}} & \rightarrow & \underline{\text{pat}}\ \underline{\text{rhs}}\ [\texttt{where}\ \underline{\text{decls}}] \\ \underline{\text{decls}} & \rightarrow & \{\underline{\text{decl}}_1; \ldots; \underline{\text{decl}}_n\},\ \text{where } n \geq 0 \end{array}$$

As an example we consider the following program that computes the solutions of a quadratic equation with the help of the following formula.

$$ax^2 + bx + c = 0 \Longleftrightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
roots :: Float -> Float -> Float -> (Float, Float)
roots a b c = ((-b - (sqrt (b*b - 4*a*c)))/(2*a),
               (-b + (sqrt (b*b - 4*a*c)))/(2*a))
```

Using local declarations, we can simplify the definition significantly by introducing two new local variables `d` and `e`.

```
roots :: Float -> Float -> Float -> (Float, Float)
roots a b c = ((-b - d)/e, (-b + d)/e)
              where { d = sqrt (b*b - 4*a*c); e = 2*a }
```

Note that without local declarations, we would have to pass the values of `a`, `b`, and `c` to `d` and the value of `a` to `e` resulting in an undesirably complicated definition:

```
roots :: Float -> Float -> Float -> (Float, Float)
roots a b c = ((-b - (d a b c))/(e a), (-b + (d a b c))/(e a))
d a b c = sqrt (b*b - 4*a*c)
e a = 2*a
```

An important advantage of local declarations is that values computed for a local declaration are only computed once, i.e., different occurrence always reference the same expression. Evaluating the expression `roots 1 5 3` creates a graph

```
((-5 - ^d)/ ^e, (-5 + ^d)/ ^e),
```

where `^d` is a pointer to a memory cell with the expression `sqrt (5*5 - 4*1*3)` while `^e` is a pointer to the expression `2*1`. In this way, the two expressions only have to evaluated once and we can avoid evaluating identical expressions more than once.

To avoid curly braces and, thereby, to increase readability, we can use HASKELL's so-called *offside rule* for denoting (local) declarations:

1. The first symbol in a collection decls of declarations determines the left margin of the declaration block.

2. A new line beginning at this left margin is a new declaration in this block.

3. A new line beginning right of this left margin belongs to the same declaration as the previous line, i.e., it continues the previous line. For exampke, the declaration

```
d = sqrt (b*b -
          4*a*c)
```

is equivalent to

```
d = sqrt (b*b - 4*a*c).
```

4. A new line beginning left of this left margin signifies, that the <u>decls</u>block is finished and this line does not belong to that collection of declarations.

In general, we can use semicolons to write multiple declarations into one line. In local declarations we can also use curly braces to denote beginnings and ends of declarations blocks. If something does not fit in one line, it can just be put into the next one by indenting it with respect to the current left margin of the declaration block.

Thus, we can write `decls` as an indented program, i.e., as a sequence of declarations that start at the same left margin. For example, we could also write the declaration of the function `roots` as follows:

```
roots a b c = ((-b - d)/e, (-b + d)/e)
              where d = sqrt (b*b - 4*a*c)
                    e = 2*a
```

Operators and Infix Declarations

To increase the readability of programs written in HASKELL, som functions should be used in infix notation instead of the usual prefix notation. Examples are the functions `+`, `*`, `==` , or the list constructor `:`, that is used to insert elements in lists. Such function symbols are called *operators*. Analagously to the prefix function symbols we also differentiate between variables and constructors. The latter do not have any function declarations associated with them, but they are used to represent objects in a data structure. Operators in HASKELL are represented by sequences of special characters. Constructor operators (such as `:`) always start with a colon and variable operators (such as `+` or `==`) always start with a special character that is not a colon.

Each infix operator can be used as a prefix function by enclosing it in parentheses. In this way, we can write "`(+) 2 3`" instead of "`2 + 3`". Analogously, every binary prefix function (with a type <u>type</u>$_1$ `->` <u>type</u>$_2$ `->` <u>type</u>$_3$) can be used as an infix operator by using so-called "backquotes" . In this way, we can write "`2 ‘plus‘ 3`" instead of "`plus 2 3`". The use of infix operators is only a different notation for the same expressions. Thus, we will only use prefix functions in the following definitions of HASKELL's syntax. The use of alternative notations with infix operations can be helpful for making real programs more understandable, though.

Exactly as with the operators in Prolog, there are two important characteristics for infix operators:

1. *Associativity*
   Consider the following algorithm.

   ```
   divide :: Float -> Float -> Float
   divide x y = x / y
   ```

   For the expression

   `36 ‘divide‘ 6 ‘divide‘ 2`

   we do not know whether the result should be `3` or `12`. To determine the result of this expression uniquely, we have to know to which side this operator *associates*. To this end, we can declare associativity for infix operators. If `divide` should associate to the left, we can insert the following declaration.

   `infixl ‘divide‘`

   This associativity is also the default for operators in HASKELL, i.e., if there is no declaration given, all operators associate to the left. In this case, the above expression corresponds to

   `(36 ‘divide‘ 6) ‘divide‘ 2`

   and the result is indeed `3` and not `12`. In contrast, if we declare

   `infixr ‘divide‘`,

   then `‘divide‘` associates to the right. Then, the above expression corresponds to `36 ‘divide‘ (6 ‘divide‘ 2)`, such that we obtain `12` as the final result. A third possibility is the declaration

   `infix ‘divide‘`.

   This means that `‘divide‘` neither associates to the left nor to the right, i.e., that it does associate at all. Then, the expression `36 ‘divide‘ 6 ‘divide‘ 2` would lead to an error.

   We already used the concept of associtativity in HASKELL when looking at the function type constructor and function application. As mentioned before, the function type constructor `->` associates to the right, i.e., `Int -> Int -> Int` corresponds to `Int -> (Int -> Int)`. Function application associates to the left. Thus, an expression like `square square 3` corresponds to `(square square) 3`, i.e., to a expression that does not have a correct type and leads to an error.

2. *Binding Priority*
   We define the following two functions.

```
(%%) :: Int -> Int -> Int
x %% y = x + y

(@@) :: Int -> Int ->Int
x @@ y = x * y
```

The question is now what the result of evaluating the expression

```
1 %% 2 @@ 3
```

is. In other words, the question is which of the two operators `%%` and `@@` has a higher priority. To this end we can declare the binding priority in infix declarations (with `infixl`, `infixr`, or `infix`). The binding priority in Haskell is a number between `0` and `9` where `9` represents the highest binding priority.[5] In case no binding priority is declared, `9` is the default value. For example, we might declare the following.

```
infixl 9 %%
infixl 8 @@
```

Then the expression `1 %% 2 @@ 3` corresponds to `(1 %% 2) @@ 3` and the result is the value `9`. In contrast, if we swap the binding priorities `9` and `8`, the expression corresponds to `1 %% (2 @@ 3)` and we obtain the value `7`. If the binding priorities are equal, evaluation works from left to right, i.e., our expression would correspond to `(1 %% 2) @@ 3`.

As we have now added infix declarations, the grammar rules for declarations have to be extended again. Here we use big curly braces for representing a number of choices in a grammar rule.

$$\begin{array}{rcl}
\underline{\mathsf{decl}} & \rightarrow & \underline{\mathsf{typedecl}} \mid \underline{\mathsf{fundecl}} \mid \underline{\mathsf{patdecl}} \mid \underline{\mathsf{infixdecl}} \\
\underline{\mathsf{infixdecl}} & \rightarrow & \left\{ \begin{array}{l} \mathtt{infix} \\ \mathtt{infixl} \\ \mathtt{infixr} \end{array} \right\} \; [\left\{ \begin{array}{c} 0 \\ 1 \\ \vdots \\ 9 \end{array} \right\}] \; \underline{\mathsf{op}}_1, \ldots, \underline{\mathsf{op}}_n, \text{ where } n \geq 1 \\
\underline{\mathsf{op}} & \rightarrow & \underline{\mathsf{varop}} \mid \underline{\mathsf{constrop}}
\end{array}$$

Finally, it should be noted that operators (similarly to prefix functions) can be applied partially, i.e., an application is also possible, wenn not both needed arguments are given. For example, the expression `(+ 2)` is a function with type `Int -> Int` that increases numbers by `2`. The function `(6 ‘divide‘)` of type `Float -> Float` takes an argument and divides the number `6` by this argument. The function `‘divide‘ 6` is a different function of type `Float -> Float` that divides its argument by `6`, though.

[5] Note that this exactly opposite to the behaviour in Prolog where the lowest number signifies the hightest priority.

Summary of the Syntax for Declarations

In summary, we obtain the following grammar for producing declarations in HASKELL.

decl	→ typedecl | fundecl | patdecl | infixdecl	
typedecl	→ var_1, ..., var_n :: type,	where $n \geq 1$
var	→ string of letters and digits starting with a lower-case letter	
fundecl	→ funlhs rhs [`where` decls]	
funlhs	→ var $\text{pat}_1 \dots \text{pat}_n$	where $n \geq 1$
rhs	→ = exp| $\text{condrhs}_1 \dots \text{condrhs}_n$	where $n \geq 1$
condrhs	→ | exp = exp	
decls	→ { decl_1; ...; decl_n },	where $n \geq 0$
patdecl	→ pat rhs [`where` decls]	
infixdecl	→ { `infix` / `infixl` / `infixr` } [{ 0 / 1 / ⋮ / 9 }] $\text{op}_1, \dots, \text{op}_n$,	where $n \geq 1$
op	→ varop | constrop	
varop	→ string of special characters not starting with :	
constrop	→ string of special characters starting with :	

3.1.2 Expressions

Expressions exp (Expressions) are the central concept of functional programming. An expression describes a value (for example a number, a letter, or a function). Entering an expression into the interpreter leads to an evaluation of that expression. Furthermore, every expression has a type.

Then entering "`:t` exp" into the HASKELL interpreter HUGS, the type of exp is computed and printed to the console. Whenever an expression is entered, before starting the evaluation, the interpreter checks whether the expression is correctly typed. The evaluation is only started if this check is successful. Otherwise, an error message is given to the user that can then correct the expression.

An expression exp can be in one of the following forms:

- var
  Variable names such as `x` are expressions. As mentioned before, variable names in HASKELL are formed by string that start with a lower-case letter.

- constr
  Another possibility for an expression is a *data constructor*. Data constructors are used to build objects of a data structure and are introduced by data type declarations. In HASKELL, names for data constructors are formed by strings that start with an upper-case letter. Examples for such data constructors are the constructors `True` and `False` of the pre-defined data structure `Bool`. Another example are the data constructors `[]` and `:` for the pre-defined data structure of lists. Here, `[]` is a constructor that represents the empty list and the infix constructor `:` can be used to build non-empty

lists. The expression `x : xs` represents a list `xs` where we insert the element `x` at the beginning. The element `x` has the type `a` and `xs` is a list of elements of type `a`, i.e., it has type `[a]`.

- integer
  The integers `0, 1, -1, 2, -2, ...` are also expressions (of type `Int`).

- float
  Floating point numbers such as `-2.5` or `3.4e+23` are also expressions (of type `Float`).

- char
  Furthermore, characters such as sind `'a',...,'z','A',...,'Z','0',...,'9'` as well as the space `' '` and non-printable control characters like `'\n'` for line end are expressions. These characters are entered and printed enclosed in single quotes and are expressions of type `Char`.

- [$\text{exp}_1, \ldots, \text{exp}_n$], where $n \geq 0$
  Such an expression denotes a list of n expressions. As mentioned before, `[]` represents the empty list and `[0,1,2,3]` is just a more convenient notation (syntactic sugar) for the expression `0 : 1 : 2 : 3 : []`, where `:` associates to the right. All elements of a list have to be of the same type. The type of the preceding list would for example be `[Int]`, i.e., the type of lists of integers.

- string
  A string is a list of characters char, i.e., it is an expression of type `[Char]`). Instead of `['h','a','l','l','o']` we can use the nicer notation `"hallo"` where we enclose the string in double quotes. The pre-defined type `String` in HASKELL is identical to the type `[Char]`.

- ($\text{exp}_1, \ldots, \text{exp}_n$), where $n \geq 0$
  This expression is a tuple of expressions. In contrast to lists, the expressions in a tuple can be of different types. An example would be the expression `(10, False)`. This expressions would for example have the type `(Int, Bool)`. Unary typles (exp) are evaluated to just exp. The tuple with no elements, i.e., , has the special type `()`.

- ($\text{exp}_1 \ldots \text{exp}_n$), where $n \geq 2$
  Such an expression represents the application of a function exp_1 to the argument exp_2 and the successive application of the resulting function to possible further expressions. Here, we usually try to use parentheses only when necessary. As mentioned before, function application associates to the left. It also has the highest binding priority. Examples for such expressions are `square 10` (of Typ `Int`), `plus 5 3` (also of type `Int`), or `plus 5` (of type `Int -> Int`). Thus, the value of an expression can again be a function (and in case of $n > 3$ has to be a function).

- `if` exp_1 `then` exp_2 `else` exp_3
  Here, the expression exp_1 has to be of type `Bool` and the two expressions exp_2 and exp_3 have to be of the same type. When evaluating this kind of expression, HASKELL

first evaluates the expression $\underline{\text{exp}}_1$ until we know whether its value is `True` or `False`. Then, depending on this value, we continue to evaluate either $\underline{\text{exp}}_2$ or $\underline{\text{exp}}_3$.

Instead of

