COMP90048

Declarative Programming

Subject Notes for Semester 1, 2018

```
search_bst :: Tree k v -> k -> Maybe v
search_bst Leaf _ = Nothing
search_bst (Node k v l r) sk =
    if sk == k then
        Just v
    else if sk < k then
        search_bst l sk
    else
        search_bst r sk
```

School of Computing and Information Systems

The University of Melbourne

A language that doesn't affect the way you think about programming, is not worth knowing.

— Alan Perlis

Section 0: Subject Introduction

Welcome to Declarative Programming
(COMP30020/COMP90048)

Lecturer:

- Peter Stuckey (`pstuckey@unimelb.edu.au`, Doug McDonell room 6.19)
- Rob Moss (`rgmoss@unimelb.edu.au`, Office 335, Melbourne School of Population and Global Health, 207 Bouverie St)

There will be two lectures per week, with one lecture given to the mid-semester test, for a total of 23. You are welcome to ask questions during or after a lecture.

There will be eleven one-hour workshops (tutorials), starting in week 2.

You should have already been allocated a workshop. Please check your personal timetable after the lecture.

- Please turn mobile phones OFF during lectures and workshops.
- Please do not use your laptop for anything distracting to others.

NOTE That includes playing games, checking Facebook, displaying still or moving pictures (even if done by screensavers), and anything that generates noise, including loud keyclicks.

Workshops

The workshops will reinforce the material from lectures, partly by asking you to apply it to small scale programming tasks.

To get the most out of each workshop, you should read and attempt the exercises *before* your workshop.

Sample solutions for each set of workshop exercises are available on the LMS, but I strongly recommend you *not* look at them until you have completed them yourself.

Most programming questions have more than one correct answer; your answer may be correct even if it differs from the sample solution.

NOTE If your laptop can access the building's wireless network, you will be able to log onto the MSE servers and use their installed versions of this subject's languages, Haskell and Prolog. If your laptop cannot access the building's wireless network, then you will be able to test your Haskell or Prolog code if you install the implementations of those languages on your machine yourself. For both languages this is typically fast and simple.

Textbooks

The lecture notes contain copies of the slides presented in lectures, plus some additional material.

All subject materials (lecture notes, workshop exercises, project specifications etc) will also be available online through the LMS.

The recommended text is

- Bryan O'Sullivan, John Goerzen and Don Stewart: Real world Haskell. O'Reilly Media,

2009. ISBN 978-0-596-51498-3. Available online at http://book.realworldhaskell.org/read.

Other recommended resources are listed on the LMS.

Assessment

- mid semester test (10%)
- programming projects (40%)
- two-hour written exam (50%)

To pass the subject (get 50%), you must pass the project component and the combined test and exam component.

There will be four projects, the first, a short Haskell project, is due after week 3, the second due in week 7 or 8, the third, a short Prolog project, is due in week 10, and the fourth due in week 11 or 12.

The exam is closed book, and will be held during the usual examination period after the end of the semester. Practice versions of both the mid-semester test and the final exam are available on the LMS.

How to succeed

Declarative programming is substantially different from imperative programming.

Even after you can understand declarative code, it can take a while before you can master writing your own.

If you have been writing imperative code all your programming life, you will probably try to write imperative code even in a declarative language. This often does not work, and when it does work, it usually does not work *well*.

Writing declarative code requires a different mindset, which takes a while to acquire.

This is why attending the workshops, and practicing, practicing and practicing some more are *essential* for passing the subject.

Sources of help

During contact hours:

- Ask me during or after a lecture (not before).
- Ask the demonstrator in your workshop.

Outside contact hours:

- The LMS discussion board (preferred: everyone can see the discussion)
- Email (if not of interest to everyone)
- Attend my consultation hours (see LMS for schedule)
- Email to schedule an appointment

Announcements will be made both in lectures and on the LMS.

Please do not pose a question *both* on the LMS and in private email; if it is on the LMS, I *will* see it.

Do you want to represent this class?

Would you like to represent this class on the department's student-staff liaison committee?

The job is to collect the concerns of your classmates and present them to me and to the committee. I will post your name and email address on the LMS so classmates can contact you.

The bribe is that you get pizza, and something to put on your resumé, and the knowledge you've done something to help both your classmates and the university, and possibly make the subject better for yourself.

If you would like to volunteer, please email me.

Objectives

On completion of this subject, students should be able to:

- apply declarative programming techniques;
- write medium size programs in a declarative language;
- write programs in which different components use different languages;
- select appropriate languages for each component task in a project.

These objectives are not all of equal weight; we will spend almost all of our time on the first two objectives.

Content

- introduction to functional programming and Haskell
- declarative programming techniques
- introduction to logic programming and Prolog
- tools for declarative programming, such as debuggers
- interfacing to imperative language code
- introduction to constraint programming

This subject will teach you Haskell and Prolog, with an emphasis on Haskell.

Why Declarative Programming

Declarative programming languages are quite different from imperative and object oriented languages.

- They give you a different perspective: a focus on *what* is to be done, rather than *how*.
- They work at a higher level of abstraction.
- They make it easier to use some powerful programming techniques.
- Their clean semantics means you can do things with declarative programs that you can't do with conventional programs.

The ultimate objective of this subject is to widen your horizons and thus to make you better programmers, and not just when using declarative programming languages.

Imperative vs functional

Imperative languages are based on *commands*, in the form of *instructions* and *statements*.

- Commands are executed.
- Commands have an effect.

Assignment statements update *state*, and later code may depend on this update. Example: `x := 3 + 4 ; ...`

Functional languages are based on *expressions*.

- Expressions are evaluated.
- Expressions have no effect.

Side effects

Code is said to have a *side effect* if, in addition to producing a value, it also modifies some state or has an observable interaction with calling functions or the outside world. For example, a function might

- modify a global or a static variable,
- modify one of its arguments,
- raise an exception (e.g. divide by zero),
- write data to a display, file or network,
- read data from a keyboard, mouse, file or network, or
- call other side-effecting functions.

NOTE Reading from a file is a side effect because it moves the current position in the file being read, so that the next read from that file will get something else.

An example: destructive update

In imperative languages, the natural way to insert a new entry into a table is to modify the table in place: a side-effect. This effectively destroys the old table.

In declarative languages, you would instead create a new version of the table, but the old version (without the new entry) would still be there.

The price is that the language implementation has to work harder to recover memory and to ensure efficiency.

The benefit is that you don't need to worry what other code will be affected by the change. It also allows you to keep previous versions, for purposes of comparison, or for implementing undo.

The *immutability of data structures* also makes parallel programming *much* easier. Some people think that programming the dozens of cores that CPUs will have in future is the killer application of declarative programming languages.

Guarantees

- If you pass a pointer to a data structure to a function, can you guarantee that the function does not update the data structure?
  If not, you will need to make a copy of the data structure, and pass a pointer to *that*.
- You add a new field to a structure. Can you guarantee that every piece of code that handles the structure has been updated to handle the new field?
  If not, you will need many more test cases, and will need to find and fix more bugs.
- Can you guarantee that this function does not read or write global variables? Can you guarantee that this function does no I/O?
  If the answer to either question is "no", you

will have much more work to do during testing and debugging, and parallelising the program will be a *lot* harder.

Some uses of declarative languages

- In a US Navy study in which several teams wrote code for the same task in several languages, declarative languages like Haskell were much more productive than imperative languages.
- Mission Critical used Mercury to build an insurance application in one third the time and cost of the next best quote (which used Java).
- Ericsson, one of the largest manufacturers of phone network switches, uses Erlang in some of their switches.
- The statistical machine learning algorithms behind Bing's advertising system are written in F#.
- Facebook used Haskell to build the system they use to fight spam. Haskell allowed them to increase power and performance over their previous system.

NOTE Erlang, F# and Lisp are of course declarative languages.

For a whole bunch of essays about programming, including some about the use of Lisp in Yahoo! Store, see paulgraham.com.

The Blub paradox

Consider Blub, a hypothetical average programming language right in the middle of the power continuum.

When a Blub programmer looks down the power continuum, he knows he is looking down. Languages below Blub are obviously less powerful, because they are missing some features he is used to.

But when a Blub programmer looks up the power continuum, he does not realize he is looking up. What he sees are merely weird languages. He thinks they are about equivalent in power to Blub, but with some extra hairy stuff. Blub is good enough for him, since he thinks in Blub.

When we switch to the point of view of a programmer using a language higher up the power continuum, however, we find that she in turn looks down upon Blub, because it is missing some things *she* is used to.

Therefore understanding the differences in power between languages requires understanding the most powerful ones.

NOTE This slide is itself paraphrased from one of Paul Graham's essays. (The full quotation is too big to fit on one slide.)

The least abstract and therefore least powerful language is machine code. One step above that is assembler, and one step above that are the lowest level imperative languages, like C and Fortran. Everyone agrees on that. Most people (but not all) would also agree that modern object-oriented languages like Java and C#, scripting languages like awk and Perl and multi-paradigm languages like Python and Ruby are more abstract and more powerful than C and Fortran, but there is no general agreement on their *relative* position on the continuum. However, almost everyone who knows declarative programming languages agrees that *they* are more abstract and more powerful than Java, C#, awk, Perl, Python and Ruby.

A large part of that extra power is the ability to

offer many more guarantees.

Section 1: Introduction to Functional Programming

Functional programming

The basis of functional programming is *equational reasoning*. This is a grand name for a simple idea:

- if two expressions have equal values, then one can be replaced by the other.

You can use equational reasoning to rewrite a complex expression to be simpler and simpler, until it is as simple as possible. Suppose `x = 2` and `y = 4`, and you start with the expression `x + (3 * y)`:

```
step 0:   x + (3 * y)
step 1:   2 + (3 * y)
step 2:   2 + (3 * 4)
step 3:   2 + 12
step 4:   14
```


Lists

Of course, programs want to manipulate more complex data than just simple numbers.

Like most functional programming languages, Haskell allows programmers to define their own types, using a much more expressive type system than the type system of e.g. C.

Nevertheless, the most frequently used type in Haskell programs is probably the builtin *list* type.

The notation `[]` means the empty list, while `x:xs` means a nonempty list whose *head* (first element) is represented by the variable `x`, and whose *tail* (all the remaining elements) is represented by the variable `xs`.

The notation `["a", "b"]` is syntactic sugar for `"a":"b":[]`. As in most languages, `"a"` represents the string that consists of a single character, the first character of the alphabet.

– section 1 slide 2 –

Functions

A function definition consists of equations, each of which establishes an equality between the left and right hand sides of the equal sign.

```
len []     = 0
len (x:xs) = 1 + len xs
```

Each equation typically expects the input arguments to conform to a given *pattern*; `[]` and `(x:xs)` are two patterns.

The set of patterns should be *exhaustive*: at least one pattern should apply for any possible call.

It is good programming style to ensure that the set of patterns is also *exclusive*, which means that *at most one* pattern should apply for any possible call.

If the set of patterns is both exhaustive and exclusive, then *exactly one* pattern will apply for any possible call.

NOTE `ghc`, the most popular Haskell compiler, has options you can specify if you want some help following these rules of programming style. If given the option `-fwarn-incomplete-patterns`, `ghc` will give you a warning if the patterns are not exhaustive; if given the option `-fwarn-overlapping-patterns`, `ghc` will give you a warning if the patterns are not exclusive.

If the set of patterns for a function is not exhaustive, and no equation applies for a given function call, then the function call will generate an exception, which will typically cause the program to be aborted.

– section 1 slide 3 –

Aside: syntax

- In most languages, a function call looks like `f(fa1, fa2, fa3)`.
- In Haskell, it looks like `f fa1 fa2 fa3`.

If the second argument is not `fa2` but the function `g` applied to the single argument `ga1`, then in Haskell you would need to write `f fa1 (g ga1) fa3`, since Haskell would interpret `f fa1 g ga1 fa3` as a call to `f` with four arguments.

In Haskell, there are no parentheses around the whole argument list of a function call, but parentheses may be needed around *individual* arguments. This applies on the left as well the right hand sides of equations.

This is why the recursive call is `len xs` and not `len(xs)`, and why the left hand side of the second equation is `len (x:xs)` instead of `len x:xs`.

– section 1 slide 4 –

More syntax issues

Comments start with two minus signs and continue to the end of the line.

The names of functions and variables are sequences of letters, numbers and/or underscores that must start with a lower case letter.

Suppose line1 starts in column n, and the following nonblank line, line2, starts in column m. The *offside rule* says that

- if $m > n$, then line2 is a continuation of the construct on line1;
- if $m = n$, then line2 is the start of a new construct at the same level as line1;
- if $m < n$, then line2 is either the continuation of something else that line1 is part of, or a new item at the same level as something else that line1 is part of.

This means that the structure of the code as shown by indentation must match the structure of the code.

NOTE Actually, it is also ok for function names to consist wholly of graphic characters like `+`, but only builtin functions should have such names.

If your source code includes tabs as well as spaces, two lines can *look* like they have the same level of indentation even if they do not. It is therefore best to ensure that Haskell source files do not contain tabs.

If you use `vim` as your editor, you can ensure this by putting

```
-- vim: ts=4 sw=4 expandtab syntax=haskell
```

at the top of the source file. The `expandtab` tells `vim` to expand all tabs into several spaces, while the `ts=4` and `sw=4` tell `vim` to set up tab stops every four columns. The `syntax=haskell` specifies the set of rules `vim` should use for syntax highlighting.

The following function definition is *wrong*: since the second line is indented further than the first, Haskell considers it to be a continuation of the first equation, rather than a separate equation.

```
len []     = 0
len (x:xs) = 1 + len xs
```

The following definition is acceptable to the offside rule, though it is *not* an example of good style:

```
len [] =
    0
len (x:xs) =
    1 +
    len xs
```

Recursion

The definition of a function to compute the length of a list, like many Haskell functions, reflect the structure of the data: a list is either empty, or has a head and a tail.

The first equation for `len` handles the empty list case.

```
len []     = 0
```

This is called the *base case*.

The second equation handles nonempty lists. This is called the *recursive case*, since it contains a recursive call.

```
len (x:xs) =  1 + len xs
```

If you want to be a good programmer in a declarative language, you have to get comfortable with recursion, because most of the things you need to do involve recursion.

Using a function

```
len []     = 0
len (x:xs) = 1 + len xs
```

Given a function definition like this, the Haskell implementation can use it to replace calls to the function with the right hand side of an applicable equation.

```
step 0:  len ["a", "b"]  -- ("a":("b":[]))
step 1:  1 + len ["b"]   -- ("b":[])
step 2:  1 + 1 + len []
step 3:  1 + 1 + 0
step 4:  1 + 1
step 5:  2
```

NOTE In general, if there is more than one applicable equation, the Haskell implementation picks the first one.