```
maxi(x, y) | x >= y     = x
           | otherwise  = y
```

we could also write the following declaration:

```
maxi(x, y) = if x >= y then x else y
```

- `let` $\underline{\text{decls}}$ `in` $\underline{\text{exp}}$
  This expression defines a local sequence of declarations $\underline{\text{decls}}$ that can only be used in the expression $\underline{\text{exp}}$. This is in many respects similar to local declarations using `where`. The main difference is that the local declaration is written before instead of after the expression.

  Instead of

```
roots a b c = ((-b - d)/e, (-b + d)/e)
              where d = sqrt (b*b - 4*a*c)
                    e = 2*a
```

  we could also write the following declaration:

```
roots a b c = let d = sqrt (b*b - 4*a*c)
                  e = 2*a
                  in ((-b - d)/e, (-b + d)/e)
```

  Note that there is indeed a difference between `let` and `where`. The latter always refers to the complete right-hand side of the declaration whereas `let` is local to the expression. This also means that while `let a = 21 in a+a` is an expression, `a+a where a = 21` is not. With both types of local declarations, we can use the offside rule for increased readability as demonstrated in the preceding example.

- `case` $\underline{\text{exp}}$ `of` $\{\underline{\text{pat}}_1$ `->` $\underline{\text{exp}}_1$ `; ... ;` $\underline{\text{pat}}_n$ `->` $\underline{\text{exp}}_n\}$, where $n \geq 1$
  This expression represents a sequence of pattern matching to try on the expression $\underline{\text{exp}}$. When evaluating such an expression, HASKELL first tries to match the expression $\underline{\text{exp}}$ with the pattern $\underline{\text{pat}}_1$. If there is indeed a substitution σ that instantiates $\underline{\text{pat}}_1$ to $\underline{\text{exp}}$, the result is the expression $\underline{\text{exp}}_1$, where the variables are instantiated by the matching substitution σ. If there is no such substitution, i.e., if pattern matching fails, we try to match $\underline{\text{exp}}$ using the pattern $\underline{\text{pat}}_2$. This process is continued until we either have found a matching pattern or we have exhausted all possibilities. If the latter is the case, HASKELL reports a program error. Like with local declarations, to ease readability, we can use the offside rule.

  Instead of

```
and True  y = y
and False y = False
```

we can also write the following declaration:

```
and x y = case x
            of True  -> y
               False -> False
```

Furthermore, instead of expressions $\underline{\text{exp}}_i$, we can also use sequences of conditional expressions `|` $\underline{\text{exp}}_{i,j,1}$ `->` $\underline{\text{exp}}_{i,j,2}$. It is also possible to use local declarations with `where` in every alternative of the `case` expression.

- \ $\underline{\text{pat}}_1 \dots \underline{\text{pat}}_n$ `->` $\underline{\text{exp}}$, where $n \geq 1$
Such an expressions is called a "lambda expression" or "lambda abstraction", because the character \ (backslash) is used to represent the greek letter λ. The value of this expression is the function that maps the arguments $\underline{\text{pat}}_1 \dots \underline{\text{pat}}_n$ to $\underline{\text{exp}}$. For example, the value of the expression `\x -> 2 * x` is the function, that takes a number as an argument and doubles it. The type of this function is `Int -> Int`. Thus, with "lambda" we can build so-called "anonymous functions", that can only be used exactly where they are defined. The expression

```
(\x -> 2 * x) 5
```

consequently evaluates to the value `10`. The function `\x y -> x + y` is the function for addition of type `Int -> Int -> Int`. In general, the expression \ $\underline{\text{pat}}_1 \dots \underline{\text{pat}}_n$ `->` $\underline{\text{exp}}$ has the type $\underline{\text{type}}_1$ `->...->` $\underline{\text{type}}_n$ `->` $\underline{\text{type}}$, if $\underline{\text{pat}}_i$ has the type $\underline{\text{type}}_i$ for all i, and $\underline{\text{exp}}$ has the type $\underline{\text{type}}$.

We can use arbitrary patterns in the definition of lambda expressions, i.e., we can also for eample use expressions such as `\(x, y) -> x + y` of type `(Int, Int) -> Int`. When looking at lambda expressions, we clearly see that functions really are first-class citizens in functional programming languages as they can simply be defined and used by an appropriate expression.

Instead of the function declaration

```
plus x y = x + y
```

we could also write

```
plus = \x y -> x + y
```

or even the following:

```
plus x = \y -> x + y
```

Summary of the Syntax for Expressions

In summary, we obtain the following grammar for expressions in HASKELL.

$$
\begin{array}{lcll}
\underline{\mathsf{exp}} & \rightarrow & \underline{\mathsf{var}} & \\
 & | & \underline{\mathsf{constr}} & \\
 & | & \underline{\mathsf{integer}} & \\
 & | & \underline{\mathsf{float}} & \\
 & | & \underline{\mathsf{char}} & \\
 & | & \texttt{[}\underline{\mathsf{exp}}_1, \ldots, \underline{\mathsf{exp}}_n\texttt{]}, & \text{where } n \geq 0 \\
 & | & \underline{\mathsf{string}} & \\
 & | & (\underline{\mathsf{exp}}_1, \ldots, \underline{\mathsf{exp}}_n), & \text{where } n \geq 0 \\
 & | & (\underline{\mathsf{exp}}_1 \ \ldots \ \underline{\mathsf{exp}}_n), & \text{where } n \geq 2 \\
 & | & \texttt{if } \underline{\mathsf{exp}}_1 \texttt{ then } \underline{\mathsf{exp}}_2 \texttt{ else } \underline{\mathsf{exp}}_3 & \\
 & | & \texttt{let } \underline{\mathsf{decls}} \texttt{ in } \underline{\mathsf{exp}} & \\
 & | & \texttt{case } \underline{\mathsf{exp}} \texttt{ of } \{\underline{\mathsf{pat}}_1 \texttt{ -> } \underline{\mathsf{exp}}_1; \ldots; \underline{\mathsf{pat}}_n \texttt{ -> } \underline{\mathsf{exp}}_n\}, & \text{where } n \geq 1 \\
 & | & \backslash \underline{\mathsf{pat}}_1 \ldots \underline{\mathsf{pat}}_n \texttt{ -> } \underline{\mathsf{exp}}, & \text{where } n \geq 1
\end{array}
$$

$\underline{\mathsf{constr}} \rightarrow$ string of letters and digits starting with an upper-case letter

3.1.3 Patterns

In function declarations and lambda expressions, we have used so-called *patterns* to match arguments. In the most general sense, a pattern restricts the form of allowed arguments, i.e., of expressions a function is applied to. Analogously, in pattern declarations and case expressions, we have used patterns to match expressions. The syntax of patterns is consequently similar to that of expressions as patterns can be viewed as prototypes for the expected values. The form of values is described by the occurring data constructors where instead of certain subexpressions a pattern may contain a variable. Thus, here, we use data constructors for disassembling an object instead of constructing it. As mentioned before, a pattern matches an expression if there is a substitution that replaces the variables in the pattern by expressions in such a way that we obtain exactly this expression. As an example we already considered the algorithms `and`, `len`, `second`, `fac`, and `half` in the previous section on function declarations.

As another example, let us consider the algorithm `append`. The analogous infix operator `++` on lists with elements of arbitrary types is pre-defined in HASKELL.

```
append :: [Int]  -> [Int]  -> [Int]
append []     ys = ys
append (x:xs) ys = x : append xs ys
```

To compute the value of the expression `len (append [1] [2])`, the argument `append [1] [2]` of `len` has to be evaluated until we can decide which pattern in the definition of `len` will match it. Here, we would only evaluate `append [1] [2]` to the expression `1:append [] [2]`. At this point it is already clear that the second equation of `len` has to be applied

and we obtain the new expression `1 + len (append [] [2])`. We continue in this way and finally obtain `2`.

Whenever we cannot determine if a pattern matches without evaluating an argument expression, we evaluate that expression until it starts with a constructor.[6] Then we can check if this constructor is the same as the outermost constructor of the pattern. If this is the case, we call the pattern matching algorithm recursively for the arguments of the expression and the pattern.

Consider for example the following declarations:

```
zeros :: [Int]
zeros = 0 : zeros

f :: [Int]  -> [Int]  -> [Int]
f [] ys = []
f xs [] = []
```

The evaluation of `f [] zeros` terminates although `zeros` alone does not terminate. The reason is that no evaluation of `zeros` is needed to determine that the first equation of `f` is applicable. More surprisingly, also the expression `f zeros []` terminates. Here, we first evaluat `zeros` in one step to the expression `0 : zeros`. Now, the outermost constructor of `f`'s argument is determined to be "`:`". As this is not the same constructor as `[]` used in the pattern for the first argument of `f`, the first equation cannot be applicable. Thus, the second equation is used. There, `zeros` does not need to be evaluated as it directly is matched by the variable pattern `xs`. Thus, we use the second equation and the computation terminates with the result `[]`.

An example for the application of pattern matching in pattern declarations is the following:

```
let x:xs = [1,2,3] in xs
```

Here, `x:xs` is a pattern that matches the expression `[1,2,3]`. The matching is successful using the matching substitution [`x/1, xs/[2,3]`]. Thus, the above expression would be evaluated to `[2,3]`.

In contrast to Prolog, patterns are restricted to be *linear*, i.e., no variable may occur in a pattern more than once. The reasons for this is that otherwise, not all evaluation strategies would yield the same result, and that one would have to check equivalence of functions (which is undecidable).

For example, we could then declare the following function:

```
equal :: [Int] -> [Int] -> Bool
equal xs xs     = True
equal xs (x:xs) = False
```

The expression `equal zeros zeros` could now be evaluated both to `True` and to `False` depending on the evaluation strategy. In general, a pattern <u>pat</u> can be of the following form:

[6] This form is often referred to as the *Weak Head Normal Form* or WHNF.

- var
  Every variable name is also a pattern. This pattern matches every value and the substitution just instantiates this variable to the matched value. An example for a function declaration with this pattern is the definition of `square`:

  ```
  square x = x * x.
  ```

- _
  The character _ (underscore) is the joker pattern. It also matches every value, but no variable needs to be instantiated. The joker pattern is consequently allows to occur more than once in a pattern. For example, we might also define the function `and` as follows:

  ```
  and True y = y
  and _ _ = False
  ```

- integer or float or char or string
  These patterns only match themselves and there is no need to instantiate any variables by the matcing substitution.

- (constr $\text{pat}_1 \ldots \text{pat}_n$), where $n \geq 0$
  Here, constr is an n-ary data constructor. This pattern matches values that are constructed using the same data constructur if, and only if, for all i, the pat_i match the i-th argument of the value. Examples for this patterns can be found in the declarations of the algorithms `and`, `len`, and `append`. Here, "`:`" as an infix constructor can be the outermost symbol without starting the expression - compare the notations `x : xs` and `((:) x xs)`. Typically we try to avoid parentheses as far as possible to improve readability.

- var@pat
  This pattern behaves exactly like pat, but if pat matches the expression, additionally we instantiate the variable var by the complete expression matched. As an example, consider the following function for duplication the first element of a list.

  ```
  f [] = []
  f (x : xs) = x : x : xs
  ```

  We could now replace the second defining equation by the following one.

  ```
  f y@(x : xs) = x : y
  ```

- (var + integer)
  Natural numbers are treated during pattern matching as if they were constructed with the help of (non-existing) data constructors `0 :: Int` and `...+ 1 :: Int -> Int`. Here, we use `Int` only for non-negative integers. A number k would then be represented as $0 + 1 \ldots + 1$ using k 1s. The pattern $\text{var} + k$ (with $k \in \mathbb{N}$) can then

be viewed as an abbreviation for var+1 ... + 1 with k 1s. The pattern var + k only matches natural numbers n with $n \geq k$ and the variable var is instantiated to the value $n-k$. We have seen examples in the definitions for the functions `fac` and `half`. As another example, consider the following function.

```
sub7 :: Int -> Int
sub7 (x + 7) = x
```

The function `sub7` subtracts `7` from its argument, if this is greater or equal than7. Otherwise, the value of `sub7` is undefined.

To decide whether var + k or just a number integermatch an expression of type `Int`, this expression has to be evaluated *completely*. The effect of this is illustrated by the following example..

```
infinity :: Int
infinity = infinity + 1

f :: Int  -> Int  -> Int
f 0 y = 0
f x 0 = 0
```

The evaluation of `f 0 infinity` terminates, while evaluation of `f infinity 0` does not terminate. The reason is that while we can evaluate `infinity` to `infinity + 1`, we cannot deduce that the first equation is not applicable. `infinity` could simply have the value `-1` which is of type `Int` and then `infinity + 1` would be matched by the pattern `0`. entsprechen. Consider for example the function defined by the equations `g (x+1) = g x + 1; g 0 = -1`. Here, the expression `g 1` evaluates to an expression `g 0 + 1` which finally yields `0`.

In contrast to other patterns, such patterns may not be used in pattern declarations. An expression such as `let (n+7) = 8 in n` is not allowed.

- [$\text{pat}_1, \ldots, \text{pat}_n$], where $n \geq 0$
  Such a pattern matches lists of length n, if for all i, pat_i matches the i-th element of the list. The following example defines a function that recognizes lists of length exactly `3`.

```
has_length_three :: [Int] -> Bool
has_length_three [x,y,z] = True
has_length_three _       = False
```

- ($\text{pat}_1, \ldots, \text{pat}_n$), where $n \geq 0$
  Analogously to the list pattern, such a tuple patterns matches tuples consisting of n compontents if, and only if, for all i, pat_i matches the i-th component of the typle. The pattern `()` matches only the value `()`. Here, we can alternatively define `maxi` as follows.

```
maxi :: (Int, Int) -> Int
maxi (0,y)     = y
maxi (x,0)     = x
maxi (x+1,y+1) = 1 + maxi (x,y)
```

Thus, in general, every linear term consisting of data constructors and variables is a pattern.

Summary of the Syntax for Patterns

We obtain the following rules for constructing patterns in HASKELL.

$$
\begin{array}{lll}
\underline{\mathsf{pat}} & \rightarrow & \underline{\mathsf{var}} \\
 & \mid & _ \\
 & \mid & \underline{\mathsf{integer}} \\
 & \mid & \underline{\mathsf{float}} \\
 & \mid & \underline{\mathsf{char}} \\
 & \mid & \underline{\mathsf{string}} \\
 & \mid & (\underline{\mathsf{constr}}\ \underline{\mathsf{pat}}_1\ \ldots\ \underline{\mathsf{pat}}_n), \quad \text{where } n \geq 0 \\
 & \mid & \underline{\mathsf{var}}@\underline{\mathsf{pat}} \\
 & \mid & (\underline{\mathsf{var}} + \underline{\mathsf{integer}}) \\
 & \mid & [\underline{\mathsf{pat}}_1, \ldots, \underline{\mathsf{pat}}_n], \quad \text{where } n \geq 0 \\
 & \mid & (\underline{\mathsf{pat}}_1, \ldots, \underline{\mathsf{pat}}_n), \quad \text{where } n \geq 0
\end{array}
$$

3.1.4 Types

Every expression in HASKELL has a type. Types are sets of related values, that are denoted by a corresponding type expression. Example for types that we already know are the pre-defined types `Bool`, `Int`, `Float`, and `Char` as well as constructed types such as `(Int,Int)`, `Int -> Int`, `(Int,Int) -> Int`, `[Int]`, `[Int -> Bool]`, `[[Int]]`, etc. In general, we have the following kinds of types $\underline{\mathsf{type}}$:

- $(\underline{\mathsf{tyconstr}}\ \underline{\mathsf{type}}_1\ \ldots\ \underline{\mathsf{type}}_n)$, where $n \geq 0$
  Types are generally constructed with the help of *type constructors* $\underline{\mathsf{tyconstr}}$ from other types $\underline{\mathsf{type}}_1, \ldots, \underline{\mathsf{type}}_n$. Examples for nullary (and pre-defined) type constructors are `Bool`, `Int`, `Float`, and `Char`. In HASKELL, type constructors are denoted by strings starting with an upper-case letter, i.e., they use the same names as data constructors that are used to construct objects in data structures instead of type. Here, once more we only write parentheses if they are strictly necessary.

- $[\underline{\mathsf{type}}]$
  Another pre-defined type constructor is the unart constructor `[...]`, that takes a type as argument and constructs a new type, for which the objects are lists of elements of the original type. Instead of writing `[...]` $\underline{\mathsf{type}}$, we write $[\underline{\mathsf{type}}]$. Examples for such types are `[Int]` and `[[Int]]` (the type of lists of lists of integers).

- ($\underline{\text{type}}_1$ -> $\underline{\text{type}}_2$)
  Another pre-defined type constructor is the function type constructor `->`, that takes two types as arguments and generates a new type of functons between these two types. An example for this is the type `Int -> Int` that is e.g. the type of the function `square` for squaring integers.

- ($\underline{\text{type}}_1, \ldots, \underline{\text{type}}_n$), where $n \geq 0$
  Furthermore, there is the pre-defined tuple constructor that can be used to construct tuple types. This constructor takes an arbitrary number of argument types and constructs a type of tuples where the components are objects from these types.

  An example for this is the type `(Int, Bool, [Int -> Int])`. We will also see that in addition to these pre-defined type constructors, the user can also define further type constructors.

- <u>var</u>
  Finally, a simple variables is also a type. We call this variable a *type variable*. These types are need for parametric polymorphism as will see now.

Parametric Polymorphism

"Polymorphism" stems from ancient greek and means "many forms". This notion is usually used in computer science to express that identical functions or functions with the same name can be used for different kinds of arguments. Here, we differentiate between *parametric polymorphism* and *ad-hoc polymorphism*. When we talk about parametric polymorphism, one and the same function is applied to arguments of different types. When we talk about ad-hoc polymorphism , the same function symbol is applied to arguments of different type, but depending on the type of the arguments, different function implementations are actually executed. While functional languages like HASKELL support both kinds of polymorphism, we will mostly focus on parametric polymorphism in this course.

Let us now indeed consider parametric polymorphism. Here, a function is used in an uniform way to a collection of data objects. Simple examples are the following two functions.