You can think of builtin functions as being implicitly defined by a *very* long list of equations; on a 32 bit machine, you would need $2^{32} * 2^{32} = 2^{64}$ equations. In practice, such functions are of course implemented using the arithmetic instructions of the platform.

Expression evaluation

To evaluate an expression, the Haskell runtime system conceptually executes a loop, each iteration of which consists of these steps:

- looks for a function call in the current expression,
- searches the list of equations defining the function from the top downwards, looking for a matching equation,
- sets the values of the variables in the matching pattern to the corresponding parts of the actual arguments, and
- replaces the left hand side of the equation with the right hand side.

The loop stops when the current expression contains no function calls, not even calls to such builtin "functions" as addition.

The actual Haskell implementation is more sophisticated than this loop, but the effect it achieves is the same.

Order of evaluation

The first step in each iteration of the loop, "look for a function call in the current expression", can find more than one function call. Which one should we select?

```
len []     = 0
len (x:xs) = 1 + len xs
```

```
evaluation order A:
step 0:  len ["a"] + len []
step 1:  1 + len [] + len []
step 2:  1 + 0 + len []
step 3:  1 + len []
step 4:  1 + 0
step 5:  1

evaluation order B:
step 0:  len ["a"] + len []
step 1:  len ["a"] + 0
step 2:  1 + len [] + 0
step 3:  1 + 0 + 0
step 4:  1 + 0
step 5:  1
```

NOTE In this example, evaluation order A chooses the leftmost call, while evaluation order B chooses the rightmost call.

Church-Rosser theorem

In 1936, Alonzo Church and J. Barkley Rosser proved a famous theorem, which says that for the rewriting system known as the *lambda calculus*, regardless of the order in which the original term's

subterms are rewritten, the final result is always the same.

This theorem also holds for Haskell and for several other functional programming languages (though not for all).

This is not that surprising, since most modern functional languages are based on one variant or another of the lambda calculus.

We will ignore the order of evaluation of Haskell expressions for now, since in most cases it does not matter. We will come back to the topic later.

The Church-Rosser theorem is *not* applicable to imperative languages.

NOTE Each rewriting step replaces the left hand side of an equation with the right hand side.

Order of evaluation: efficiency

```
all_pos [] =  True
all_pos (x:xs) =  x > 0  && all_pos xs
```

```
evaluation order A:
0: all_pos [-1, 2]
1: -1 > 0 && all_pos [2]
2: False && all_pos [2]
3: False

evaluation order B:
0: all_pos [-1, 2]
1: -1 > 0 && all_pos [2]
2: -1 > 0 && 2 > 0 && all_pos []
3: -1 > 0 && 2 > 0 && True
4: -1 > 0 && True && True
5: -1 > 0 && True
6: False && True
7: False
```

NOTE The definition of conjunction for booleans does not need to know the value of second operand if the value of the first operand is False:

```
False & _ = False
True  & b = b
```

Note that `all_pos`, like `len`, has one equation for empty lists and one equation for nonempty lists. Functions that operate on data with the similar structures often have similar structures themselves.

Imperative vs declarative languages

In the presence of side effects, a program's behavior depends on history; that is, the order of evaluation matters.

Because understanding an effectful program requires thinking about all possible histories, side effects often make a program harder to understand.

When developing larger programs or working in teams, managing side-effects is critical and difficult; Haskell guarantees the absence of side-effects.

What really distinguishes pure declarative languages from imperative languages is that they do not allow side effects.

There is only one benign exception to that: they do allow programs to generate exceptions.

We will ignore exceptions from now on, since in the programs we deal with, they have only one effect: they abort the program.

NOTE This is OK. The only thing that depends on the order of evaluation is which of several exceptions that a program can raise will actually be raised. Unless you are writing the exception handler, you still don't have to understand all possible histories.

Referential transparency

The absence of side effects allows pure functional languages to achieve *referential transparency*, which means that an expression can be replaced with its value. This requires that the expression has no side effects and is pure, i.e. always returns the same results on the same input.

By contrast, in imperative languages such as C, functions in general are not pure and are thus not functions in a mathematical sense: two identical calls may return different results.

Impure functional languages such as Lisp are called impure precisely because they *do* permit side effects like assignments, and thus their programs are not referentially transparent.

NOTE Impure functional languages share characteristics both with imperative languages and with pure functional languages, so they are effectively somewhere between them on the spectrum of programming languages.

In the rest of the subject, we will use "functional languages" as a shorthand to mean "pure functional languages", except in contexts where we are specifically talking about impure functional languages.

Even in imperative languages, *some* functions are pure. This is usually true e.g. of implementations of actual mathematical functions, such as logarithm, sine, cosine, etc, but also of many others.

Single assignment

One consequence of the absence of side effects is that assignment means something different in a functional language than in an imperative language.

- In conventional, imperative languages, even object-oriented ones (including C, Java, and Python), each variable has a current value (a garbage value if not yet initialized), and assignment statements can change the current value of a variable.
- In functional languages, variables are *single assignment*, and there are no assignment statements. You can define a variable's value, but you cannot redefine it. Once a variable has a value, it has that value until the end of its lifetime.

Giving variables values

Haskell programs can give a variable a value in one of two ways.

The explicit way is to use a *let clause*:

```
let pi = 3.14159 in ...
```

This defines `pi` to be the given value in the expression represented by the dots. It does *not* define `pi` anywhere else.

The implicit way is to put the variable in a pattern on the left hand side of an equation:

```
len (x:xs) = 1 + len xs
```

If `len` is called with a nonempty list, Haskell will bind `x` to its head and `xs` to its tail.

NOTE Actually, there are other ways, both explicit and implicit, but these are enough for now.

Section 2:
Builtin Haskell types

The Haskell type system

Haskell has a *strong*, *safe* and *static* type system.

The *strong* part means that the system has no loopholes; one cannot tell Haskell to e.g. consider an integer to be a pointer, as one can in C with `(char *) 42`.

The *safe* part means that a running program is guaranteed never to crash due to a type error. (A C program that dereferenced the above pointer would almost certainly crash.)

The *static* part means that types are checked when the program is compiled, not when the program is run.

This is partly what makes the *safe* part possible; Haskell will not even start to run a program with a type error.

NOTE There is an "out" from the static nature of the type system, for use in cases where this is warranted Haskell also supports a *dynamic* type, and operations on values of this type are checked for type correctness only at runtime.

Note also that different people mean different things when they talk about e.g. the strength or safety of a type system, so these are not the only definitions you will see in the computer science literature. However, we will use these definitions in this subject.

Basic Haskell types

Haskell has the usual basic types. These include:

- The boolean type is called `Bool`. It has two values: `True` and `False`.
- The native integer type is called `Int`. Values of this type are 32 or 64 bits in size, depending on the platform. Haskell also has a type for integers of unbounded size: `Integer`.
- The usual floating-point type is `Double`. (`Float` is also available, but its use is discouraged.)
- The character type is called `Char`.

There are also others, e.g. integer types with 8, 16, 32 and 64 bits regardless of platform.

There are more complex types as well.

The types of lists

In Haskell, list is not a type; it is a *type constructor*.

Given any type `t`, it constructs a type for lists whose elements are all of type `t`. This type is written as `[t]`, and it is pronounced as "list of `t`".

You can have lists of any type. For example,

- `[Bool]` is the type of lists of booleans,
- `[Int]` is the type of lists of native integers,
- `[[Int]]` is the type of lists of lists of native integers.

These are similar to `ArrayList<Boolean>`, `ArrayList<Integer>`, and `ArrayList<ArrayList<Boolean>>` in Java.

Haskell considers strings to be lists of characters, whose type is `[Char]`; `String` is a synonym for `[Char]`.

The names of types and type constructors should be identifiers starting with an upper case letter; the list type constructor is an exception.

NOTE In fact, you can have lists of lists of lists of lists `[[[[Int]]]]`, lists of lists of lists of lists of lists `[[[[[Int]]]]]`, and so on. The only effective limit is the programmer's ability to do something useful with the values of the type.

---

ghci

The usual implementation of Haskell is `ghc`, the Glasgow Haskell Compiler. It also comes with an interpreter, `ghci`.

```
$ ghci
...
Prelude> let x = 2
Prelude> let y = 4
Prelude> x + (3 * y)
14
```

The *prelude* is Haskell's standard library.

`ghci` uses its name as the prompt to remind users that they can call its functions.

NOTE Once you invoke `ghci`,

- you type an expression on a line;
- it typechecks the expression;
- it evaluates the expression (if it is type correct);
- it prints the resulting value.

You can also load Haskell code into `ghci` with `:load filename.hs`. The suffix `.hs`, the standard suffix for Haskell source files, can be omitted.

Types and `ghci`

You can ask `ghci` to tell you the type of an expression by prefacing that expression with `:t`. The command `:set +t` tells `ghci` to print the type as well as the value of every expression it evaluates.

```
Prelude> :t "abc"
"abc" :: [Char]
Prelude> :set +t
Prelude> "abc"
"abc"
...
it :: [Char]
```

The notation `x::y` says that expression `x` is of type `y`. In this case, it says `"abc"` is of type `[Char]`.

`it` is `ghci`'s name for the value of the expression just evaluated.

Function types

You can also ask `ghci` about the types of functions. Consider this function, which checks whether a list is empty:

```
isEmpty []    = True
isEmpty (_:_) = False
```

(`_` is a special pattern that matches anything.)

If you ask `ghci` about its type, you get

```
> :t isEmpty
isEmpty :: [t] -> Bool
```

A function type lists the types of all the arguments and the result, all separated by arrows. We'll see what the `t` means a bit later.

NOTE This function is already defined in standard Haskell; it is called `null`. The `len` function is also already defined, under the name `length`.

Function types

Programmers should declare the type of each function. The syntax for this is similar to the notation printed by `ghci`: the function name, a double colon, and the type.

```
module Emptiness where
\par
\vspace2mm
isEmpty :: [t] -> Bool
isEmpty [] = True
isEmpty _  = False
```

Declaring the type of functions is required only by good programming style. The Haskell implementation will infer the types of functions if not declared.

Haskell also infers the types of all the local variables.

Later in the subject, we will briefly introduce the algorithm Haskell uses for type inference.

NOTE A Haskell source file should contain one Haskell module. We will cover the Haskell module system later.

Function type declarations

With type declarations, Haskell will report an error and refuse to compile the file if the declared type of a function is incompatible with its definition.

It's also an error if a *call* to the function is incompatible with its declared type.

Without declarations, Haskell will report an error if the types in any call to any function are incompatible with its definition. Haskell will never allow code to be run with a type error.

Type declarations improve Haskell's error messages, and make function definitions much easier to understand.

Number types

Haskell has several numeric types, including `Int`, `Integer`, `Float`, and `Double`. A plain integer constant belongs to all of them. So what does Haskell say when asked what the type of e.g. `3` is?

```
Prelude> :t 3
3 :: (Num t) => t
Prelude> :t [1, 2]
[1, 2] :: (Num a) => [a]
```

In these messages, `a` and `t` are *type variables*; they are variables that stand for *types*, not *values*.

The notation `(Num t)` means "the type `t` is a member of type class `Num`". `Num` is the class of numeric types, including the four types above.

The notation `3 :: (Num t) => t` means that "if `t` is a numeric type, then `3` is a value of that type".

NOTE Similarly, the notation `[1, 2] :: (Num a) => [a]` means that "if `a` is a numeric type, then `[1, 2]` is a list of values of that type".

– section 2 slide 9 –

Number type flexibility

The usual arithmetic operations, such as addition, work for any numeric type:

```
Prelude> :t (+)
(+) :: (Num a) => a -> a -> a
```

The notation `a -> a -> a` denotes a function that takes two arguments and returns a result, all of which have to be of the same type (since they are denoted by the same type variable, `a`), which in this case must be a member of the `Num` type class.

This flexibility is nice, but it does result in confusing error messages:

```
Prelude> [1, True]
  No instance for (Num Bool)
    arising from the literal ‘1’
  Probable fix: add an
    instance declaration for (Num Bool)
```

NOTE The error message is trying to say that `Bool`, the type of `True`, is not a member or *instance* of `Num` type class. If it were a numeric type, then the list could a list of elements of that type, since the integer constant is a member of any numeric type, and thus the types of the two elements would be the same (which must hold for all Haskell lists).

The fix suggested by the error message is to have the programmer add a declaration to the effect that `Bool` *is* a numeric type. Since booleans are *not* numbers, this fix would be worse than useless, but since `ghc` has no idea about what each type actually means, it does not know that.

Programmers new to Haskell should probably handle type errors by simply going to the location specified in the error message (which has been removed from this example to make it fit on the slide), and looking around for the error, ignoring the rest of the error message.

Normally, + is a binary infix operator, but you can tell Haskell you want to use it as an ordinary function name by wrapping it in parentheses. (You can do the same with any other operator.) This means that `(+) 1 2` means the same as `1 + 2`.

– section 2 slide 10 –

---

if-then-else

```
-- Definition A
iota n =
    if n == 0 then [] else iota (n-1) ++ [n]
```

Definition A uses an if-then-else. If-then-else in Haskell differs from if-then-elses in imperative languages in that

- the else arm is not optional, and
- the then and else arms are expressions, not statements.

NOTE The expressions representing the then and else arms must be of the same type, since the result of the if-then-else will be one of them. It will be the expression in the then arm if the condition is true, and it will be the expression in the else arm if the condition is false.

In Haskell, `=` separates the left and right hand sides of equations, while `==` represents a test for equality.

A language called APL in the 1960s had a whole bunch of builtin functions working with numbers and vectors and matrices, and many of these were named after letters of the greek alphabet. This function is named after the iota function of APL, which also returned the first n integers when invoked with n as its argument.

– section 2 slide 11 –

Guards

```
-- Definition B
iota n
  | n == 0  = []
  | n  > 0  = iota (n-1) ++ [n]
```

Definition B uses *guards* to specify cases. Note the first line does not end with an "="; each guard line specifies a case and the value for that case, much as in definition A.

Note that the second guard specifies `n > 0`. What should happen if you do `iota (-3)`? What do you expect to happen? What about for definition A?

– section 2 slide 12 –

Structured definitions

Some Haskell functions do not fit on one line, and even the ones that do fit are often better split across several. Guards are only one example of this.

```
-- Definition C
iota n =
    if n == 0
    then
        []
    else
        iota (n-1) ++ [n]
```

The offside rule says that

- the keywords `then` and `else`, if they start a line, must be at the same level of indentation as the corresponding `if`, and
- if the `then` and `else` expressions are on their own lines, these must be *more* indented than those keywords.

– section 2 slide 13 –

Parametric polymorphism

Here is a version of the code of `len` complete with type declaration:

```
len :: [t] -> Int
\par
\vspace2mm
len []     = 0
len (_:xs) = 1 + len xs
```

This function, like many others in Haskell, is *polymorphic*. The phrase "poly morph" means "many shapes" or "many forms". In this context, it means that `len` can process lists of type `t` *regardless* of what type `t` is, i.e. regardless of what the form of the elements is.

The *reason* why `len` works regardless of the type of the list elements is that it does not do anything with the list elements.

This version of `len` shows this in the second pattern: the underscore is a placeholder for a value you want to ignore.

NOTE This is called *parametric* polymorphism because the type variable `t` is effectively a type *parameter*.

Since the underscore matches all values in its position, it is often called a wild card.

– section 2 slide 14 –

Section 3:
Defining Haskell types

Type definitions

Like most languages, Haskell allows programmers to define their own types. The simplest type definitions define types that are similar to enumerated types in C:

```
data Gender = Female | Male
data Role = Staff | Student
```

This defines two new types. The type called `Gender` has the two values `Female` and `Male`, while the type called `Role` has the two values `Staff` and `Student`.

Both types are also considered arity-0 type constructors; given zero argument types, they each construct a type.

The four values are also called *data constructors*. Given zero arguments, they each construct a value (a piece of data).

The names of type constructors and data constructors must be identifiers starting with upper-case letters.

NOTE "Arity" means the number of arguments. A function of arity 0 takes 0 arguments, a function of arity 1 takes 1 argument, a function of arity 2 takes 2 arguments, and so on. Similarly for type constructors. A type constructor of arity 0 constructs a type from 0 other types, a type constructor of arity 1 constructs a type from 1 other type, a type constructor of arity 2 constructs a type from 2 other types, and so on.

Using booleans

You do not have to use such types. If you wish, you can use the standard boolean type instead, like this:

```
show1 :: Bool -> Bool -> String
-- intended usage: show1 isFemale isStaff
show1 True True   = "female staff"
show1 True False  = "female student"
show1 False True  = "male staff"
show1 False False = "male student"
```

You can use such a function like this:

```
> let isFemale = True
> let isStaff = False
> show1 isFemale isStaff
```


Using defined types vs using booleans

```
> show1 isFemale isStaff
> show1 isStaff isFemale
```

The problem with using booleans is that of these two calls to `show1`, only one matches the programmer's intention, but since both are type correct (both supply two boolean arguments), Haskell cannot catch errors that switch the arguments.

```
show2 :: Gender -> Role -> String
```

With `show2`, Haskell can and *will* detect and report any accidental switch. This makes the program safer and the programmer more productive.

In general, you should use separate types for separate semantic distinctions. You can use this technique in any language that supports enumerated types.

– section 3 slide 3 –

Representing cards

Here is one way to represent standard western playing cards:

```
data Suit = Club | Diamond | Heart | Spade
data Rank
    = R2 | R3 | R4 | R5 | R6 | R7 | R8
    | R9 | R10 | Jack | Queen | King | Ace
\par
\vspace2mm
data Card = Card Suit Rank
```

The types `Suit` and `Rank` would be enumerated types in C, while the type `Card` would be a structure type.

On the right hand side of the definition of the type `Card`, `Card` is the name of the *data constructor*, while `Suit` and `Rank` are the types of its two arguments.

NOTE In general, each alternative starts with the name of the constructor, which is followed by the types of all the arguments (if there are any).

– section 3 slide 4 –

Creating structures

```
data Card = Card Suit Rank
```

In this definition, `Card` is not just the name of the type (from its first appearance), but (from its second appearance) also the name of the data constructor which constructs the "structure" from its arguments.

In languages like C, creating a structure and filling it in requires a call to `malloc` or its equivalent, a check of its return value, and an assignment to each field of the structure. This typically takes several lines of code.

In Haskell, you can construct a structure just by writing down the name of the data constructor, followed by its arguments, like this: `Card Club Ace`. This typically takes only *part* of one line of code.

In practice, this seemingly small difference has a significant impact, because it removes much clutter (details irrelevant to the main objective).

NOTE This is the reason for the name "data constructor".

– section 3 slide 5 –

Printing values

Many programs have code whose job it is to print out the values of a given type in a way that is meaningful to the programmer. Such functions are particularly useful during various forms of debugging.

The Haskell approach is use a function that returns a string. However, writing such functions by hand can be tedious, because each data constructor requires its own case:

```
showrank :: Rank -> String
showrank R2 = "R2"
showrank R3 = "R3"
...
```

– section 3 slide 6 –

Show

The Haskell prelude has a standard string conversion function called `show`. Just as the arithmetic functions are applicable to all types that are members of the type class `Num`, this function is applicable to all types that are members of the type class `Show`.

You can tell Haskell that the show function for values of the type `Rank` is `showrank`:

```
instance Show Rank where show = showrank
```

This of course requires defining `showrank`. If you don't want to do that, you can get Haskell to define the show function for a type by adding `deriving Show` to the type's definition, like this:

```
data Rank =
    R2 | R3 | R4 | R5 | R6 | R7 | R8 |
    R9 | R10 | Jack | Queen | King | Ace
    deriving Show
```

NOTE The offside rule applies to type definitions as well as function definitions, so e.g.

```
data Rank =
    R2 | R3 | R4 | R5 | R6 | R7 | R8 |
    R9 | R10 | Jack | Queen | King | Ace
deriving Show
```

would not be valid Haskell: since `deriving` is part of the definition of the type `Rank`, it must be indented *more* than the line that starts the type definition.

Eq and Ord

Another operation even more important than string conversion is comparison for equality.

To be able to use Haskell's `==` comparison operation for a type, it must be in the `Eq` type class. This can also be done automatically by putting `deriving Eq` at the end of a type definition.

To compare values of a type for order (using `<`, `<=`, *etc.*), the type must be in the `Ord` type class, which can also be done by putting `deriving Ord` at the end of a type definition. To be in `Ord`, the type must also be in `Eq`.

To derive multiple type classes, parenthesise them:

```
data Suit = Club | Diamond | Heart | Spade
          deriving (Show, Eq, Ord)
```

Disjunction and conjunction

```
data Suit = Club | Diamond | Heart | Spade
data Card = Card Suit Rank
```

A value of type `Suit` is *either* a `Club` *or* a `Diamond` *or* a `Heart` *or* a `Spade`. This *disjunction* of values corresponds to an enumerated type.

A value of type `Card` contains a value of type `Suit` *and* a value of type `Rank`. This *conjunction* of values corresponds to a structure type.

In most imperative languages, a type can represent either a disjunction or a conjunction, but not both at once.

Haskell and related languages do not have this limitation.

Discriminated union types

Haskell has *discriminated union* types, which can include both disjunction and conjunction at once.

Since disjunction and conjunction are operations in Boolean algebra, type systems that allows them to be combined in this way are often called algebraic type systems, and their types algebraic types.

```
data JokerColor = Red | Black
data JCard =
    NormalCard Suit Rank |
    JokerCard JokerColor
```

A value of type `JCard` is constructed

- *either* using the `NormalCard` constructor, in which case it contains a value of type `Suit` *and* a value of type `Rank`,
- *or* using the `JokerCard` constructor, in which case it contains a value of type `JokerColor`.

Discriminated vs undiscriminated unions

In C, you could try to represent `JCard` like this:

```
struct normalcard_struct { ... };
struct jokercard_struct { ... };
union card_union {
    struct normalcard_struct    normal;
    struct jokercard_struct     joker;
};
```

but you wouldn't know which field of the union is applicable in any given case. In Haskell, you do (the data constructor tells you), which is why Haskell's unions are said to be *discriminated*.

Note that unlike C's union types, C's enumeration types and structure types are special cases of Haskell's discriminated union types.

Discriminated union types allow programmers to define types that describe *exactly* what they mean.

NOTE C's unions are undiscriminated.

A Haskell discriminated union type in which none of the data constructors have arguments corresponds to a C enumeration type. A Haskell discriminated union type with only one data constructor corresponds to a C structure type.

Maybe

In languages like C, if you have a value of a type that is a pointer-to-T for some type T, such as a structure type, can this pointer be null?

If not, the value represents a value of type T. If yes, the value *may* represent a value of type T, or it may represent nothing. The problem is, often the reader of the code has *no idea* whether the pointer can be null.

In Haskell, if a value is optional, you indicate this by using the maybe type defined in the prelude:

```
data Maybe t = Nothing | Just t
```

For any type `t`, a value of type `Maybe t` is either `Nothing`, or `Just x`, where `x` is a value of type `t`.

NOTE This problem exists even in newer imperative languages, such as Java. In Java, if a method accepts an object as a parameter, the caller may pass to it an actual object, but it may also pass the special value `null` instead.

Section 4:
Using Haskell Types

Representing expressions in C

```
typedef enum {
    EXPR_NUM, EXPR_VAR,
    EXPR_BINOP, EXPR_UNOP
} ExprKind;
\par
\vspace{2mm}
typedef struct expr_struct *Expr;
struct expr_struct
{
    ExprKind kind;
    int      value;    /* if EXPR_NUM */
    char     *name;    /* if EXPR_VAR */
    Binop    binop;    /* if EXPR_BINOP */
    Unop     unop;     /* if EXPR_UNOP */
    Expr     subexpr1; /* if EXPR_BINOP
                          or EXPR_UNOP */
    Expr     subexpr2; /* if EXPR_BINOP */
};
```

– section 4 slide 1 –

Representing expressions in Java

```
public abstract class Expr {
    ... abstract methods ...
}
\par
\vspace{2mm}
public class NumExpr extends Expr {
    int value;
    ... implementation of abstract methods ...
}
\par
\vspace{2mm}
public class VarExpr extends Expr {
    String name;
    ... implementation of abstract methods ...
}
```

– section 4 slide 2 –

Representing expressions in Java (2)

```
public class BinExpr extends Expr {
    Binop binop;
    Expr arg1;
    Expr arg2;
    ... implementation of abstract methods ...
}
\par
\vspace{2mm}
public class UnExpr extends Expr {
    Unop unop;
    Expr arg;
    ... implementation of abstract methods ...
}
```

– section 4 slide 3 –

Representing expressions in Haskell

```
data Expr
    = Number Int
    | Variable String
    | Binop Binop Expr Expr
    | Unop Unop Expr
\par
\vspace2mm
data Binop = Plus | Minus | Times | Divide
data Unop  = Negate
```

As you can see, this is a much more direct definition of the set of values that the programmer wants to represent.

It is also much shorter, and entirely free of notes that are meaningless to the compiler and understood only by humans.

NOTE In this example, `Binop` is the name of a type as well as the name of a data constructor, and the same is true for `Unop`.

Having a type and a data constructor with the same name occurs fairly frequently in Haskell programs.

- If a type has only one data constructor, Haskell programmers will often give it the same name as the type. Example: `Card`.
- If values of a type are intended to appear as the arguments of only one data constructor, Haskell programmers will often give that data constructor the same name as the type. Example: `Binop` and `Unop`.

Comparing representions: errors

By far the most important difference is that the C representation is quite error-prone.

- You can access a field when that field is not meaningful, e.g. you can access the `subexpr2` field instead of the `subexpr1` field when `kind` is `EXPR_UNOP`.
- You can forget to initialize some of the fields, e.g. you can forget to assign to the `name` field when setting `kind` to `EXPR_VAR`.
- You can forget to process some of the alternatives, e.g. when switching on the `kind` field, you may handle only three of the four enum values.

The first mistake is literally impossible to make with Haskell, and would be caught by the Java compiler. The second is guaranteed to be caught by Haskell, but not Java. The third will be caught by Java, and by Haskell if you ask ghc to be on the lookout for it.

NOTE You can do this by invoking `ghc` with the option `-fwarn-incomplete-patterns`, or by setting the same option inside `ghci` with

```
> :set -fwarn-incomplete-patterns
```

Comparing representions: memory

The C representation requires more memory: seven words for every expression, whereas the Java and Haskell representations needs a maximum of four (one for the kind, and three for arguments/members).

Using unions can make the C representation more compact, but only at the expense of more complexity, and therefore a higher probability of programmer error.

Even with unions, the C representation needs four words for all kinds of expressions. The Java and Haskell representations need only two for numbers and variables, and three for expressions built with unary operators.

This is an example where a Java or Haskell program can actually be *more efficient* than a C program.

Comparing representions: maintenance

Adding a new kind of expression requires:

Java: Adding a new class and implementing all the methods for it

C: Adding a new alternative to the enum and adding the needed members to the type, and adding code for it to all functions handling that type

Haskell Adding a new alternative, with arguments, to the type, and adding code for it to all functions handling that type

Adding a new operation for expressions requires:

Java: Adding a new method to the abstract `Expr` class, and implementing it for each class

C: Writing one new function

Haskell Writing one new function

Switching on alternatives

You do not have to have separate equations for each possible shape of the arguments. You can test the value of a variable (which may or may not be the value of an argument) in the body of an equation, like this:

```
is_static :: Expr -> Bool
is_static expr =
    case expr of
        Number _     -> True
        Variable _   -> False
        Unop _ expr1 -> is_static expr1
        Binop _ expr1 expr2 ->
            is_static expr1 &&
            is_static expr2
```

This function figures out whether the value of an expression can be known statically, i.e. without having to know the values of variables.

NOTE As with equations, Haskell matches the value being switched on against the given patterns in order, from the top down. If you want, you can use an underscore as a wildcard pattern that matches any value, like in this example:

```
is_atomic :: Expr -> Bool
is_atomic expr =
    case expr of
        Unop _ _ -> False
        Binop _ _ _ -> False
        _ -> True
```

Missing alternatives

If you specify the option `-fwarn-incomplete-patterns`, `ghc` and `ghci` will warn about any missing alternatives, both in case expressions and in sets of equations.

This option is particularly useful during program maintenance. When you add a new alternative to an existing type, all the switches on values of that type instantly become incorrect. To fix them, a programmer must add a case to each such switch to handle the new alternative.

If you always compile the program with this option, the compiler will tell you all the switches in the program that must be modified.

Without such help, programmers must look for such switches themselves, and they may not find them all.

The consequences of missing alternatives

If a Haskell program finds a missing alternative at runtime, it will throw an exception, which (unless caught and handled) will abort the program.

Without a default case, a C program would simply go on and silently compute an incorrect result. If a default case is provided, it is likely to just print an error message and abort the program. C programmers thus have to do more work than Haskell programmers just to get up to the level of safety offered by Haskell.

If an abstract method is used in Java, this gives the same safety as Haskell. However, if overriding is used alone, forgetting to write a method for a subclass will just inherit the (probably wrong) behaviour of the superclass.