```
id :: a -> a
id x = x

len :: [a] -> Int
len []     = 0
len (x:xs) = len xs + 1
```

The type of the functions `id` and `len` contains a type variable `a`. This means that the function is applicable for ever possible instantiation of the type variable by actual types. For example, we can evaluate both the expression `len [True, False]` and the expression `len [1,2,3]`.

Analogously, the function `append` and its pre-defined variant `++` are also defined using parametric polymorphism.

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

The multiple occurrence of the same type variable `a` in the type `[a] -> [a] -> [a]` forces that the types of the two arguments of `++` coincide. A function of the type $\underline{\mathsf{type}}_1$ `->` $\underline{\mathsf{type}}_2$ kann be applied to an argument of type $\underline{\mathsf{type}}$ if, and only if, there is a most general unifier σ such that $\sigma(\underline{\mathsf{type}}_1) = \sigma(\underline{\mathsf{type}})$. The result then has the type $\sigma(\underline{\mathsf{type}}_2)$.

As an example, consider the expression `[True] ++ []`. The subexpression `[True]` has the type `[Bool]` while the second argument `[]` has the type `[b]`. The most general unifier is a substitution σ such that $\sigma(\texttt{[a]}) = \sigma(\texttt{[Bool]}) = \sigma(\texttt{[b]})$. Here, we have $\sigma = [\,\texttt{a/Bool},\ \texttt{b/Bool}\,]$. Thus, this expression is correctly typed and has the type `[Bool]`. The checking of type correctness and the computation of most general types can be performed automatically using a unificiation algorithm like the one used by Prolog.

Type Definitions: Introduction of new Types

To introduce new types and type constructors, respectively, HASKELL offers special kinds of declarations. In contrast to all other kinds of declarations, these declarations can only be defined on the top level and not in local blocks of declarations. For this reason, we now differentiate between general declarations $\underline{\mathsf{decl}}$ and declarations $\underline{\mathsf{topdecl}}$, that are only allows at the top level. A program is then a sequence of left-aligned $\underline{\mathsf{topdecl}}$declarations. The grammar rule for $\underline{\mathsf{topdecl}}$ can be given as follows:

$$
\begin{array}{lll}
\underline{\mathsf{topdecl}} \rightarrow & \underline{\mathsf{decl}} & \\
 & \mid\ \texttt{type}\ \underline{\mathsf{tyconstr}}\ \underline{\mathsf{var}}_1 \ldots \underline{\mathsf{var}}_n = \underline{\mathsf{type}}, & \text{where } n \geq 0 \\
 & \mid\ \texttt{data}\ \underline{\mathsf{tyconstr}}\ \underline{\mathsf{var}}_1 \ldots \underline{\mathsf{var}}_n = & \\
 & \qquad \underline{\mathsf{constr}}_1\ \underline{\mathsf{type}}_{1,1} \ldots \underline{\mathsf{type}}_{1,n_1} & \\
 & \qquad \mid \ldots & \\
 & \qquad \mid \underline{\mathsf{constr}}_k\ \underline{\mathsf{type}}_{k,1} \ldots \underline{\mathsf{type}}_{k,n_k}, & \text{where } n \geq 0, k \geq 1, n_i \geq 0
\end{array}
$$

In particular, all other declarations $\underline{\mathsf{decl}}$ can occur on top level. Additionally, we can use the key words `type` and `data` to introduce new types.

The first possibility to define new types and type constructors, respectively, is to use *type synonyms*. These abbreviations are declared using the keyword `type` . For example, we can define three new type constructors by the following three top-level declarations.

```
type Position = (Float, Float)
type String   = [Char]
type Pair a b = (a, b)
```

Note that `String` has already been pre-defined in this way in the HASKELL libraries. The type (or rather the nullary type constructor) `Position` is just an abbreviation for the type `(Float, Float)`, i.e., these two types are considered to be equal. A type synonym may have arguments, i.e., `Pair` is a binary type constructor. Also in this case, this type is just an abbreviation, i.e., the types `Pair Float Float` and `Position` are identical.

The restrictions for type synonyms are that the variables $\underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n$ have to be pairwise distinct and that the type $\underline{\mathsf{type}}$ on the right-hand side contains no other variables than these. These restrictions are very similar to the restrictions for function declarations. Furthermore, type synonyms may not be recursive, i.e., the type $\underline{\mathsf{type}}$ may not depend on the type constructor $\underline{\mathsf{tyconstr}}$just being defined.

The other possibility for defining new types is the introduction of algebraic data types by giving a EBNF-like context-free grammar. This can be done by using the keyword `data`. For example, we can define the following enumeration types.

```
data Color = Red | Yellow | Green
data MyBool = MyTrue | MyFalse
```

A definition of an algebraic data type like `Color` is an enumeration of the different possibilities, how one can construct objects with corresponding data constructors (such as `Red`, `Yellow`, `Green`). By this definition, two new nullary type constructors `Color` and `MyBool` have been introduced. The following two functions demonstrate how pattern matching can be used also for self-defined data structures. Remember that every linear term constructed from variables and data constructors is a pattern.

```
traffic_light :: Color -> Color
traffic_light Red    = Green
traffic_light Green  = Yellow
traffic_light Yellow = Red

and :: MyBool -> MyBool -> MyBool
and MyTrue y = y
and _      _ = MyFalse
```

It is not possible to directly print values of self-defined types. Whenever a type is printed to the screen, a pre-defined function `show` is called to transform the value into a `String` (just like `toString` in Java). This function exists for pre-defined types. For self-defined types, it has to be written by the user. Alternatively, it can be generated automatically. To this end one adds `deriving Show` when declaring the new data type. To define an own implementation of this function, one has to introduce the concept of type classes in HASKELL or use functons of the form `printColor :: Color -> String` etc.

Let us consider another example, where the data structure of the natural numbers is defined.

```
data Nats = Zero | Succ Nats
```

The data type `Nats` has to data constructors `Zero` and `Succ`. The data type declaration specifies the types of the arguments of these constructors after their names. Thus, `Zero` has no arguments, i.e., it is simply of type `Nats`. The second data constructor `Succ` has the type `Nats -> Nats`. If `Succ` represents the successor function, then `Succ (Succ Zero)` represents the natural number 2. Now we can define functions for this data type like `plus` orr `half`.

```
plus :: Nats -> Nats -> Nats
plus Zero     y = y
plus (Succ x) y = Succ (plus x y)

half :: Nats -> Nats
half Zero            = Zero
half (Succ Zero)     = Zero
half (Succ (Succ x)) = Succ (half x)
```

The difference between `Nats` and `Int` becomes clear, if we consider the `infinity` example now defined on our new data type `Nats`.

```
infinity :: Nats
infinity = Succ infinity

f :: Nats  -> Nats  -> Nats
f Zero y    = Zero
f x    Zero = Zero
```

In contrast to the version working with the type `Int`, the evaluation of the expression `f infinity Zero` now terminates. The reason is that when evaluating `infinity` for one step, we can already determine that the result of the evaluation will start with `Succ` beginnt. As the representation using data constructors is unique, the first equation of `f` can never be applicable. Thus, we can use the second equation and obtain the result `Zero`.

Of course, we can also define parametric types (i.e., non-nullary type constructors). The following declaration shows how we can define a type of lists that corresponds to the pre-defined lists in HASKELL.

```
data List a = Nil | Cons a (List a)
```

By this declaration, a new parametric algebraic data type `List a` for lists with elements of type `a` is introduced. `List` is a unary type constructor, i.e., `List Int` would be the type of lists of integers. The data type `List a` has to data constructors `Nil` and `Cons`. `Nil` has no arguments, i.e., its type is simply `List a`. The other data constructor `Cons` has the type `a -> (List a) -> (List a)`. If `Cons` represents the insertion of a new element into a list, then `Cons 1 Nil` represents the singleton list with the element `1`. The following examples show the algorithms `len` and `append` working on self-defined lists and natural numbers.

```
len :: List a -> Nats
len Nil         = Zero
len (Cons x xs) = Succ (len xs)

append :: List a -> List a -> List a
append Nil         ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

As with type synonyms, we also have to make sure that the variables $\underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n$ used in the definition of an algebraic data type are pairwise distinct and that the type $\underline{\mathsf{type}}$ on

the right-hand side contains no other variables than these. In contrast to type synonyms, data types may be defined rekursively. This is for example the case in the definition of `Nats`, where objects of type `Nats` can be constructed using a data constructor `Succ`, whose argument is again of type `Nats`. Analgously, the parametric data type `List a` is also defined recursively.