NOTE Switches in C usually need default clauses to compensate for the absence of type safety. Consider our expression representation example. In theory, a switch on `expr->kind` should need only four cases: `EXPR_NUM`, `EXPR_VAR`, `EXPR_BINOP`, and `EXPR_UNOP`. However, some other part of the program could have violated type safety by assigning e.g. `(ExprKind) 42` to `expr->kind`. You need a default case to catch and report such errors.

In Haskell, such bugs cannot happen, since the compiler will not let the programmer violate type safety.

Binary search trees

Here is one possible representation of binary search trees in C:

```
typedef struct bst_struct *BST;
struct bst_struct {
    char    *key;
    int     value;
    BST     left;
    BST     right;
};
```

Here it is in Haskell:

```
data Tree
    = Leaf
    | Node String Int Tree Tree
```

The Haskell version has two alternatives, one of which has no associated data. The C version uses a null pointer to represent this alternative.

Counting nodes in a BST

```
countnodes :: Tree -> Int
countnodes Leaf = 0
countnodes (Node _ _ l r) =
    1 + (countnodes l) + (countnodes r)
```

```
int countnodes(BST tree)
{
    if (tree == NULL) {
        return 0;
    } else {
        return 1 +
            countnodes(tree->left) +
            countnodes(tree->right);
    }
}
```

NOTE Since all we are doing is counting notes, the values of the keys and values in nodes do not matter.

In most cases, using one-character variable names is not a good idea, since such names are usually not readable. However, for anyone who knows what binary search trees are, it should be trivial to figure out that `l` and `r` refer to the left and right subtrees. Similarly for `k` and `v` for keys and values.

Pattern matching vs pointer dereferencing

The left-hand-side of the second equation in the Haskell definition naturally gives names to each of the fields of the node that actually need names (because they are used in the right hand side).

These variables do not have to be declared, and Haskell infers their types.

The C version refers to these fields using syntax that dereferences the pointer `tree` and accesses one of the fields of the structure it points to.

The C code is longer, and using Haskell-like names for the fields would make it longer still:

```
BST l = tree->left;
BST r = tree->right;
...
```

Searching a BST in C (iteration)

```
int search_bst(BST tree, char *key,
    int *value_ptr)
{
    while (tree != NULL) {
        int cmp_result;
\par
\vspace{2mm}
        cmp_result =
            strcmp(key, tree->key);
        if (cmp_result == 0) {
            *value_ptr = tree->value;
            return TRUE;
        } else if (cmp_result < 0) {
            tree = tree->left;
        } else {
            tree = tree->right;
        }
    }
\par
\vspace{2mm}
    return FALSE;
}
```

NOTE C allows functions to assign to their formal parameters (`tree` in this case), since it considers these to be ordinary local variables that just happen to be initialized by the caller.

Searching a BST in C (recursion)

```
int search_bst(BST tree, char *key,
    int *value_ptr)
{
    if (tree == NULL) {
        return FALSE;
    } else {
        int cmp_result;
\par
\vspace{2mm}
        cmp_result =
            strcmp(key, tree->key);
        if (cmp_result == 0) {
            *value_ptr = tree->value;
            return TRUE;
        } else if (cmp_result < 0) {
            return search_bst(tree->left,
                key, value_ptr);
        } else {
            return search_bst(tree->right,
                key, value_ptr);
        }
    }
}
```

Searching a BST in Haskell

```
search_bst :: Tree -> String -> Maybe Int
search_bst Leaf _ = Nothing
search_bst (Node k v l r) sk =
    if sk == k then
        Just v
    else if sk < k then
        search_bst l sk
    else
        search_bst r sk
```

- If the search succeeds, this function returns `Just v`, where `v` is the searched-for value.
- If the search fails, it returns `Nothing`.

NOTE When the tree being searched is the empty tree, the value of the key being searched for doesn't matter.

Data structure and code structure

The Haskell definitions of `countnodes` and `search_bst` have similar structures:

- an equation handling the case where the tree is empty (a `Leaf`), and
- an equation handling the case where the tree is nonempty (a `Node`).

The type we are processing has two alternatives, so these two functions have two equations: one for each alternative.

This is quite a common occurrence:

- a function whose input is a data structure will need to process all or a selected part of that data structure, and
- what the function needs to do often depends on the shape of the data, so the structure of the code often mirrors the structure of the data.

NOTE If a function doesn't need to process any part of a data structure, it shouldn't be passed that data structure as an argument.

Section 5: Adapting to Declarative Programming

Writing code

Consider a C function with two loops, and some other code around and between the loops:

```
... somefunc(...)
{
    straight line code A
    loop 1
    straight line code B
    loop 2
    straight line code C
}
```

How can you get the same effect in Haskell?

The functional equivalent

```
loop1func base case
loop1func recursive case
\par
\vspace2mm
loop2func base case
loop2func recursive case
\par
\vspace2mm
somefunc =
    let ... = ... in
    let r1 = loop1func ... in
    let ... = ... in
    let r2 = loop2func ... in
    ...
```

The only effect of the absence of iteration constructs is that instead of writing a loop inside `somefunc`, the Haskell programmer needs to write an auxiliary recursive function, usually outside `somefunc`.

NOTE In fact, Haskell allows one function definition to contain another, and if he or she wished, the programmer could put the definition of e.g. `loop1func` inside the definition of `somefunc`.

Example: C

```
int f(int *a, int size)
{
    int i;
    int target;
    int first_gt_target;
\par
\vspace{2mm}
    i = 0;
    while (i < size && a[i] <= 0) {
        i++;
    }
\par
\vspace{2mm}
    target = 2 * a[i];
    i++;
    while (i < size && a[i] <= target) {
        i++;
    }
    first_gt_target = a[i];
    return 3 * first_gt_target;
}
```

Example: Haskell version 1

```
f :: [Int] -> Int
f list =
    let
        after_skip =
            skip_init_le_zero list
    in
    case after_skip of
        (x:xs) ->
            let target = 2 * x in
            let
                first_gt_target =
                    find_gt xs target
            in
            3 * first_gt_target
\par
\vspace2mm
skip_init_le_zero :: [Int] -> [Int]
skip_init_le_zero [] = []
skip_init_le_zero (x:xs) =
    if x <= 0 then
        skip_init_le_zero xs
    else
        (x:xs)
\par
\vspace2mm
find_gt :: [Int] -> Int -> Int
find_gt (x:xs) target =
    if x <= target then
        find_gt xs target
    else
        x
```

NOTE This version has the steps of `f` directly one after another.

Example: Haskell version 2

```
f :: [Int] -> Int
f list =
  let
    skip_init_le_zero [] = []
    skip_init_le_zero (x:xs) =
```

```
    if x <= 0 then
      skip_init_le_zero xs
    else
      (x:xs)
  after_skip = skip_init_le_zero list
in
  case after_skip of
    (x:xs) ->
      let
        target = 2 * x
        find_gt (x:xs) t =
          if x <= t then
              find_gt xs t
          else
              x
        first_gt_target =
          find_gt xs target
      in
        3 * first_gt_target
```

NOTE This version has the definitions of the auxiliary functions just before their calls.

Recursion vs iteration

Functional languages do not have language constructs for iteration. What imperative language programs do with iteration, functional language programs do with recursion.

For a programmer who has known nothing but imperative languages, the absence of iteration can seem like a crippling limitation.

In fact, it is not a limitation at all. From some points of view, it can even be considered an *advantage*.

There are several viewpoints to consider.

- How does this affect the process of writing code?
- How does this affect the reliability of the resulting code?
- How does this affect the productivity of the programmers?
- How does this affect the efficiency of the resulting code?

NOTE There *are* a few languages called functional languages, such as Lisp, that do have constructs for iteration. However, these languages are hybrids, with some functional programming features and some imperative programming features, and their constructs for iteration belong to the second category.

C version vs Haskell versions

The Haskell versions use lists instead of arrays, since in Haskell, lists are the natural representation.

With `-fwarn-incomplete-patterns`, both Haskell versions will warn you that

- there may not be a strictly positive number in the list;
- there may not be a number greater than the target in the list,

and that these situations need to be handled.

The C compiler cannot generate such warnings. If the Haskell code operated on an array, the Haskell compiler couldn't either.

The Haskell versions give meaningful names to the jobs done by the loops.

NOTE Haskell supports arrays, but their use requires concepts we have not covered yet.

Reliability

The names of the auxiliary functions should remind readers of their tasks.

These functions should be documented like other functions. The documentation should give the meaning of the arguments, and describe the relationship between the arguments and the return value.

This description should allow readers to construct a correctness argument for the function.

The imperative language equivalent of these function descriptions are *loop invariants*, but they are as rare as hen's teeth in real-world programs.

The act of writing down the information needed for a correctness argument gives programmers a chance to notice situations where the (implicit or explicit) correctness argument doesn't hold water.

The fact that such writing down occurs much more often with functional programs is one factor that tends to make them more reliable.

NOTE As we saw on the previous slide, the Haskell compiler can help spot bugs as well.

Productivity

Picking a meaningful name for each auxiliary function and writing down its documentation takes time.

This cost imposed on the original author of the code is repaid manyfold when

- other members of the team read the code, and find it easier to read and understand,
- the *original author* reads the code much later, and finds it easier to read and understand.

Properly documented functions, whether created as auxiliary functions or not, can be reused. Separating the code of a loop out into a function allows the code of that function to be reused, requiring less code to be written overall.

In fact, modern functional languages come with large libraries of prewritten useful functions.

NOTE Programmers who do not document their code often find later themselves in the position of reading a part of the program they are working on, finding that they do not understand it, ask "who wrote this unreadable mess?", and then finding out they *they* wrote it.

Just because you understand your code today does *not* mean that you will understand it a few months or few years from now. Using meaningful names, documenting the meaning of each data structure, the purpose of each function, the reasons for each design decision, and in general following the rules of good programming style will help your future self as well as your teammates in both the present and the future.

Efficiency

The recursive version of e.g. `search_bst` will allocate one stack frame for each node of the tree it traverses, while the iterative version will just allocate one stack frame period.

The recursive version will therefore be less efficient, since it needs to allocate, fill in and then later deallocate more stack frames.

The recursive version will also need more stack space. This should not be a problem for `search_bst`, but the recursive versions of some other functions can run out of stack space.

However, compilers for declarative languages put huge emphasis on the optimization of recursive

code. In many cases, they can take a recursive algorithm in their source language (e.g. Haskell), and generate iterative code in their target language.

– section 5 slide 10 –

Efficiency in general

Overall, programs in declarative languages are typically slower than they would be if written in C. Depending on which declarative language and which language implementation you are talking about, and on what the program does, the slowdown can range from a few percent to huge integer factors, such as 10% to a factor of a 100.

However, popular languages like Python and Javascript typically also yield significantly slower programs than C. In fact, their programs will typically be significantly *slower* than corresponding Haskell programs.

In general, the higher the level of a programming language (the more it does for the programmer), the slower its programs will be on average. The price of C's speed is the need to handle all the details yourself.

The right point on the productivity vs efficiency tradeoff continuum depends on the project (and component of the project).

– section 5 slide 11 –

Immutable data structures

In declarative languages, data structures are *immutable*: once created, they cannot be changed. So what do you do if you *do* need to update a data structure?

You create another version of the data structure, one which has the change you want to make, and use *that* version from then on.

However, if you want to, you can hang onto the old version as well. You will definitely want to do so if some part of the system still needs the old version (in which case imperative code must also make a modified copy).

The old version can also be used

- to inspect the initial version in a debugger
- to implement undo
- to gather statistics, e.g. about how the size of a data structure changes over time

– section 5 slide 12 –

Updating a BST

```
insert_bst :: Tree -> String -> Int -> Tree
insert_bst Leaf ik iv = Node ik iv Leaf Leaf
insert_bst (Node k v l r) ik iv =
    if ik == k then
        Node ik iv l r
    else if ik < k then
        Node k v (insert_bst l ik iv) r
    else
        Node k v l (insert_bst r ik iv)
```

Note that *all* of the code of this function is concerned with the job at hand; there is no code concerned with memory management.

In Haskell, as in Java, memory management is automatic. Any unreachable cells of memory are recovered by the garbage collector.

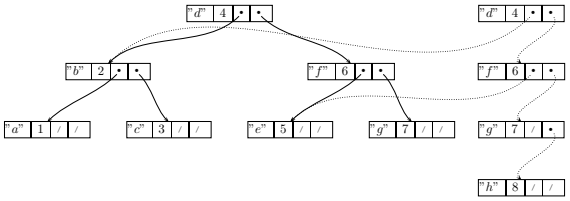
– section 5 slide 13 –

Reusing memory

When `insert_bst` inserts a new node into the tree, it allocates new versions of every node on the path

from the root to the insertion point. However, *every other node in the tree* will become part of the new tree as well as the old one.

This shows what happens when you insert the key "h" into a binary search tree that already contains "a" to "g".



NOTE In this small example, the old and the new tree share the entire subtree rooted at the node whose key is “b’, and the node whose key is “e’.

Memory efficiency

Insertion into a BST replaces one node on each level of the tree: the node on the path from the root to the insertion site.

In (mostly) balanced trees with n nodes, the height of the tree tends to be about $log_2(n)$.

Therefore the number of nodes allocated during an insertion tends to be logarithmic in the size of the tree.

- If the old version of the tree is not needed, imperative code can do better: it must allocate only the new node.
- If the old version of the tree is needed, imperative code will do worse: it must copy the entire tree, since without that, later updates to the new version would update the old one as well.

Converting a list of pairs to a BST

One can build a BST by starting with the empty tree and inserting key/value pairs into it. This could be done by this function:

```
assoc_list_to_bst :: [(String, Int)] -> Tree
assoc_list_to_bst [] = Leaf
assoc_list_to_bst ((hk, hv):kvs) =
    let t0 = assoc_list_to_bst kvs
    in insert_bst t0 hk hv
```

In Haskell, an *n-tuple* is a sequence of n values within parentheses.

Here, `(String, Int)` is the type of 2-tuples (usually called *pairs*) in which the first element is of type `String` and the second element is of type `Int`.

`((hk, hv):t)` is a pattern for a list in which the head is a key/value pair. The outer parentheses are for grouping, while the inner ones denote a tuple.

Section 6: Polymorphism

Polymorphic types

Our definition of the tree type so far was this:

```
data Tree =
    Leaf |
    Node String Int Tree Tree
```

This type assumes that the keys are strings and the values are integers. However, the functions we have written to handle trees (`countnodes` and `search_bst` *do not really care about the types of the keys and values.*

We could also define trees like this:

```
data Tree k v
    = Leaf
    | Node k v (Tree k v) (Tree k v)
```

In this case, `k` and `v` are *type variables*, variables standing in for the types of keys and values, and `Tree` is a *type constructor*, which constructs a new type from two other types.

Using polymorphic types: `countnodes`

With the old, *monomorphic* definition of `Tree`, the type declaration or *signature* of `countnodes` was:

```
countnodes :: Tree -> Int
```

With the new, *polymorphic* definition of `Tree`, it will be

```
countnodes :: Tree k v -> Int
```

Regardless of the types of the keys and values in the tree, `countnodes` will count the number of nodes in it.

The *exact same code* works in these cases.

NOTE In C++, you can use templates to arrange to get the same piece of code to work on values of different types, but the C++ compiler will generate different object code for each type. This can make executables significantly bigger than they need to be.

Using polymorphic types: `search_bst`

`countnodes` does not touch keys or values, but `search_bst` does perform some operations on keys. Replacing

```
search_bst :: Tree -> String -> Maybe Int
```

with

```
search_bst :: Tree k v -> k -> Maybe v
```

will not work; it will yield an error message.

The reason is that `search_bst` contains these two tests:

- a comparison for equality: `sk == k`, and
- a comparison for order: `sk < k`.

Comparing values for equality and order

Some types cannot be compared for equality. For example, two functions should be considered equal

if for all sets of input argument values, they compute the same result. Unfortunately, it has been proven that testing whether two functions are equal is *undecidable*. This means that building an algorithm that is guaranteed to decide in finite time whether two functions are equal is *impossible*.

Some types that can be compared for equality cannot be compared for order. Consider a set of integers. It is obvious that `{1, 5}` is not equal to `{2, 4}`, but using the standard method of set comparison (set inclusion), they are otherwise incomparable; neither can be said to be greater than the other.

`Eq` and `Ord`

In Haskell,

- comparison for equality can only be done on values of types that belong to the type class `Eq`, while
- comparison for order can only be done on values of types that belong to the type class `Ord`.

Membership of `Ord` implies membership of `Eq`, but not vice versa.

The declaration of `search_bst` should be this:

```
search_bst ::
    Ord k => Tree k v -> k -> Maybe v
```

The construct `Ord k =>` is a type class constraint; it says `search_bst` requires whatever type `k` stands for to be in `Ord`. This guarantees its membership of `Eq` as well.

Deriving membership automatically

```
data Suit = Club | Diamond | Heart | Spade
    deriving (Show, Eq, Ord)
data Card = Card Suit Rank
    deriving (Ord, Show)
```

The automatically created comparison function takes the order of data constructors from the order in the declaration itself: a constructor listed earlier is less than a constructor listed later (e.g. `Club < Diamond`).

If the two values being compared have the same top level data constructor, the automatically created comparison function compares their arguments in turn, from left to right. This means the argument types must also be instances of `Ord`. If the corresponding arguments are not equal, the comparison stops (e.g. `Card Club Ace < Card Spade Jack`); if the corresponding argument are equal, it goes on to the next argument, if there is one (e.g. `Card Spade Ace > Card Spade Jack`). This is called *lexicographic* ordering.

Recursive vs nonrecursive types

```
data Tree
    = Leaf
    | Node String Int Tree Tree
data Card = Card Suit Rank
```

`Tree` is a recursive type because some of its data constructors have arguments of type `Tree`.

`Card` is a non-recursive type because none of its data constructors have an arguments of type `Card`.

A recursive type needs a nonrecursive alternative, because without one, all values of the type would have infinite size.

Mutually recursive types

Some types are recursive but not *directly* recursive.

```
data BoolExpr
    = BoolConst Bool
    | BoolOp BoolOp BoolExpr BoolExpr
    | CompOp CompOp IntExpr IntExpr
data IntExpr
    = IntConst Int
    | IntOp IntOp IntExpr IntExpr
    | IntIfThenElse BoolExpr IntExpr IntExpr
```

In a mutually recursive set of types, it is enough for one of the types to have a nonrecursive alternative.

These types represent Boolean- and integer-valued expressions in a program. They must be mutually recursive because comparison of integers returns a Boolean and integer-valued conditionals use a Boolean.

NOTE Provided that at least one of the recursive alternatives is only mutually recursive, not directly recursive. That way, a value built using that alternative can be of finite size.

Structural induction

Code that follows the shape of a nonrecursive type tends to be simple. Code that follows the shape of a directly or mutually recursive type tends to be more interesting.

Consider a recursive type with one nonrecursive data constructor (like `Leaf` in `Tree`) and one recursive data constructor (like `Node` in `Tree`). A function that follows the structure of this type will typically have

- an equation for the nonrecursive data constructor, and
- an equation for the recursive data constructor.

Typically, recursive calls will occur only in the second equation, and the switched-on argument in the recursive call will be *strictly smaller* than the corresponding argument in the left hand side of the equation.

Proof by induction

You can view the function definition's structure as the outline of a correctness argument.

The argument is a proof by induction on n, the number of data constructors of the switched-on type in the switched-on argument.

- *Base case:* If $n = 1$, then the applicable equation is the base case. If the first equation is correct, then the function correctly handles the case where $n = 1$.
- *Induction step:* Assume the induction hypothesis: the function correctly handles all cases where $n \leq k$. This hypothesis implies that all the recursive calls are correct. If the second equation is correct, then the function correctly handles all cases where $n \leq k + 1$.

The base case and the induction step together imply that the function correctly handles all inputs.

Formality

If you want, you can use these kinds of arguments to formally *prove* the correctness of functions, and of entire functional programs.

This typically requires a formal specification of the expected relationship between each function's arguments and its result.

Typical software development projects do not do formal proofs of correctness, regardless of what kind of language their code is written in.

However, projects using functional languages do tend to use *informal* correctness arguments slightly more often.

The support for this provided by the original programmer usually consists of nothing more than a natural language description of the criterion of correctness of each function. Readers who want a correctness argument can then construct it for themselves from this and the structure of the code.

Structural induction for more complex types

If a type has nr nonrecursive data constructors and r recursive data constructors, what happens when $nr > 1$ or $r > 1$, like `BoolExpr` and `IntExpr`?

You can do structural induction on such types as well.

Such functions will typically have nr nonrecursive equations and r recursive equations, but not always. Sometimes you need more than one equation to handle a constructor, and sometimes one equation can handle more than one constructor. For example, sometimes all base cases need the same treatment.

Picking the right representation of the data is important in every program, but when the structure of the code follows the structure of the data, it is particularly important.

NOTE If all the base cases can be handled the same way, then the function can start with r equations for the recursive constructors, followed by a single equation using a wildcard pattern that handles all the nonrecursive constructors.

Let clauses and where clauses

```
assoc_list_to_bst ((hk, hv):kvs) =
    let t0 = assoc_list_to_bst kvs
    in insert_bst t0 hk hv
```

A let clause `let name = expr in mainexpr` introduces a name for a value to be used in the main expression.

```
assoc_list_to_bst ((hk, hv):kvs) =
    insert_bst t0 hk hv
    where t0 = assoc_list_to_bst kvs
```

A where clause `mainexpr where name = expr` has the same meaning, but has the definition of the name *after* the main expression.

Which one you want to use depends on where you want to put the emphasis.

But you can only use `where` clauses *at the top level of a function*, while you can use a `let` for any expression.

NOTE In theory, you can also mix the two, like this:

```
let name1 = expr1
in mainexpr
where name2 = expr2
```

However, you should *not* do this, since it is definitely *bad* programming style.

Defining multiple names

You can define multiple names with a single let or where clause:

```
let name1 = expr1
    name2 = expr2
in mainexpr
\par
\vspace2mm
mainexpr
where
    name1 = expr1
    name2 = expr2
```

The scope of each name includes the right hand sides of the definitions of the following names, as well as the main expression, *unless* one of the later definitions defines the same name, in which case the original definition is *shadowed* and not visible from then on.

Section 7: Higher order functions

First vs higher order

First order values are data.

Second order values are functions whose arguments and results are first order values.

Third order values are functions whose arguments and results are first or second order values.

In general, nth order values are functions whose arguments and results are values of any order from first up to $n - 1$.

Values that belong to an order higher than first are higher order values.

Java 8, released mid-2014, supports higher order programming. C also supports it, if you work at it. Higher order programming is a central aspect of Haskell, often allowing Haskell programmers to avoid writing recursive functions.

A higher order function in C

```
IntList filter(Bool (*f)(int), IntList list)
{
    IntList filtered_tail, new_list;
\par
\vspace{2mm}
    if (list == NULL) {
        return NULL;
    } else {
```

```
        filtered_tail =
            filter(f, list->tail);
        if ((*f)(list->head)) {
            new_list = checked_malloc(
                sizeof(*new_list));
            new_list->head = list->head;
            new_list->tail = filtered_tail;
            return new_list;
        } else {
            return filtered_tail;
        }
    }
}
```

NOTE This code assumes type definitions like these:

```
typedef struct intlist_struct *IntList;
\par
\vspace{2mm}
struct intlist_struct {
    int     head;
    IntList tail;
};
\par
\vspace{2mm}
typedef int Bool;
```

Unfortunately, although the last typedef allows programmers to write `Bool` instead of `int` to tell readers of the code that something (in this case, the return value of the function) is meant to be used to represent only a TRUE/FALSE distinction, the C compiler will not report as errors any arithmetic operations on booleans, any mixing of booleans and integers, or in general any unintended use of a boolean as an integer or vice versa. Since booleans and integers are distinct builtin types in Haskell, any such errors in Haskell programs *will* be caught by the Haskell implementation.

A higher order function in Haskell

Haskell's syntax for passing a function as an argument is much simpler than C's syntax. All you need to do is wrap the type of the higher order argument in parentheses to tell Haskell it is one argument.

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter f (x:xs) =
    if f x then x:fxs else fxs
    where fxs = filter f xs
```

Even though it is significantly shorter, this function is actually more general than the C version, since it is polymorphic, and thus works for lists with *any* type of element.

`filter` is defined in the Haskell prelude.

Using higher order functions

You can call `filter` like this:

```
... filter is_even [1, 2, 3, 4] ...
... filter is_pos [0, -1, 1, -2, 2] ...
... filter is_long ["a", "abc", "abcde"] ...
```

given definitions like this:

```
is_even :: Int -> Bool
is_even x =
    if (mod x 2) == 0 then
        True
    else
        False
\par
\vspace2mm
is_pos :: Int -> Bool
is_pos x = if x > 0 then True else False
\par
\vspace2mm
is_long :: String -> Bool
is_long x =
```

```
if length x > 3 then
    True
else
    False
```

NOTE `length` is a function defined in the Haskell prelude. As its name implies, it returns the length of a list. Remember that in Haskell, a string is a list of characters.

The `mod` function is similar to the `%` operator in C: in this case, it returns the remainder after dividing `x` by 2.

Backquote

Modulo is a built-in infix operator in many languages. For example, in C or Java, 5 modulo 2 would be written `5 % 2`.

Haskell uses `mod` for the modulo operation, but Haskell allows you to make any function an infix operator by surrounding the function name with backquotes (backticks, written ‘).

So a friendlier way to write the `is_even` function would be:

```
is_even :: Int -> Bool
is_even x = x `mod` 2 == 0
```

Operators written with backquotes have high precedence and associate to the left.

It's also possible to explicitly declare non-alphanumeric operators, and specify their associativity and fixity, but this feature should be used sparingly.

Anonymous functions

In some cases, the only thing you need a function for is to pass as an argument to a higher order function like `filter`. In such cases, readers may find it more convenient if the call contained the *definition* of the function, not its name.

In Haskell, anonymous functions are defined by *lambda expressions*, and you use them like this.

```
... filter (\x -> x `mod` 2 == 0)
        [1, 2, 3, 4] ...
... filter (\s -> length s > 3)
        ["a", "abc", "abcde"] ...
```

This notation is based on the lambda calculus, the basis of functional programming.

In the lambda calculus, each argument is preceded by a lambda, and the argument list is followed by a dot and the expression that is the function body. For example, the function that adds together its two arguments is written as $\lambda a.\lambda b.a + b$.

NOTE These calls are equivalent to the calls

```
... filter is_even [1, 2, 3, 4] ...
... filter is_long ["a", "abc", "abcde"] ...
```

given our earlier definitions of `is_even` and `is_long`.

Map

`map` is one of the most frequently used Haskell functions. (It is defined in the Haskell prelude.) Given a function and a list, `map` applies the function to every member of the list.

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

Many things that an imperative programmer would do with a loop, a functional programmer would do with a call to `map`. An example:

```
get_names :: [Customer] -> [String]
get_names customers =
    map customer_name customers
```

This assumes that `customer_name` is a function whose type is `Customer -> String`.

Partial application

Given a function with n arguments, *partially applying* that function means giving it its first k arguments, where $k < n$.

The result of the partial application is a *closure* that records the identity of the function and the values of those k arguments.

This closure behaves as a function with $n - k$ arguments. A call of the closure leads to a call of the original function with both sets of arguments.

```
is_longer :: Int -> String -> Bool
is_longer limit x = length x > limit
\par
\vspace2mm
...
filter (is_longer 4)
    ["ab", "abcd", "abcdef"]
...
```

In this case, the function `is_longer` takes two arguments. The expression `is_longer 4` partially applies this function, and creates a closure which records `4` as the value of the first argument.

NOTE There is no way to partially apply a function (as opposed to an operator, see below) by supplying it with k arguments if those not are not the *first* k arguments.

Sometimes, programmers define small, sometimes anonymous helper functions which simply call another function with a different order of arguments, the point being to bring the k arguments you want to supply to that function to the start of the argument list of the helper function.

Calling a closure: an example

```
filter f (x:xs) =
    if f x then x:fxs else fxs
    where fxs = filter f xs
\par
\vspace2mm
...
filter (is_longer 4)
    ["ab", "abcd", "abcdef"]
...
```

In this case, the code of `filter` will call `is_longer` three times:

- `is_longer 4 "ab"`
- `is_longer 4 "abcd"`
- `is_longer 4 "abcdef"`

Each of these calls comes from the higher order call `f x` in `filter`. In this case `f` represents the closure `is_longer 4`. In each case, the first argument comes from the closure, with the second being the value of `x`.

Operators and sections

If you enclose an infix operator in parentheses, you can partially apply it by enclosing its left or right operand with it; this is called a *section*.

```
Prelude> map (*3) [1, 2, 3]
[3,6,9]
```

You can use section notation to partially apply *either* of its arguments.