It is also possible to define mutually recursive data types. In the following example, the type constructor `Tree` is defined using the data type `Forest` while the definition of `Forest` needs `Tree`. Here, the data type `Tree` can be used to implement trees with arbitrary branching and `Forest` represents lists of such trees.

```
data Tree element = Node element (Forest element)

data Forest element = NoTrees | Trees (Tree element) (Forest element)
```

Summary of the Syntax for Types

The syntax rules for types and type definitions can be summarized as follows:

$$
\begin{array}{lll}
\underline{\text{type}} & \rightarrow & (\underline{\text{tyconstr}}\ \underline{\text{type}}_1\ \ldots\ \underline{\text{type}}_n), \quad \text{where } n \geq 0 \\
& | & [\underline{\text{type}}] \\
& | & (\underline{\text{type}}_1\ \texttt{->}\ \underline{\text{type}}_2) \\
& | & (\underline{\text{type}}_1, \ldots, \underline{\text{type}}_n), \quad \text{where } n \geq 0 \\
& | & \underline{\text{var}}
\end{array}
$$

$$\underline{\text{tyconstr}} \rightarrow \text{string of letters and digits starting with an upper-case letter}$$

$$
\begin{array}{lll}
\underline{\text{topdecl}} & \rightarrow & \underline{\text{decl}} \\
& | & \texttt{type}\ \underline{\text{tyconstr}}\ \underline{\text{var}}_1 \ldots \underline{\text{var}}_n = \underline{\text{type}}, \quad \text{where } n \geq 0 \\
& | & \texttt{data}\ [\underline{\text{context}} \Rightarrow]\ \underline{\text{tyconstr}}\ \underline{\text{var}}_1 \ldots \underline{\text{var}}_n = \\
& & \quad \underline{\text{constr}}_1\ \underline{\text{type}}_{1,1} \ldots \underline{\text{type}}_{1,n_1} \\
& & \quad | \ldots \\
& & \quad | \ \underline{\text{constr}}_k\ \underline{\text{type}}_{k,1} \ldots \underline{\text{type}}_{k,n_k}, \quad \text{where } n \geq 0, k \geq 1, n_i \geq 0 \\
\underline{\text{context}} & \rightarrow & (\underline{\text{tyconstr}}_1\ \underline{\text{var}}_1, \ldots, \underline{\text{tyconstr}}_n\ \underline{\text{var}}_n), \quad \text{where } n \geq 1 \\
\underline{\text{typedecl}} & \rightarrow & \underline{\text{var}}_1, \ldots, \underline{\text{var}}_n :: [\underline{\text{context}} \Rightarrow]\ \underline{\text{type}}, \quad \text{where } n \geq 1
\end{array}
$$

3.2 Higher-order Functions

Higher-order functions are functions that are characterized by the fact that some of their arguments and/or their result are functions. For example, the function `square :: Int -> Int` is a first-order function whereas `plus :: Int -> Int -> Int` is a higher-order function, because `plus 1` (the result of `plus` after application to the argument `1`) is again a function. We will now first consider some typical higher-order functions that take functions as their arguments.

Function Composition "."

A very often used higher-order function is the function composition $(f \circ g)$. In HASKELL, function composition for two unary functions is already pre-defined as an infix operator `.`. If `g` is a function of type `a -> b` and `f` a function of type `b -> c`, then `f.g` is the function that results from first applying `g` and then applying `f`.

```
infixr 9 .
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

For example, `half.square` is the function that takes an argument x, first squares it and then halves it, i.e., it computes $\frac{x^2}{2}$). For example, the evaluation of the expression `(half.square) 4` yields the result `8` $(= \frac{4^2}{2})$ and `((\x -> x+1).square) 5` yields the value `26`.

The Functions `curry` and `uncurry`

As mentioned before, we denote the process of transforming tupled arguments into a sequence as arguments as *currying*. This happens for example when going from the first of the following two definitions of `plus` to the second by currying. The step backwards is called *uncurrying*.

```
plus :: (Int, Int) -> Int
plus (x,y) = x + y

plus :: Int -> Int -> Int
plus x y = x + y
```

In general, this transformation can be carried out by the following two higher-order functions (which are pre-defined in the HASKELL libraries).

```
curry :: ((a,b) -> c) -> a -> b -> c
curry f = g
      where g x y = f (x,y)

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry g = f
        where f (x,y) = g x y
```

Not surprisingly, we have `curry (uncurry g) = g` and `uncurry (curry f) = f`. Using these two functons, we can now use a function arbitrarily in its curried or its uncurried variant. The evaluation of `uncurry (+) (1,2)` for example results in the value `3`.

The `map` Function

In particular, higher-order functions allow to structure problems and programs in a more concise and more readable way. To this end, we typically use a set of quite universal higher-order functions that implement different popular recursion patterns on the given

data structure. Programs written using these higher-order functions are more readable easier to reuse. In the following we present some of these classical higher-order functions.

Let us consider a function `suclist` that increases all numbers in a list of integers by `1`.

```
suc :: Int -> Int
suc = plus 1

suclist :: [Int] -> [Int]
suclist []     = []
suclist (x:xs) = suc x : suclist xs
```

If we call `suclist` with the argument

$$[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n]$$

we obtain the result

$$[\mathtt{suc}\,\mathtt{x}_1,\ \mathtt{suc}\,\mathtt{x}_2,\ \ldots, \mathtt{suc}\,\mathtt{x}_n].$$

Analogously, the function `sqrtlist` computes the square root for each element of a list of floating point numbers.

```
sqrtlist :: [Float] -> [Float]
sqrtlist []     = []
sqrtlist (x:xs) = sqrt x : sqrtlist xs
```

If we call `sqrtlist` with the argument

$$[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n]$$

we obtain the result

$$[\mathtt{sqrt}\,\mathtt{x}_1,\ \mathtt{sqrt}\,\mathtt{x}_2,\ \ldots, \mathtt{sqrt}\,\mathtt{x}_n].$$

We observe that the two functions `suclist` and `sqrtlist` are very similar, as both functions work through a list element by element and apply to each of these elements a functon (`suc` and `sqrt`, respectively). Then, the processed list is returned, i.e., the strucutre of the list remains unchanged. It seems obvious to implement these two function by one function that performs that part of the processing that is common to both. To this end, we have to take the following steps.

- Abstraction of the data type of the list elements (`Int` and `Float`, respectively). This is only possible in programming languages with (parametric) polymorphism.
- Abstraction from the function that is applied to each element of the list (`suc` and `sqrt`, respectively). This is only possible in programming languages, where functions can be treated as data objects.

Thus, in general, a function `g` is applied to all elements of the list. Therefore we need a function function `f` such that when we call `f` with the argument

$$[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n]$$

we obtain the following result.

$$[\mathtt{g}\,\mathtt{x}_1,\ \mathtt{g}\,\mathtt{x}_2,\ \ldots, \mathtt{g}\,\mathtt{x}_n]$$

The general form of functions of this schema is then as follows.

```
f :: [a] -> [b]
f [] = []
f (x:xs) = g x : f xs
```

As `g` is an arbitrary function, we should pass it as another argument of the function. In this way we obtain the function `map` (where `f` as above now corresponds to the function `map g`).

```
map :: (a -> b) -> [a] -> [b]
map g [] = []
map g (x:xs) = g x : map g xs
```

The function `map` is a higher-order function as both its result and one of its arguments are functions. As we would expect, this function is already pre-defined in the HASKELL libraries. The result of `map g` $[x_1, x_2, \ldots, x_n]$ is

$$[g\,x_1,\; g\,x_2,\; \ldots, g\,x_n].$$

Thus, we implement the recursion pattern "Go through the list and apply a function to each of the elements" by a higher-order function `map`. The use of a given set of such functions that implement recursion patterns leads to higher modularisation, reuse, and readability of programs.

The functions `suclist` and `sqrtlist` as defined above can now be implemented non-recursively by the following simple definitions.

```
suclist  l = map suc  l
sqrtlist l = map sqrt l
```

We can make it even easier by dropping the argument `l`.

```
suclist  = map suc
sqrtlist = map sqrt
```

Analogously to the case for pre-defined lists, we can build `map` functions for other data structures, for example for the self-defined lists that we defined as follows.

```
data List a = Nil | Cons a (List a)
```

Here the corresponding definition of a `map` function is as follows:

```
mapList :: (a -> b) -> List a -> List b
mapList g Nil         = Nil
mapList g (Cons x xs) = Cons (g x) (mapList g xs)
```

As a further example, let us consider the following data structure of arbitrarily branching trees. Compared to our previous definition, we directly implement the type `Forest` using pre-defined lists, i.e., as `[Tree]`.

```
data Tree a = Node a [Tree a]
```

The map function for this data structure can then be defined as follows.

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree g (Node x ts) = Node (g x) (map (mapTree g) ts)
```

When calling mapTree g t, the function g is called for the element of each node of the tree t. If t is the tree

$$\texttt{Node}\ x_1\ \texttt{[Node}\ x_2\ \texttt{[]]},$$

evaluating mapTree g t results in the tree

$$\texttt{Node}\ (g\,x_1)\ \texttt{[Node}\ (g\,x_2)\ \texttt{[]]}.$$

To illustrate the use of mapTree, we now implement a functionsucTree, that increases all numbers in a tree of integers by 1. It can easily and elegantly be defined using our new function mapTree.

```
sucTree :: Tree Int -> Tree Int
sucTree = mapTree suc
```

In general, map functions apply a function g to each subvalue of a object from a given data structure.

The zipWith function

Let us now consider two further functions addlist and multlist, which combine the elements of two lists by addition and multiplication, respectively.

```
addlist :: Num a => [a] -> [a] -> [a]
addlist (x:xs) (y:ys) = (x + y) : addlist xs ys
addlist _      _      = []

multlist :: Num a => [a] -> [a] -> [a]
multlist (x:xs) (y:ys) = (x * y) : addlist xs ys
multlist _      _      = []
```

The implementation of these two functions is again very similar. Thus, we now introduce the zipWith function that implements this recursion pattern. This function is, of course, pre-defined in the HASKELL libraries and works quite similar to map, but applies a function to two argments – one from each list.