```
Prelude> map (5 `mod`) [3, 4, 5, 6, 7]
[2,1,0,5,5]
Prelude> map (`mod` 3) [3, 4, 5, 6, 7]
[0,1,2,0,1]
```

Types for partial application

In most languages, the type of a function with n arguments would be something like:

```
f :: (at1, at2, ... atn) -> rt
```

where `at1`, `at2` etc are the argument types, `(at1, at2, ... atn)` is the type of a tuple containing all the arguments, and `rt` is the result type.

To allow the function to be partially applied by supplying the first argument, you need a function with a different type:

```
f :: at1 -> ((at2, ... atn) -> rt)
```

This function takes a single value of type `at1`, and returns as its result another function, which is of type `(at2, ... atn) -> rt`.

Currying

You can keep transforming the function type until every single argument is supplied separately:

```
f :: at1 -> (at2 -> (at3 -> ...
    (atn -> rt)))
```

The transformation from a function type in which all arguments are supplied together to a function type in which the arguments are supplied one by one is called *currying*.

In Haskell, *all* function types are curried. This is why the syntax for function types is what it is. The arrow that makes function types is right associative, so the second declaration below just shows explicitly the parenthesization implicit in the first:

```
is_longer :: Int -> String -> Bool
is_longer :: Int -> (String -> Bool)
```

NOTE Currying and the Haskell programming language are both named after the same person, the English mathematician Haskell Brooks Curry. He did a lot to develop the lambda calculus, the mathematical foundation of functional programming, and a result, he is a popular guy in functional programming circles. In fact, there are *two* functional programming languages named for him: besides Haskell, there is another one named Curry.

Functions with all their arguments

Given a function with curried argument types, you can supply the function its first argument, then its second, then its third, and so on. What happens when you have supplied them all?

```
is_longer 3 ["a", "ab", "abc", "abcd"]
```

There are two things you can get:

- a closure that contains all the function's arguments, or
- the result of the evaluation of the function.

In C and in most other languages, these would be very different, but in Haskell, as we will see later, they are equivalent.

Composing functions

Any function that makes a higher order function call or creates a closure (e.g. by partially applying another function) is a second order function. This means that both `filter` and its callers are second order functions.

`filter` has a piece of data as an argument (the list to filter) as well as a function (the filtering function). Some functions do not take *any* piece of data as arguments; *all* their arguments are functions.

The builtin operator '.' composes two functions. The expression `f . g` represents a function which first calls `g`, and then invokes `f` on the result:

```
(f . g) x = f (g x)
```

If the type of `x` is represented by the type variable `a`, then the type of `g` must be `a -> b` for some `b`, and the type of `f` must be `b -> c` for some `c`. The type of `.` itself is therefore `(b -> c) -> (a -> b) -> (a -> c)`.

NOTE There is nothing preventing some or all of those type variables actually standing for the same type.

Composing functions: some examples

Suppose you already have a function that sorts a list and a function that returns the head of a list, if it has one. You can then compute the minimum of the list like this:

```
min = head . sort
```

If you also have a function that reverses a list, you can also compute the maximum with very little extra code:

```
max = head . reverse . sort
```

This shows that functions created by composition, such as `reverse . sort`, can themselves be part of further compositions.

This style of programming is sometimes called *point-free style*, though *value-free style* would be a better description, since its distinguishing characteristic is the absence of variables representing (first order) values.

NOTE The `.` operator is right associative, so `head . reverse . sort` parenthesizes as `head . (reverse . sort)`, not as `(head . reverse) . sort`, even though the two parenthesizations in fact yield functions that compute the same answers for all possible argument values.

Given the above definition, `max xs` is equivalent to `(head . reverse . sort) xs`, which in turn is equivalent to `head (reverse (sort xs))`.

Code written in point-free style can contain variables representing functions; the definition of `.` is an example.

Code written in point free style is usually very short, and it can be very elegant. However, while elegance is nice, it is not the most important characteristic that programmers should strive for. If most readers cannot understand a piece of code, its elegance to the few readers that do understand it is of little concern, and unfortunately, a large fraction of programmers find code written in point-free style hard to understand.

Composition as sequence

Function composition is one way to express a sequence of operations. Consider the function composition `step3f . step2f . step1f`.

1. You start with the input, `x`.

2. You compute `step1f x`.
3. You compute `step2f (step1f x)`.
4. You compute `step3f (step2f (step1f x))`.

This idea is the basis of *monads*, which is the mechanism Haskell uses to do input/output.

Section 8: Functional design patterns

Higher order programming

Higher order programming is widely used by functional programmers. Its advantages include

- code reuse,
- a higher level of abstraction, and
- a set of canned solutions to frequently encountered problems.

In programs written by programmers who do not use higher order programming, you frequently find pieces of code that have the same structure but slot different pieces of code into that structure.

Such code typically qualifies as an instance of the copy-and-paste programming *antipattern*, a pattern that programmers should strive to avoid.

NOTE This is an antipattern because the many different copies of the code structure violate a variant of the principle of single point of control. If you find a bug in one of the copies, that bug may be present in other copies as well, but you don't get help in finding out where those copies are. With higher order code, fixing the code of the higher order function itself fixes that bug in all calls to it.

Folds

We have already seen the functions `map` and `filter`, which operate on and transform lists.

The other class of popular higher order functions on lists are the *reduction* operations, which reduce a list to a single value.

The usual reduction operations are *folds*. There are three main folds: left, right and balanced.

left $((((I\ o\ X_1)\ o\ X_2)\ldots)\ o\ X_n)$

right $(X_1\ o\ (X_2\ o\ (\ldots(X_n\ o\ I))))$

balanced $((X_1\ o\ X_2)\ o\ (X_3\ o\ X_4))\ o\ \ldots$

Here o denotes the folding operation, and I denotes the identity element of that operation. (The balanced fold also needs the identity element in case the list is empty.)

Foldl

```
foldl :: (v -> e -> v) -> v -> [e] -> v
foldl _ base [] = base
foldl f base (x:xs) =
    let newbase = f base x in
    foldl f newbase xs
\par
\vspace2mm
suml :: [Integer] -> Integer
suml = foldl (+) 0
\par
\vspace2mm
productl :: [Integer] -> Integer
productl = foldl (*) 1
\par
\vspace2mm
concatl :: [[a]] -> [a]
concatl = foldl (++) []
```

Foldr

```
foldr :: (e -> v -> v) -> v -> [e] -> v
foldr _ base [] = base
foldr f base (x:xs) =
    let fxs = foldr f base xs in
    f x fxs
\par
\vspace2mm
sumr = foldr (+) 0
productr = foldr (*) 1
concatr = foldr (++) []
```

You can define sum, product and concatenation in terms of both `foldl` and `foldr` because addition and multiplication on integers, and list append, are all associative operations.

NOTE The declarations of `sumr`, `productr` and `concatr` differ from the declarations of `suml`, `productl` and `concatl` only in the function name.

Addition and multiplication are *not* associative on floating point numbers, because of their limited precision. For the sake of simplicity in the discussion, suppose the limit is four decimal digits in the fraction, and suppose you want to sum up the list of numbers [0.25, 0.25,0.25, 0.25, 1000]. If you do the additions from left to right, the first addition adds 0.25 and 0.25 giving 0.5, the second adds 0.5 and 0.25 giving 0.75, the third adds 0.75 and 0.25 giving 1.0, and the fourth adds 1 and 1000, yielding 1001. The final addition of the identity element 0 does not change this result. However, if you do the additions right to left, the result is different. This is because adding 0.25 and 1000 cannot yield 1000.25, since that has too many digits. Instead, 1000.25 must be rounded to the nearest number with four decimal digits, which will be 1000. The next three additions of 0.25 and the final addition of 0 will still leave the overall result at 1000.

While `concatl` and `concatr` are guaranteed to generate the same result, `concatr` is much more efficient than `concatl`. We will discuss this later.

Balanced fold

```
balanced_fold :: (e -> e -> e) -> e ->
    [e] -> e
balanced_fold _ b [] = b
balanced_fold _ _ (x:[]) = x
balanced_fold f b l@(_:_:_) =
    let
        len = length l
        (half1, half2) =
            splitAt (div len 2) l
        value1 = balanced_fold f b half1
        value2 = balanced_fold f b half2
    in
        f value1 value2
```

`splitAt n l` returns a pair of the first `n` elements of `l` and the rest of `l`. It is defined in the standard Prelude.

NOTE This code does some wasted work. Each call to `balanced_fold` computes the length of its list, but for recursive calls, the caller already knows the length. (It is `div len 2` for `half1` and `len - (div len 2)` for `half2`.) Also, `balanced_fold` does not make recursive calls unless the length of the list is at least two. This means that the length of both `half1` and `half2` will be at least one, which means that the test for zero length lists can succeed only for the top level call; for all the recursive calls, that test is wasted work. It is of course possible to write a balanced fold that does not have either of these performance problems.

MapReduce

MapReduce is Google's second most important algorithm after PageRank. Google uses it to implement searches on its distributed data store. It has two stages:

- The Map stage asks each computer holding a part of the data store (which is a set of web page descriptions) for the set of pages relevant to the query, in decreasing order of relevance.
- The Reduce stage merges these ordered lists of relevant page descriptions into a single ordered list.

The Map stage is effectively an application of the Haskell builtin function map, except implemented in parallel.

The Reduce stage is effectively an application of a fold. For efficiency, the fold is a balanced fold executed in parallel, so its complexity is logarithmic, not linear, in the number of machines.

NOTE If the query is set to display only the top N results on the first page of results, then

- it is ok for each machine to report only its top N results, and
- it is ok for each merge to keep only the top N items in the merged list.

A later query (if there is one) for the kth page of results (where k=2,3, etc, or even k=1) can then retrieve the top kN results, and show only the last N.

PageRank is the algorithm on which Google was founded. It ranks the web pages associated with a search query in order of likely relevance, using links from other pages with terms related to query as votes for the relevance of the linked-to page to the query.

More folds

The Haskell prelude defines `sum`, `product`, and `concat`.

For `maximum` and `minimum`, there is no identity element, and it is an error if the list is empty. For such cases, the Haskell Prelude defines:

```
foldl1 :: (a -> a -> a) -> [a] -> a
foldr1 :: (a -> a -> a) -> [a] -> a
```

that compute

foldl1 $o\ [X_1, X_2, \ldots, X_n] = ((X_1\ o\ X_2) \ldots\ o\ X_n)$

foldr1 $o\ [X_1, X_2, \ldots, X_n] = (X_1\ o\ (X_2\ o\ \ldots X_n))$

```
maximum = foldr1 max
minimum = foldr1 min
```

You could equally well use `foldl1` for these.

Folds are really powerful

You can compute the length of a list by summing 1 for each element, instead of the element itself. So if we can define a function that takes anything and returns 1, together with `(+)` we can use fold to define `length`.

```
const :: a -> b -> a
const a b = a
\par
\vspace2mm
length = foldr ((+) . const 1) 0
```

You can map over a list with `foldr`:

```
map f = foldr ((:) . f) []
```

Fold can reverse a list

If we had a "backwards" `(:)` operation, call it `snoc`, then `foldl` could reverse a list:

`reverse` $[X_1, X_2, \ldots X_n] = []$ `` `snoc` `` X_1 `` `snoc` `` X_2 $\ldots$ `` `snoc` `` X_n

```
snoc :: [a] -> a -> [a]
snoc tl hd = hd:tl
\par
\vspace2mm
reverse = foldl snoc []
```

But the Haskell Prelude defines a function to flip the arguments of a binary function:

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
\par
\vspace2mm
reverse = foldl (flip (:)) []
```

List comprehensions

Haskell has special syntax for one class of higher order operations. These two implementations of quicksort do the same thing, with the first using conventional higher order code, and the second using list comprehensions:

```
qs1 [] = []
qs1 (x:xs) = qs1 littles ++
             [x] ++
             qs1 bigs
    where
        littles = filter (<x) xs
        bigs    = filter (>=x) xs
\par
\vspace2mm
```

```
qs2 [] = []
qs2 (x:xs) = qs2 littles ++
             [x] ++
             qs2 bigs
    where
        littles = [l | l <- xs, l < x]
        bigs    = [b | b <- xs, b >= x]
```

List comprehensions

List comprehensions can be used for things other than filtering a *single* list.

In general, a list comprehension consists of

- a template (an expression, which is often just a variable)
- one or more generators (each of the form `var <- list`),
- zero or more tests (boolean expressions),
- zero or more let expressions defining local variables.

Some more examples:

```
columns = "abcdefgh"
rows =    "12345678"
chess_squares = [[c, r]
                | c <- columns, r <- rows]
\par
\vspace2mm
pairs = [(a, b)
        | a <- [1, 2, 3], b <- [1, 2, 3]]
nums  = [10*a+b
        | a <- [1, 2, 3], b <- [1, 2, 3]]
```

Traversing HTML documents

Types to represent HTML documents:

```
type HTML = [HTML_element]
data HTML_element
    = HTML_text String
    | HTML_font Font_tag HTML
    | HTML_p HTML
data Font_tag = Font_tag (Maybe Int)
                         (Maybe String)
                         (Maybe Font_color)
data Font_color
    = Colour_name String
    | Hex Int
    | RGB Int Int Int
```

Collecting font sizes

```
font_sizes_in_html ::
    HTML -> Set Int -> Set Int
font_sizes_in_html elements sizes =
    foldr font_sizes_in_elt sizes elements
\par
\vspace2mm
font_sizes_in_elt ::
    HTML_element -> Set Int -> Set Int
font_sizes_in_elt (HTML_text _) sizes =
    sizes
font_sizes_in_elt
    (HTML_font font_tag html) sizes =
    let
        Font_tag maybe_size _ _ = font_tag
        newsizes = case maybe_size of
            Nothing ->
                sizes
            Just fontsize ->
                Data.Set.insert
                    fontsize sizes
    in
        font_sizes_in_html html newsizes
font_sizes_in_elt (HTML_p html) sizes =
    font_sizes_in_html html sizes
```

NOTE Normally, `font_sizes_in_html` would be invoked with the empty set as the second argument, which would mean that the returned set is the set of font sizes appearing in the given HTML page description. The second argument of `font_sizes_in_html` thus plays the role of an accumulator.