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys)   = f x y : zipWith f xs ys
zipWith _ _      _        = []
```

With this function we have reduced the task of combining two lists into one to defining the function that takes a pair of elements of the two lists and produces an element of the target list. The non-recursive definitions for addlist and multlist can now be written as follows..

```
addlist  = zipWith (+)
multlist = zipWith (*)
```

`filter` functions

Let us consider a function `dropEven` that removes all even numbers from a list of integers and a function `dropUpper` that removes all upper-case letters from a list of characters. We will make use of the pre-defined auxiliary functions `odd` and `isLower`.

```
dropEven :: [Int] -> [Int]
dropEven [] = []
dropEven (x:xs) | odd x     = x : dropEven xs
                | otherwise = dropEven xs

dropUpper :: [Char] -> [Char]
dropUpper [] = []
dropUpper (x:xs) | isLower x = x : dropUpper xs
                 | otherwise = dropUpper xs
```

For example, the evaluation of the expression `dropEven [1,2,3,4]` yields the result `[1,3]` and the evaluation of `dropUpper "HelloWorld"` yields the result `"elloorld"`.

We immediately see that the two functions `dropEven` and `dropUpper` are very similar, because both traverse a list and delete all elements, that do not satisfy a certain condition, i.e., a certain predicate. It seems obvious to implement these two functions by one function. To this end, we have to take the following steps.

- Abstraction of the data type of the list elements (`Int` and `Char`, respectively). This is only possbile in programming languages with (parametric) polymorphism.
- Abstraction from the predicate (i.e., the Boolean function), that is used to filer the Listelements (`odd` and `isLower`, respectively). This is only possible in programming languages, where functions can be treated as data objects.

The most general form of functions constructed in this way is thus the following (where `g` is the predicate used for filtering).

```
f :: [a] -> [a]
f [] = []
f (x:xs) | g x       = x : f xs
         | otherwise = f xs
```

As `g` is an arbitrary Boolean function of the type `a -> Bool`, we should add it as an argument to the function `f`. In this way, we obtain the function `filter` (where `f` as above now corresponds to `filter g`). This function is, of course, pre-defined in the HASKELL libraries.

```
filter :: (a -> Bool) -> [a] -> [a]
filter g [] = []
filter g (x:xs) | g x       = x : filter g xs
                | otherwise = filter g xs
```

The functions `dropEven` and `dropUpper` as defined above can now be defined non-recursively in the following way.

```
dropEven  = filter odd
dropUpper = filter isLower
```

Analogously, corresponding `filter` functions on self-defined data structures can be defined, e.g. on self-defined lists. In general, `filter` functionen delete all subvalues of a data object for which the predicate `g` is not satisfied.

`fold` functions

Let us consider a function `add`, that adds all numbers in a list of integers and a function `prod`, that multiplies all numbers in a list of integers. We first illustrate this using our self-defined data types for lists as this is somewhat easier to grasp.

```
plus :: Int -> Int -> Int
plus x y = x + y

times :: Int -> Int -> Int
times x y = x * y

add :: (List Int) -> Int
add Nil         = 0
add (Cons x xs) = plus x (add xs)

prod :: (List Int) -> Int
prod Nil         = 1
prod (Cons x xs) = times x (prod xs)
```

When we evaluate a call to `add` with the argument

$$\texttt{Cons}\,\texttt{x}_1\,(\texttt{Cons}\,\texttt{x}_2\,(\ldots\,(\texttt{Cons}\,\texttt{x}_{n-1}\,(\texttt{Cons}\,\texttt{x}_n\,\texttt{Nil}))\,\ldots))$$

we obtain the result

$$\texttt{plus}\,\texttt{x}_1\,(\texttt{plus}\,\texttt{x}_2\,(\ldots\,(\texttt{plus}\,\texttt{x}_{n-1}\,(\texttt{plus}\,\texttt{x}_n\,\texttt{0}))\,\ldots)).$$

Here, the data constructor `Cons` is replaced by the function `plus` and the data constructor `Nil` by the number `0`. Analogously, for a call to `prod` with the same argument we obtain the result

$$\texttt{times}\,\texttt{x}_1\,(\texttt{times}\,\texttt{x}_2\,(\ldots\,(\texttt{times}\,\texttt{x}_{n-1}\,(\texttt{times}\,\texttt{x}_n\,\texttt{1}))\,\ldots)).$$

Here, the data constructor `Cons` is replaced by the function `times` and the data constructor `Nil` by the number `1`.

Once more, our goal is to define a higher-order function that implements the recursion pattern of these two functions. We can then use this function to implement `add` and `prod` non-recursively.

In general, the data constructor `Cons` is replaced by a function `g` and the data constructor `Nil` is replaced by an initial value `e`. Thus, we need a function `f` such that when we call `f` with the argument

$$\texttt{Cons}\,\texttt{x}_1\,(\texttt{Cons}\,\texttt{x}_2\,(\ldots\,(\texttt{Cons}\,\texttt{x}_{n-1}\,(\texttt{Cons}\,\texttt{x}_n\,\texttt{Nil}))\,\ldots))$$

we obtain the following result.

$$\texttt{g}\,\texttt{x}_1\,(\texttt{g}\,\texttt{x}_2\,(\ldots\,(\texttt{g}\,\texttt{x}_{n-1}\,(\texttt{g}\,\texttt{x}_n\,\texttt{e}))\,\ldots))$$

Once more we have to abstract from the type of the list elements and from the functions `g` and `e` that are used to replace the data constructors. The general form for such functions is shown in the form of a definition for the function `f` as follows.

```
f :: (List a) -> b
f Nil          = e
f (Cons x xs) = g x (f xs)
```

Here, the initial value `e` has the type `b` of the result and the auxiliary function `g` must have the type `a -> b -> b`. As `e` and `g` are arbitrary functions, they should be additional arguments to the function. In this way, we obtain the function `fold` (where `f` as above corresponds to the function `fold g e`).

```
fold :: (a -> b -> b) -> b -> (List a) -> b
fold g e Nil          = e
fold g e (Cons x xs) = g x (fold g e xs)
```

Then, the result of

$$\texttt{fold g e}\,(\texttt{Cons}\,\texttt{x}_1\,(\texttt{Cons}\,\texttt{x}_2\,(\ldots\,(\texttt{Cons}\,\texttt{x}_{n-1}\,(\texttt{Cons}\,\texttt{x}_n\,\texttt{Nil}))\,\ldots)))$$

is exactly

$$\texttt{g}\,\texttt{x}_1\,(\texttt{g}\,\texttt{x}_2\,(\ldots\,(\texttt{g}\,\texttt{x}_{n-1}\,(\texttt{g}\,\texttt{x}_n\,\texttt{e}))\,\ldots)).$$

Now, we can define the two functions `add` and `prod` non-recursively.

```
add  = fold plus  0
prod = fold times 1
```

Another example is the function `conc` that takes a list of lists as argument and retrns the concatenation of all these lists. Here, we use the following implementation of the function `append`.

```
append :: List a -> List a -> List a
append Nil         ys  = ys
append (Cons x xs) ys  = Cons x (append xs ys)
```

When we call conc with the argument

$$\texttt{Cons}\,\texttt{l}_1\,(\texttt{Cons}\,\texttt{l}_2\,(\ldots\,(\texttt{Cons}\,\texttt{l}_{n-1}\,(\texttt{Cons}\,\texttt{l}_n\,\texttt{Nil}))\,\ldots))$$

we obtain the result

$$\texttt{append}\,\texttt{l}_1\,(\texttt{append}\,\texttt{l}_2\,(\ldots\,(\texttt{append}\,\texttt{l}_{n-1}\,(\texttt{append}\,\texttt{l}_n\,\texttt{Nil}))\,\ldots)).$$

The implementation of conc is possible using the recursion pattern implemented by fold.

```
conc :: List (List a) -> List a
conc = fold append Nil
```

Analogously to fold, for pre-defined lists there is a function pre-defined in the HASKELL libraries. It is called foldr and is defined as follows.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr g e []     = e
foldr g e (x:xs) = g x (foldr g e xs)
```

If we implement for example add, prod, and conc for pre-defined lists, then we just use foldr instead of fold. The function add is pre-defined as sum in the HASKELL libraries and conc is pre-defined as concat.

```
add :: [Int] -> Int
add  = foldr plus  0

prod :: [Int] -> Int
prod = foldr times 1

conc :: [[a]] -> [a]
conc = foldr (++) []
```

Let us finally also consider a fold function on the self-defined data structure of arbitrarily branching trees.

```
data Tree a = Node a [Tree a]
```

The following function replaces all occurrences of the data constructor Node by the function g.

```
foldTree :: (a -> [b] -> b) -> Tree a -> b
foldTree g (Node x ts) = g x (map (foldTree g) ts)
```

If the tree t is given as

$$\texttt{Node}\,\texttt{x}_1\,\texttt{[Node}\,\texttt{x}_2\,\texttt{[]]},$$

then fold g t yields the result

$$\texttt{g}\,\texttt{x}_1\,\texttt{[g}\,\texttt{x}_2\,\texttt{[]]}.$$

To illustrate the use of foldTree, we implement the function addTree, that add up all the numbers in the nodes of a tree. Here, we additionally need a function addtolist, where $\texttt{addtolist}\,\texttt{x}\,[\texttt{y}_1,\ldots,\texttt{y}_n] = \texttt{y}_1\texttt{+}(\texttt{y}_2\texttt{+}\ldots\texttt{+}(\texttt{y}_n\texttt{+x})\ldots)$.

```
addtolist :: Num a => a -> [a] -> a
addtolist = foldr (+)

addTree :: Num a => Tree a -> a
addTree = foldTree addtolist
```

We then obtain `addTree (Node 1 [Node 2 []])` = `addtolist 1 [addtolist 2 []]` = 3.

In general, a `fold` function replaces the constructors of a data structure by given function `g`, `e`, etc. Thus, fold is actually more general than for example map and filter. To see this, consider that we can actually implement `map` or `filter` using `foldr`.

```
map :: (a -> b) -> [a] -> [b]
map f = foldr (\x ys -> f x : ys) []

filter :: (a -> Bool) -> [a] -> [a]
filter g = foldr (\x ys -> if g x then x : ys else ys) []
```

3.3 Programming using Lazy Evaluation

The programming language HASKELL uses a non-strict evaluation strategy:

- In general, expressions are evaluated using the leftmost outermost strategy.
- Pre-defined arithmetical operators and comparison operators require that their arguments are fully evaluated.
- During pattern matching the arguments are only evaluated until we can decide if a pattern matches or not.

These items are illustrated by the following example .

```
infinity :: Int
infinity = infinity + 1

mult :: Int -> Int -> Int
mult 0     y = 0
mult (x+1) y = y + mult x y
```

The evaluation of `mult 0 infinity` terminates with the result `0`. In contrast, the evaluation of the expression `0 * infinity` lead to non-termination. Here we can clearly see the difference between the general non-strict evaluation (for `mult`) and the evaluation for pre-defined operators (such as `*`). Furthermore, we observe that even with out evaluating the argument `infinity`, we can determine that the patterns of the first defining equation of `mult` match the arguments `0` and `infinity`.

In HASKELL it is possible to define infinite data objects. To see this, let us consider the following algorithm.

```
from :: Num a => a -> [a]
from x = x : from (x+1)
```

The expression `from x` corresponds to the infinite list `[x, x+1, x+2, ...]`. We can also denote this list in HASKELL as `[x ..]`. Although the evaluation of the expression `from 5` does not terminate, such infinite lists can be quite useful for programming. In HASKELL, there is a pre-defined function `take` that returns a prefix of a list. In general, we have `take n` $[x_1,\ldots,x_n,x_{n+1},\ldots] = [x_1,\ldots,x_n]$.

```
take :: Int -> [a] -> [a]
take 0     _      = []
take _     []     = []
take (n+1)(x:xs) = x : take n xs
```

As arguments are only evaluated as much as needed during pattern matching, we get:

```
  take 2 (from 5)
= take 2 (5 : from 6)
= 5 : take 1 (from 6)
= 5 : take 1 (6 : from 7)
= 5 : 6 : take 0 (from 7)
= 5 : 6 : []
= [5,6].
```

Thus, the evaluation of `take 2 (from 5)` indeed terminates. Functions that are never defined when the evaluation of one of their arguments is undefined, are called *strict*. Examples for such functions are, of course, the arithmetic operators and the comparison operators. In constrast, the functions `mult` and `take` are *non-strict*. We also differentiate strictness for different arguments. For example, the functions `mult` and `take` are strict in their first argument, but non-strict in their second argument.

Programming with Infinite Data Objects

The general approach for programming with infinite data objects is as follows. We first generate a potentially infinite list of approximations for the solution(s). Then, we filter the specific solution(s) we want from that list.

As an example, consider programming the *sieve of Eratosthenes* for generating prime numbers. The algorithm works as follows.

1. Generate the list of all natural numbers beginning from 2.
2. Mark the first unmarked number in the list.
3. Remove all multiples of the marked number from the list.
4. Go back to Step 2.

Thus, we begin with the list `[2,3,4,...]`. If we illustrate marking by underlining, we now mark the so far unmarked number 2. Then, we have the list `[2,3,4,...]`. Now, we eliminate all multiples of this marked number, i.e., all multiples of 2. This lead to the list `[2,3,5,7,9,11,...]`. Now, we mark the next unmarked number in the list resulting in `[2,3,5,7,9,11,...]`. Next, we eliminate the multiples of this marked number, i.e., all multiples of 3, and obtain `[2,3,5,7,11,13,17,...]`. We see that in the end, only prime numbers will remain as elements of the list.

These natural language description of the algorithm can be transformed almost directly into a program using infinite data objects and non-strict evaluation. Here, we naturally have to work with infinite lists as we indeed build the list of all (infinitely many) prime numbers. If we are now only interested in the first 100 primes or all primes less than 42, the sieve only has to be process a finite prefix of the list.

The implementation of Step 1 is obvious, because `from 2` or `[2 ..]` exactly compute the infinite list of natural numbers beginning with 2. To eliminate multiples of a number `x` from a list `xs` we use the following function. It computes all elements `y` from `xs` that are not divisible by `x`. Here, `y` is divisble by `x`, if there is no remainder `` y `mod` x `` when we perform integer division.

```
drop_mult :: Int -> [Int] -> [Int]
drop_mult x xs = filter (\y -> mod y x /= 0) xs
```

Thus, the expression `drop_mult 2 [3 ..]` computes the infinite list `[3,5,7,9,11,...]`.

To repeatedly remove all multiples of the first unmarked numbers in the list, we use a function `dropall`. It first eliminates all multiples of the first element from the rest of the list. Afterwards, it calls itself recusive on that list. Now, all multiples of the second element are eliminated, etc.

```
dropall :: [Int] -> [Int]
dropall (x:xs) = x : dropall (drop_mult x xs)
```

The list `primes` of all prime numbers can then be computed as given by the following pattern declaration.

```
primes :: [Int]
primes = dropall [2 ..]
```

When wie evaluate `primes` we obtain the infinite list

```
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,...].
```

The list of the first 100 primes can be computed by evaluating the expression `take 100 primes`. Thus, we see that we can indeed compute with infinite data objects in a meaningful way. The reason is that during the computation we always only consider a finite part of these objects.

To compute the list of all primes that are less than 42, we cannot use the expression `filter (< 42) primes`. The evaluation of this expression does not terminate, because the condition `x < 42` has to be tested for all (infinitely many) elements `x` of `primes`. (The evaluator cannot know that we compute a monotonically increasing sequence.) Instead we should use the function `takeWhile` that is pre-defined in the HASKELL libraries.

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x:xs) | p x       = x : takeWhile p xs
                   | otherwise = []
```

The evaluation of the expression `takeWhile` stops as soon as the condition `p` does not hold for some element of the list. Consequently, the evaluation of the expression `takeWhile (< 42) primes` terminates and indeed returns the list `[2,3,5,7,11,13,17,19,23,29,31,37,41]`.

Cyclic Data Objects

To increase the efficiency when computing with infinite data objects, it is a good idea to represent such data objects (if possible) in a cyclic fashion. The simplest example is the infinite list of ones computed by evaluating `ones`, i.e., the the list `[1,1,1,1,...]`.

```
ones :: [Int]
ones = 1 : ones
```

If a non-function variable such as `ones` occurs in its own definition, the constructed data object becomes cyclic as show in Figure 3.2. The advantage of such objects is in the low space consumption and the reuse of computations by sharing.

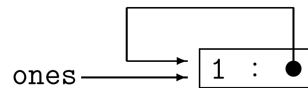


Figure 3.2: Zirkuläres Datenobjekt

To illustrate the gain in efficiency, we consider the Hamming problem (named after the famous mathematican W. R. Hamming), which can be solved very efficiently by using infinite (and cyclic) data objects. The task here is to generate a list with the following characteristics.

- The list is sorted ascendingly and there are no duplicates.
- The list begins with `1`.
- If the list contains the welement x, then it also contains the elements $2x$, $3x$, and $5x$.
- Except for these numbers, the list contains no further elements.

Thus, the list has the following form.

`[1,2,3,4,5,6,8,9,10,12,15,16,...]`

The Hamming problem is often used to analyse how adequate programming languages are for efficiently implementing certain classes of algorithms.[7]

The idea for an efficient implementation of this problem is to use a function `mer` that merges two (infinite) ordered lists into one ordered list without duplicates.

[7] These classes are “recursive stream computation”, “producer/consumer parallelism”, and “dynamic task creation”. The Hamming problem is a typical instance of a “closure problems”.

```
mer Ord a => [a] -> [a] -> [a]
mer (x : xs) (y: ys) | x < y     = x : mer xs (y:ys)
                     | x == y    = x : mer xs ys
                     | otherwise = y : mer (x:xs) ys
```

The infinite list `hamming` can now be defined as follows.

```
hamming :: [Int]
hamming = 1 : mer (map (2*) hamming)
                  (mer (map (3*) hamming)
                       (map (5*) hamming))
```

In the beginning, `hamming` is represented by a cyclic data obejct, where the three occurrences of `hamming` on the right-hand side are pointers to the whole expression. It is a good exercise to follow some steps of the evaluation of `hamming` on this cyclic data object. We observe that the computation of a new list element of `hamming` can be performed by at most three multiplications (to compute the first elements of `(map (2*) hamming)`, `(map (3*) hamming)`, and `(map (5*) hamming)`) and four comparisons (for `<` and `==` in the two calls to `mer`). The runtime complexity is thus *linear* in the number of elements of `hamming` computed. I.e., the complexity of the computation of `take n hamming` is $O(\texttt{n})$.

We clearly see that it is advantageous to define non-function variables that represent infinite structure (if possible) in a way that these variables occur on the right-hand sides of their definitions. As mentioned before, we create cyclic data objects in this way. Altogether, we see that infinite data structures can be very beneficial to both the efficiency and the conciseness of programs.

3.4 Input/Output using Monads

A very basic and important property of (pure) functional programming languages is that programs cannot have side effects. The value of an expression is always the same, i.e., it does not depend on the environment. As mentioned before, this concept is called *referential transparency*. But input and output are – by definition – side effects from and on the environment. If we had a function that reads a character from the keyboard and returns this character as the result, this would violate the concept of referential transparency. The evaluation of this function would not always return the same result, but the result would depend on the environment, i.e., the user that enters the character. The question is now how one can use input and output without violating referential transparency. Here, we will make use of monads which are a concept from category theory that allows keep referential transparency while still allowing for side effects.

More concretely, we will use the pre-defined data type `IO ()`, whose value are *actions*. The evaluation of an expression of type `IO ()` then means that this action is executed. The type `IO ()` is an abstract data type that hides the representation of its values from the user. Here, it is only important which operations are available for this type. For example, there is a pre-defined function `putChar` that outputs a character.