Collecting font names

```
font_names_in_html ::
    HTML -> Set String -> Set String
font_names_in_html elements names =
    foldr font_names_in_elt names elements
\par
\vspace2mm
font_names_in_elt ::
    HTML_element -> Set String -> Set String
font_names_in_elt (HTML_text _) names = names
font_names_in_elt
    (HTML_font font_tag html) names =
    let
        Font_tag _ maybe_name _ = font_tag
        newnames = case maybe_name of
            Nothing ->
                names
            Just fontname ->
                Data.Set.insert
                    fontname names
    in
        font_names_in_html html newnames
font_names_in_elt (HTML_p html) names =
    font_names_in_html html names
```

Collecting any font information

```
font_stuff_in_html ::
    (Font_tag -> a -> a) -> HTML -> a -> a
font_stuff_in_html f elements stuff =
    foldr (font_stuff_in_elt f) stuff
        elements
\par
\vspace2mm
font_stuff_in_elt :: (Font_tag -> a -> a) ->
    HTML_element -> a -> a
font_stuff_in_elt f
    (HTML_text _) stuff = stuff
font_stuff_in_elt f
    (HTML_font font_tag html) stuff =
    let newstuff = f font_tag stuff in
    font_stuff_in_html f html newstuff
font_stuff_in_elt f (HTML_p html) stuff =
    font_stuff_in_html f html stuff
```

Collecting font sizes again

```
font_sizes_in_html' ::
    HTML -> Set Int -> Set Int
font_sizes_in_html' html sizes =
    font_stuff_in_html accumulate_font_sizes
        html sizes
\par
\vspace2mm
accumulate_font_sizes font_tag sizes =
    let Font_tag maybe_size _ _ = font_tag in
    case maybe_size of
        Nothing ->
            sizes
        Just fontsize ->
            Data.Set.insert
            fontsize sizes
```

Using the higher order version avoids duplicating the code that traverses the data structure. The benefit you get from this scales linearly with the complexity of the data structure being traversed.

Comparison to the visitor pattern

The function `font_stuff_in_html` does a job that is very similar to the job that the visitor design pattern would do in an object-oriented language like Java: they both traverse a data structure, invoking a function at one or more selected points in the code. However, there are also differences.

- In the Haskell version, the type of the higher order function makes it clear whether the code executed at the selected points just gathers information, or whether it modifies the traversed data structure. In Java, the invoked code is imperative, so it can do either.
- The Java version needs an `accept` method in every one of the classes that correspond to Haskell types in the data structure (in this case, `HTML` and `HTML_element`).
- In the Haskell version, the functions that implement the traversal can be (and typically are) next to each other. In Java, the corresponding methods have to be dispersed to the classes to which they belong.

Libraries vs frameworks

The way typical libraries work in any language (including C and Java as well as Haskell) is that code written by the programmer calls functions in the library.

In some cases, the library function is a higher order function, and thus it can call back a function supplied to it by the programmer.

Application frameworks are libraries but they are not typical libraries, because they are intended to be the top layer of a program.

When a program uses a framework, the framework is in control, and it calls functions written by the programmer when circumstances call for it.

For example, a framework for web servers would handle all communication with remote clients. It would itself implement the event loop that waits for the next query to arrive, and would invoke user code only to generate the response to each query.

Frameworks: libraries vs application generators

Frameworks in Haskell can be done like this, with `framework` simply being a library function:

```
main = framework plugin1 plugin2 plugin3
\par
\vspace2mm
plugin1 = ...
plugin2 = ...
plugin3 = ...
```

This approach could also be used in other languages, since even C and Java support callbacks, though sometimes clumsily.

Unfortunately, many frameworks instead just generate code (in C, C#, Java, ...) that the programmer is then expected to modify. This approach throws abstraction out the window, and is much more error-prone.

NOTE Application generators expose to the programmer all the details of their implementation. This allows the programmer to modify those details if needed, but the next invocation of the application generator will destroy those modifications, which means the application generator can be used only once. If a new version of the application generator comes out that fixes some problems with the old version, the programmer has no good options: there is no easy way to integrate the new version's bug fixes with his or her own earlier modifications.

However, bugs in which programmers modify the wrong part of the generated code or forget to modify a part they *should* have modified can be expected to be made considerably more frequently.

Section 9: Introduction to Logic Programming

Logic programming

Functional programming languages are based on the lambda calculus of Alonzo Church and the concept of the *function*: a thing that maps inputs to outputs.

Logic programming languages are based on the predicate calculus of Gottlob Frege and the concept of the *relation*, the relationship among a number of individuals, and the *predicate* that relates them.

A function is a special kind of relation that can only be used in one direction (inputs to outputs), and can only have one result. Relations do not have these limitations. In this sense, logic programming generalises functional programming.

While the first functional programming language was Lisp, implemented by John McCarthy's group at MIT in 1958, the first logic programming language was Prolog, implemented by Alain Colmerauer's group at Marseille in 1971.

NOTE

Since the early 1980s, the University of Melbourne has been one of the world's top centers for research in logic programming.

Lee Naish designed and implemented MU-Prolog, and led the development of its successor, NU-Prolog.

Zoltan Somogyi led the development of Mercury, and was one of its main implementors.

The name "Prolog" was chosen by Philippe Roussel as an abbreviation for "programmation en logique", which is French for "programming in logic".

MU-Prolog and NU-Prolog are two closely-related dialects of Prolog. There are many others, since most centers of logic programming research in the 1980s implemented their own versions of the language.

The other main centers of logic programming research are in Leuven, Belgium; Uppsala, Sweden; Madrid, Spain; and Las Cruces, New Mexico, USA.

Relations

A *relation* specifies a relationship; for example, a family relationship. In Prolog syntax,

```
parent(queen_elizabeth, prince_charles).
```

specifies (a small part of the) parenthood relation, which relates parents to their children. This says that Queen Elizabeth is a parent of Prince Charles.

The name of a relation is called a *predicate*. Predicates have no directionality: it makes just as much sense to ask of whom is Queen Elizabeth a parent as to ask who is Prince Charles's parent. There is also no promise that there is a unique answer to either of these questions.

Facts

A statement such as:

```
parent(queen_elizabeth, prince_charles).
```

is called a *fact*. It may take many facts to define a relation:

```
% (A small part of) the British Royal family
parent(queen_elizabeth, prince_charles).
parent(prince_philip, prince_charles).
parent(prince_charles, prince_william).
parent(prince_charles, prince_harry).
parent(princess_diana, prince_william).
parent(princess_diana, prince_harry).
⋮
```

Text between a percent sign (`%`) and end-of-line is treated as a comment.

Using Prolog

Most Prolog systems have an environment similar to GHCi. A file containing facts like this should be written in a file whose name begins with a lowercase letter and contains only letters, digits, and underscores, and ends with "`.pl`".

A source file can be loaded into Prolog by typing its filename (without the `.pl` extension) between square brackets at the Prolog prompt (`?-`). Prolog prints a message to say the file was compiled, and `true` to indicate it was successful (user input looks like this):

```
?- [royals].
% royals compiled 0.00 sec, 8 clauses
true.
\par
\vspace2mm
?-
```

NOTE

Some Prolog GUI environments provide other, more convenient, ways to load code, such as menu items or drag-and-drop.

– section 9 slide 4 –

Queries

Once your code is loaded, you can use or test it by issuing *queries* at the Prolog prompt. A Prolog query looks just like a fact. When written in a source file and loaded into Prolog, it is treated as a true statement. At the Prolog prompt, it is treated as a query, asking if the statement is true.

```
?- parent(prince_charles, prince_william).
true .
\par
\vspace2mm
?- parent(prince_william, prince_charles).
false.
```

– section 9 slide 5 –

Variables

Each predicate argument may be a *variable*, which in Prolog begins with a capital letter or underscore and follows with letters, digits, and underscores. A query containing a variable asks if there exists a value for that variable that makes that query true, and prints the value that makes it true.

If there is more than one answer to the query, Prolog prints them one at a time, pausing to see if more solutions are wanted. Typing semicolon asks for more solutions; just hitting enter (return) finishes without more solutions.

This query asks: *of whom Prince Charles is a parent?*

```
?- parent(prince_charles, X).
X = prince_william ;
X = prince_harry.
```

– section 9 slide 6 –

Multiple modes

The same parenthood relation can be used just as easily to ask who is a parent of Prince Charles or even who is a parent of whom. Each of these is a different *mode*, based on which arguments are *bound* (inputs; non-variables) and which are *unbound* (outputs; variables).

```
?- parent(X, prince_charles).
X = queen_elizabeth ;
X = prince_philip.
?- parent(X, Y).
X = queen_elizabeth,
Y = prince_charles ;
X = prince_philip,
Y = prince_charles ;
⋮
```

– section 9 slide 7 –

Compound queries

Queries may use multiple predicate applications (called *goals* in Prolog and *atoms* in predicate logic). The simplest way to combine multiple goals is to separate them with a comma. This asks Prolog for all bindings for the variables that satisfy both (or all) of the goals. The comma can be read as "and". In relational algebra, this is called an *inner join* (but do not worry if you do not know what that is).

```
?- parent(queen_elizabeth, X), parent(X, Y).
X = prince_charles,
Y = prince_william ;
X = prince_charles,
Y = prince_harry.
```

Rules

Predicates can be defined using *rules* as well as facts. A rule has the form

$$Head \texttt{ :- } Body,$$

where *Head* has the form of a fact and *Body* has the form of a (possibly compound) query. The `:-` is read "if", and the clause means that the *Head* is true if the *Body* is. For example

```
grandparent(X,Z) :- parent(X, Y), parent(Y, Z).
```

means "X is grandparent of Z if X is parent of Y and Y is parent of Z."

Rules and facts are the two different kinds of *clauses*. A predicate can be defined with any number of clauses of either or both kinds, intermixed in any order.

Recursion

Rules can be recursive. Like Haskell, Prolog has no looping constructs, so recursion is widely used. Prolog does not have as well-developed a library of higher-order operations as Haskell, so recursion is used more in Prolog than in Haskell.

A person's ancestors are their parents and the ancestors of their parents.

```
ancestor(Anc, Desc) :-
        parent(Anc, Desc).
ancestor(Anc, Desc) :-
        parent(Parent, Desc),
        ancestor(Anc, Parent).
```


Equality

Equality in Prolog, written "=" and used as an infix operator, can be used both to bind variables and to check for equality. Like Haskell, Prolog is a *single-assignment* language: once bound, a variable cannot be reassigned.

```
?- X = 7.
X = 7.
\par
\vspace2mm
?- a = b.
false.
\par
\vspace2mm
?- X = 7, X = a.
false.
\par
\vspace2mm
?- X = 7, Y = 8, X = Y.
false.
```


Disjunction

Goals can be combined with disjunction (or) as well as conjunction (and). Disjunction is written ";" and used as an infix operator. Conjunction (",") has higher precedence (binds tighter) than disjunction, but parentheses can be used to achieve the desired precedence.

Who are the children of Queen Elizabeth or Princess Diana?

```
?- parent(queen_elizabeth, X) ; parent(princess_diana, X).
X = prince_charles ;
X = prince_william ;
X = prince_harry.
```

Negation

Negation in Prolog is written "`\+`" and used as a prefix operator. Negation has higher (tighter) precedence than both conjunction and disjunction. Be sure to leave a space between the `\+` and an open parenthesis.

Who are the parents of Prince William other than Prince Charles?

```
?- parent(X, prince_william), \+ X = prince_charles.
X = princess_diana.
```

Disequality in Prolog is written as an infix "`\=`". `X \= Y` is the same as `\+ X = Y`.

```
?- parent(X, prince_william), X \= prince_charles.
X = princess_diana.
```


The Closed World Assumption

Prolog assumes that all true things can be derived from the program. This is called the *closed world assumption*. Of course, this is not true for our `parent` relation (that would require tens of billions of clauses!).

```
?- \+ parent(queen_elizabeth, princess_anne).
true.
```

but Princess Anne *is* a daughter of Queen Elizabeth. Our program simply does not know about her.

So use negation with great care on predicates that are not complete, such as `parent`.

Negation as failure

Prolog executes `\+`G by first trying to prove G. If this fails, then `\+`G succeeds; if it succeeds, then `\+`G fails. This is called *negation as failure*.

In Prolog, failing goals can never bind variables, so any variable bindings made in solving G are thrown away when `\+`G fails. Therefore, `\+`G cannot solve for any variables, and goals such as these cannot work properly.

Is there anyone of whom Queen Elizabeth is not a parent?
Is there anyone who is not Queen Elizabeth?

```
?- \+ parent(queen_elizabeth, X).
false.
\par
\vspace2mm
?- X \= queen_elizabeth.
false.
```


Execution Order

The solution to this problem is simple: ensure all variables in a negated goal are bound before the goal is executed.

Prolog executes goals in a query (and the body of a clause) from first to last, so put the goals that will bind the variables in a negation before the negation (or `\=`).

In this case, we can generate all people who are either parents or children, and ask whether any of them is different from Queen Elizabeth.

```
?- (parent(X,_) ; parent(_,X)), X \= queen_elizabeth.
X = prince_philip ;
⋮
```


Datalog

The fragment of Prolog discussed so far, which omits data structures, is called *Datalog*. It is a generalisation of what is provided by relational databases. Many modern databases now provide Datalog features or use Datalog implementation techniques.

```
capitol(australia, canberra).
capitol(france, paris).
⋮
continent(australia, australia).
continent(france, europe).
⋮
population(australia, 22_680_000).
population(france, 65_700_000).
⋮
```


Datalog Queries

```
What is the capitol of France?
?- capitol(france, Capitol).
Capitol = paris.
\par
\vspace2mm
What are capitols of European countries?
?- continent(Country, europe), capitol(Country, Capitol).
Country = france,
Capitol = paris.
\par
\vspace2mm
What European countries have populations > 50,000,000?
?- continent(Country, europe), population(Country, Pop),
|  Pop > 50_000_000.
Country = france,
Pop = 65700000.
```


Section 10: Beyond Datalog

Terms

In Prolog, all data structures are called *terms*. A term can be *atomic* or *compound*, or it can be a variable. Datalog has only atomic terms and variables.

Atomic terms include integers and floating point numbers, written as you would expect, and atoms.

An atom begins with a lower case letter and follows with letters, digits and underscores, for example `a`, `queen_elizabeth`, or `banana`.

An atom can also be written beginning and ending with a single quote, and have any intervening characters. The usual character escapes can be used, for example `\n` for newline, `\t` for tab, and `\'` for a single quote. For example: `'Queen Elizabeth'` or `'Hello, World!\n'`.

Compound Terms

In the syntax of Prolog, each compound term is a *functor* (sometimes called *function symbol*) followed by zero or more arguments; if there are any arguments, they are shown in parentheses, separated by commas. Functors are Prolog's equivalent of data constructors, and have the same syntax as atoms.

For example, the small tree that in Haskell syntax would be written as

```
Node Leaf 1 (Node Leaf 2 Leaf)
```

would be written in Prolog syntax as the term

```
node(leaf, 1, node(leaf, 2, leaf))
```

Because Prolog is dynamically typed, each argument of a term can be *any* term, and there is no need to declare types.

Prolog has special syntax for some functors, such as infix notation.

– section 10 slide 2 –

Variables

A variable is also a term. It denotes a single unknown term.

A variable name begins with an upper case letter or underscore, followed by any number of letters, digits, and underscores.

A single underscore `_` is special: it specifies a different variable each time it appears, much like `_` in Haskell pattern matching.