```
putChar ::  Char -> IO ()
```

The value of the expression `putChar '!'` is an action that outputs a `!`. Note that the output of a character does not contain the single quotes while its value does.

Further, there is pre-defined function `>>` (called "then") for combining actions.

```
(>>) ::  IO () -> IO () -> IO ()
```

The value of the expression `x >> y` is an action where first the action `x` is executed and then the action `y` is executed. If we enter the expression

```
putChar 'a' >> putChar 'b'
```

into the interpreter, then this expression is evaluated and the effect is that `ab` is printed to the console.

To create an empty action, we can use the function `return`.

```
return () ::  IO ()
```

The value of the expression `return ()` is the empty action, i.e., when evaluating the expression `putChar '!'  >> return ()` we also just print `!` to the console.

We can, of course, also define recursive algorithms using the data type `IO ()`. Let us consider the (pre-defined) declaration of a function that prints a `String` to the console.

```
putStr :: String -> IO ()
putStr []     = return ()
putStr (x:xs) = putChar x >> putStr xs
```

We can alternatively use `foldr` and `map` to define `putStr` as follows:

```
putStr = \xs -> foldr (>>) (return ()) (map putChar xs)
```

The data type `IO ()` contains only output actions. We need to generalize it for input actions. To this end we use a pre-defined type `IO a` of actions that compute a value of type `a`. There is for example a (pre-defined) function for reading a character.

```
getChar ::  IO Char
```

The value of the expression `getChar` is an action that reads a character from standard input (i.e., from the keyboard). The value of `getChar` is *not the character read* but the action "read a character". We can represent such actions graphically as follows.

This illustrates that input and output actions occur and that the action encapsulates a vale of type `a`.

The type `IO ()` is a special case of the type `IO a`. The type `()` contains only a single element (namely the empty tuple) `()`). Thus, we treat actions without input in such a way as if we read the fixed value `()`.

We also extend the function `return` to be able to generate empty actions of arbitrary `IO` types:

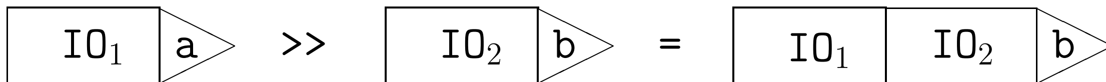
```
return :: a -> IO a
```

Then `return '!'` is the action of type `IO Char` that does nothing but encapsulate a character `'!'`.

The extension of the function `>>` for combining actions is an operator of the following type.

```
(>>) :: IO a -> IO b -> IO b
```

If `p` and `q` are actions, then `p >> q` is an action that first executes `p`, discards the determined value ausführt, and then executes action `q`. If `p` is of type `IO a` and `q` of type `IO b`, then we obtain the following graphical representation.

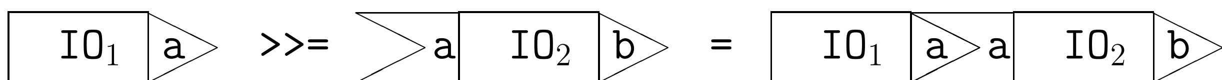


For example, if we evaluate the expression `getChar >> return ()`, first a character is read from the keyboard. Then the empty action is executed and the combined action is done. The type of the expression `getChar >> return ()` is `IO ()`. Expressions of this type can directly be evaluated in the interpreter by performing the associated actions. The value of the expression (which is this action) is not displayed (there is no `show` function for actions). In contrast, expressions of type `IO a` for `a` $\neq$ `()` can not be evaluated directly in the interpreter. If we only enter `getChar` into the interpreter, we will obtain an error message.

The combination `p >> q` of two actions is, of course, only meaningful if the envapsulated value determined by the action `p` is not of further interest, as the action `q` cannot depend on this value. As we have seen, the value determined by `p` (of tyoe `a`) is simply ignored in the action we obtain by combining `p` and `q` using `>>`. If we want to execute two actions after each other and want the second action to depend on the value determined by the first action, we have to use another pre-defined operator `>>=` (called "bind").

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

When we evaluate the expression `p >>= f`, first the action `p` is executed. Here, we determine a value `x` of type `a`. Then, `f x` is evaluated which eventually determines an encapsulated value of type `b`. Graphically we can represent this combination as follows.



We observe that the "then" operator can be defined using the "bind" operator in the following way.

```
p >> q   =   p >>= \_ -> q
```

Now we can define a function `echo` such that the value of the expression `echo` is an action that first reads a character from the keyboard and then prints this character to the console.

```
echo :: IO ()
echo = getChar >>= putChar
```

Here, the concept of referential transparence is not violated. Thus, the value of the two expressions `echo >> echo` and `let x = echo in x >> x` is identical. The value is the action that first reads a character and prints it and then reads another (possibly different) character and again prints this.

Further pre-defined primitive functions for input and output are the following two.

```
readFile  :: String -> IO String
writeFile :: String -> String -> IO ()
```

Here, the argument of `readFile` is the name of the file and the value of `readFile "myfile"` is an action, that opens the file `myfile` and determines its contents, i.e., the `String` of its contents is encapsulated in the returned action of type `IO String`. The action `writeFile "myfile" "hello"` overwrites the content of the file `myfile` with the world `hello`.

Let us now consider a function `gets` for reading a given number of characters, i.e., `gets n` returns an action that reads `n` characters and encapsulates the resulting object of type `String`.

```
gets :: Int -> IO String
gets 0     = return []
gets (n+1) = getChar >>= \x  ->
                         gets n  >>= \xs ->
                                     return (x:xs)
```

If we only want to read `0` characters, the empty action with the empty string is executed. Otherwise, we first execute the action `getChar` that reads a character `x`. Then this character is bound to the variable `x` and the action `gets n` is executed. This actions reads a string `xs` of length `n`. Finally, the resulting string is bound to the variable `xs` and the empty action is executed with the (encapsulated) result `(x:xs)`.

This form of sequential execution of actions using `>>=`, where the result of the last action is bound to variables using lambda expressions, occurs quite often. As the above notation using `>>=` and lambda expressions is hard to read, we introduce a new notation for this kind of sequencing. Instead of

```
p >>= \x ->
      q
```

we can now simply write the following.

```
do { x <- p;
     q
   }
```

This means "Set the variable `x` to the result determined by the action `p` and then execute the action `q`." We can also make use of the offside rule when using the `do` notation such that we can write the following instead.

```
do x <- p
   q
```

Analogously, we can replace the longer sequence

```
p >>= \x ->
      q >>= \y ->
            r
```

by the following.

```
do x <- p
   y <- q
   r
```

In general, we use the following translation rules that define the semantics of the `do` notation. Here, `S` is a non-empty sequence of expressions of the form `x <- p` or `r`.

```
do {x <- p; S}  =  p >>= \x -> do {S}
do {p; S}       =  p >> do {S}
do {p}          =  p
```

Using this simplified notation we can now write the function `gets` for reading a string of given length as follows.

```
gets :: Int -> IO String
gets 0     = return []
gets (n+1) = do x  <- getChar
                xs <- gets n
                return (x:xs)
```

As another example, consider a function `doubleChar` that reads a character and returns a string consisting of two copies of this character.

```
doubleChar = do x <- getChar
                putString [x,x]
```

In this way, we can use an imperative style of programming in functional programs. The `do` notation reminds us of sequences of assignments in imperative programs. The advantage of this input output framework is that the sequentiality of the program is ensured, i.e., `getChar` is executed before `gets n` and this is executed before `return (x:xs)`. Such sequentiality is very important as the order of actions should not depend on the evaluation strategy of the functional language. An example for this would be a program that first has to read a certain character before it may continue. Thus, for input and output a sequential programming style with such imperative properties is adequate. But it is important that (in contrast to real imperative programs), referential transparency is not violated. The value of an expression is always the same, i.e., it does not depend on the environment. As mentioned before, the value of the expression `getChar` is an action that reads a character, not the character itself. (The character is encapsulated in the action.)

The type `IO a` allows for a strict separation of program parts dealing with input and output (and thus with side effects) from the purely functional program parts. Side effects only happen during `IO` actions, but they never influence the value of an expression. Thus, there cannot be any function of type `IO a -> a` that extracts the value encapsulated in an action.[8] The encapsulated value can only be used in later actions. This ensures that only expressions of type `IO a` can cause side effects while other expressions can never have side effects.

The reason for disallowing functions that extract encapsulated values from actions is that we could then violate referential transparency. If we had a function `result :: IO a -> a` that has access to the encapsulated value, the value of the expression `result getChar` would be the character read. Now, this would violate referential transparency, because if we evaluate `result getChar` more than once, we could have different results. In each evaluation, the value would depend on the character entered by the user. For example, the variable `x` in the expression

```
let x = getChar in x >> x
```

denotes the same action (`getChar`) and `x >> x` is the action "Execute `getChar` two times". `IO` actions are similar to functions, i.e., every occurrence denotes the same function. But if we consider the expression

```
let x = result getChar in ...  x ...  x ...
```

the variable `x` would denote a certain value that would be the same in different positions of the expression `...  x ...  x ...`. The value of this expression would, consequently, be different from the value of the expression

```
...  result getChar ...  result getChar ....
```

This is the reason for disallowing functions like `result`.

Functions for extracting encapsulated values are only disallowed for the `IO` monad. There are other monads (pre-defined and self-defined) where this functionality is naturally available and meaningful. The reason for this special restriction of the pre-defined `IO` monad is that only here side effects may happen and, thus, only here this restriction is needed to avoid violation of referential transparency. Objects of type `IO a` can only be handled by pre-defined functions such as `return`, `>>`, and `>>=`. These functions are also available for other monads, but there we can define further functions – for example for directly accessing the encapsulated values.

A further difference between input and ouput in imperative languages and the monadic input output framework used in HASKELL is that actions are now regular values. Thus, they can be passed as arguments to functions, stored in data structures (e.g. lists of actions), etc.

[8]It is, of course, possible to have different functions of the type `IO a -> a` – for example the function that maps every action to the number `5`.