Like Haskell, Prolog is a *single-assignment* language: a variable can only be bound (assigned) once.

Because the arguments of a compound term can be any terms, and variables are terms, variables can appear in terms.

For example `f(A,A)` denotes a term whose functor is `f` and whose two arguments can be anything, as long as they are the same; `f(_,_)` denotes a term whose functor is `f` and has any two arguments.

– section 10 slide 3 –

List syntax

Like Haskell, Prolog has a special syntax for lists.

Both denote the empty list by `[]`.

Both denote the list with the three elements 1, 2 and 3 by `[1, 2, 3]`.

While Haskell uses `x:xs` to denote a list whose head is `x` and whose tail is `xs`, the Prolog syntax is `[X | Xs]` (not to be confused with list comprehensions, which Prolog lacks).

The Prolog syntax for what Haskell would represent with `x1:x2:xs` is `[X1, X2 | Xs]`.

– section 10 slide 4 –

Ground vs nonground terms

A term is a *ground term* if it contains no variables, and it is a *nonground term* if it contains at least one variable.

`3` and `f(a, b)` are ground terms.

Since `Name` and `f(a, X)` each contain at least one variable, they are *nonground* terms.

– section 10 slide 5 –

Substitutions

A *substitution* is a mapping from variables to terms.

Applying a substitution to a term means consistently replacing all occurrences of each variable in the map with the term it is mapped to.

Note that a substitution only replaces variables, never atomic or compound terms.

For example, applying the substitution $\{$`X1` $\mapsto$ `leaf`, `X2` $\mapsto$ `1`, `X3` $\mapsto$ `leaf`$\}$ to the term `node(X1,X2,X3)` yields the term `node(leaf,1,leaf)`.

Since you can get `node(leaf,1,leaf)` from `node(X1,X2,X3)` by applying a substitution to it, `node(leaf,1,leaf)` is an *instance* of `node(X1,X2,X3)`.

Any ground Prolog term has only one instance, while a nonground Prolog terms has an infinite number of instances.

– section 10 slide 6 –

Unification

The term that results from applying a substitution θ to a term t is denoted $t\theta$.

A term u is therefore an instance of term t if there is some substitution θ such that $u = t\theta$.

A substitution θ *unifies* two terms t and u if $t\theta = u\theta$.

Consider the terms `f(X, b)` and `f(a, Y)`.

Applying a substitution $\{\texttt{X} \mapsto \texttt{a}\}$ to those two terms yields `f(a, b)` and `f(a, Y)`, which are not syntactically identical, so this substitution is not a unifier.

On the other hand, applying the substitution $\{\texttt{X} \mapsto \texttt{a}, \texttt{Y} \mapsto \texttt{b}\}$ to those terms yields `f(a, b)` in both cases, so this substitution *is* a unifier.

Recognising proper lists

A proper list is either empty (`[]`) or not (`[X|Y]`), in which case, the tail of the list must be a proper list. We can define a predicate to recognise these.

```
proper_list([]).
proper_list([Head|Tail]) :-
        proper_list(Tail).
```

```
?- [list].
Warning: list.pl:3:
        Singleton variables: [Head]
% list compiled 0.00 sec, 1 clauses
true.
```

Detour: singleton variables

```
Warning: list.pl:3:
        Singleton variables: [Head]
```

The variable `Head` appears only once in this clause:

```
proper_list([Head|Tail]) :-
        proper_list(Tail).
```

This often indicates a typo in the source code. For example, if `Tail` were spelled `Tial` in one place, this would be easy to miss. But Prolog's singleton warning would alert us to the problem.

Detour: singleton variables

In this case, there is no problem; to avoid the warning, we should begin the variable name `Head` with an underscore, or just name the variable `_`.

```
proper_list([]).
proper_list([_Head|Tail]) :-
        proper_list(Tail).
```

```
?- [list].
% list compiled 0.00 sec, 1 clauses
true.
```

General programming advice: always fix compiler warnings (if possible). Some warnings may indicate a real problem, and you will not see them if they're lost in a sea of unimportant warnings. It is easier to fix a problem when the compiler points it out than when you have to find it yourself.

Append

Appending two lists is a common operation in Prolog. This is a built in predicate in most Prolog systems, but could easily be implemented as:

```
append([], C, C).
append([A|B], C, [A|BC]) :-
        append(B, C, BC).
```

```
?- append([a,b,c],[d,e],List).
List = [a, b, c, d, e].
```

This is similar to `++` in Haskell.

`append` is like `proper_list`

Compare the code for `proper_list` to the code for `append`:

```
proper_list([]).
proper_list([Head|Tail]) :-
        proper_list(Tail).

append([], C, C).
append([A|B], C, [A|BC]) :-
        append(B, C, BC).
```

This is common: code for a predicate that handles a term often follows the structure of that term (as we saw in Haskell).

While the `proper_list` predicate is not very useful itself, it was worth designing, as it gives a hint at the structure of other code that traverses lists. Since types are not declared in Prolog, predicates like `proper_list` can serve to indicate the notional type.

Appending backwards

Unlike `++` in Haskell, append in Prolog can work in other modes:

```
?- append([1,2,3], Rest, [1,2,3,4,5]).
Rest = [4, 5].
?- append(Front, [3,4], [1,2,3,4]).
Front = [1, 2] ;
false.
?- append(Front,Back,[a,b,c]).
Front = [],
Back = [a, b, c] ;
Front = [a],
Back = [b, c] ;
Front = [a, b],
Back = [c] ;
Front = [a, b, c],
Back = [] ;
false.
```

Length

The `length` built-in predicate relates a list to its length:

```
?- length([a,b,c], Len).
Len = 3.
\par
\vspace2mm
?- length(List, 3).
List = [_G273, _G276, _G279].
```

The `_G...` terms are how Prolog prints out unbound variables. The number part reflects when the variable was created; because these variables are all printed differently, we can tell they are all distinct variables.

`[_G273, _G276, _G279]` is a list of three distinct unbound variables, and each unbound variable can be any term, so this can be any three-element list, as specified by the query.

Putting them together

How would we implement `take` in Prolog?

`take(N,List,Front)` should hold if `Front` is the first `N` elements of `List`. So `length(Front,N)` should hold.

Also, `append(Front, _, List)` should hold. Then:

```
take(N, List, Front) :-
        length(Front,N),
        append(Front, _, List).
```

Prolog coding hint: sometimes it is easier to write code if you think about *checking* if the result is correct rather than *computing* it. That is, *think declaratively*.

Then you need to think about whether your code will work the ways you want it to. We will return to that.

Member

Here is list membership, two ways:

```
member1(Elt, List) :- append(_,[Elt|_], List).
\par
\vspace2mm
member2(Elt, [Elt|_]).
member2(Elt, [_|Rest]) :- member2(Elt, Rest).
```

These behave the same, but the second is a bit more efficient because the first builds and ignores the list of elements before `Elt` in `List`, and the second does not.

Note the recursive version does not exactly match the structure of our earlier `proper_list` predicate. This is because `Elt` is never a member of the empty list, so we do not need a clause for `[]`. In Prolog, we do not need to specify when a predicate should fail; only when it should succeed. We also have two cases to consider when the list is non-empty (like Haskell in this respect).

Arithmetic

In Prolog, terms like `6 * 7` are just data structures, and `=` does not evaluate them, it just unifies them.

The built-in predicate `is` (an infix operator) evaluates expressions.

```
?- X = 6 * 7.
X = 6*7.
\par
\vspace2mm
?- X is 6 * 7.
X = 42.
```


Arithmetic modes

Use `is/2` to evaluate expression

```
square(N, N2) :- N2 is N * N.
```

Unfortunately, `square` only works when the first argument is bound. This is because `is` only works if its second argument is ground.

```
?- square(5, X).
X = 25.
?- square(X, 25).
ERROR: is/2: Arguments are not sufficiently instantiated
?- 25 is X * X.
ERROR: is/2: Arguments are not sufficiently instantiated
```

Later we shall see how to write code to do arithmetic in different modes.

Arithmetic

Prolog provides the usual arithmetic operators, including:

`+ - *`	add, subtract, multiply
`/`	division (may return a float)
`//`	integer division (rounds toward 0)
`mod`	modulo (result has same sign as second argument
`-`	unary minus (negation)
`integer float`	coersions (not operators)

More arithmetic predicates (infix operators; both arguments must be ground expressions):

`< =<`	less, less or equal (note!)
`> >=`	greater, greater or equal
`=:= =\=`	equal, not equal (only numbers)

Section 11: Understanding and Debugging Prolog code

List Reverse

To reverse a list, put the first element of the list at the end of the reverse of the tail of the list.

```
rev1([], []).
rev1([A|BC], CBA) :-
        rev1(BC, CB),
        append(CB, [A], CBA).
```

`reverse/2` is an SWI Prolog built-in, so we use a different name to avoid conflict.

NOTE This version of `reverse/2` is called *naive* reverse, because it has quadratic complexity. We'll see a more efficient version a little later.

List Reverse

The *mode* of a Prolog goal says which arguments are bound and which are bound (inputs) and which are unbound (outputs).

`rev1/2` works as intended when the first argument is ground and the second is free, but not for the opposite mode.

```
?- rev1([a,b,c], Y).
Y = [c, b, a].
\par
\vspace2mm
?- rev1(X, [c,b,a]).
X = [a, b, c] ;
```

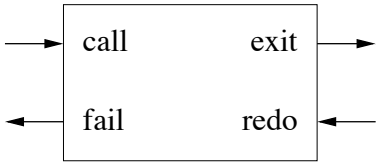
Prolog hangs at this point. We will use the Prolog debugger to understand why. For now, hit control-C and then 'a' to abort.

The Prolog Debugger

To understand the debugger, you will need to understand the Byrd box model. Think of goal execution as a box with port for each way to enter and exit.

A conventional language has only one way to enter and one way to exit; Prolog has two of each.

The four debugger ports are:



call initial entry **exit** successful completion

fail final failure **redo** backtrack into goal

Turn on debugger with `trace`, and off with `nodebug`, at the Prolog prompt.

Using the Debugger

The debugger prints the current port, execution depth, and goal (with the current variable bindings) at each step.

```
?- trace, rev1([a,b], Y).
   Call: (7) rev1([a, b], _G12717) ? creep
   Call: (8) rev1([b], _G12834) ? creep
   Call: (9) rev1([], _G12834) ? creep
   Exit: (9) rev1([], []) ? creep
   Call: (9) lists:append([], [b], _G12838) ? creep
   Exit: (9) lists:append([], [b], [b]) ? creep
   Exit: (8) rev1([b], [b]) ? creep
   Call: (8) lists:append([b], [a], _G12717) ? creep
   Exit: (8) lists:append([b], [a], [b, a]) ? creep
   Exit: (7) rev1([a, b], [b, a]) ? creep
Y = [b, a].
```

"`lists:`" in front of `append` is a module name.

Reverse backward

Now try the "backwards" mode of `rev1/2`. We shall use a smaller test case to keep it manageable.

```
?- trace, rev1(X, [a]).
   Call: (7) rev1(_G11553, [a]) ? creep
   Call: (8) rev1(_G11661, _G11671) ? creep
   Exit: (8) rev1([], []) ? creep
   Call: (8) lists:append([], [_G11660], [a]) ? creep
   Exit: (8) lists:append([], [a], [a]) ? creep
   Exit: (7) rev1([a], [a]) ? creep
X = [a] ;
```

Reverse backward, continued

after showing the first solution, Prolog goes on forever like this:

```
Redo: (8) rev1(_G11661, _G11671) ? creep
Call: (9) rev1(_G11664, _G11674) ? creep
Exit: (9) rev1([], []) ? creep
Call: (9) lists:append([], [_G11663], _G11678) ? creep
Exit: (9) lists:append([], [_G11663], [_G11663]) ? creep
Exit: (8) rev1([_G11663], [_G11663]) ? creep
Call: (8) lists:append([_G11663], [_G11660], [a]) ? creep
Fail: (8) lists:append([_G11663], [_G11660], [a]) ? creep
Redo: (9) rev1(_G11664, _G11674) ? creep
Call: (10) rev1(_G11667, _G11677) ? creep
Exit: (10) rev1([], []) ? creep
Call: (10) lists:append([], [_G11666], _G11681) ? creep
Exit: (10) lists:append([], [_G11666], [_G11666]) ? creep
Exit: (9) rev1([_G11666], [_G11666]) ? creep
Call: (9) lists:append([_G11666], [_G11663], _G11684) ? cr
Exit: (9) lists:append([_G11666], [_G11663], [_G11666, _G1
Exit: (8) rev1([_G11663, _G11666], [_G11666, _G11663]) ? c
Call: (8) lists:append([_G11666, _G11663], [_G11660], [a])
Fail: (8) lists:append([_G11666, _G11663], [_G11660],
.
.
.
```

Infinite backtracking loop

```
rev1([], []).
rev1([A|BC], CBA) :-
        rev1(BC, CB),
        append(CB, [A], CBA).
```

The problem is that the goal `rev1(X,[a])`, resolves to the goal `rev1(BC, CB), append(CB, [A], [a])`. The call `rev1(BC, CB)` produces an infinite backtracking sequence of solutions $\{$`BC` $\mapsto$ $[], $`CB` $\mapsto$ $[]\}$, $\{$`BC` $\mapsto$ $[Z], $`CB` $\mapsto$ $[Z]\}$, $\{$`BC` $\mapsto$ $[Y, Z], $`CB` $\mapsto$ $[Z, Y]\}, \ldots$. For each of these solutions, we call `append(CB, [A], [a])`.

`append([], [A], [a])` succeeds, with $\{$`A` $\mapsto$ $[a]\}$. However, `append([Z], [A], [a])` fails, as does this goal for all following solutions for `CB`. This is an infinite backtracking loop.

Infinite backtracking loop

We could fix this problem by executing the body goals in the other order:

```
rev2([], []).
rev2([A|BC], CBA) :-
        append(CB, [A], CBA),
        rev2(BC, CB).
```

But this definition does not work in the forward direction:

```
?- rev2(X, [a,b]).
X = [b, a] ;
false.
\par
\vspace2mm
?- rev2([a,b], Y).
Y = [b, a] ;
^CAction (h for help) ? abort
% Execution Aborted
```

NOTE

Prolog uses a fixed left-to-right execution order, so neither order works for both directions.

Working in both directions

The solution is to ensure that when `rev1` is called, the first argument is always bound to a list. We do this by observing that the length of a list must always be the same as that of its reverse. When `samelength/2` succeeds, both arguments are bound to lists of the same fixed length.

```
rev3(ABC, CBA) :-
        samelength(ABC, CBA),
        rev1(ABC, CBA).
\par
\vspace2mm
samelength([], []).
samelength([_|Xs], [_|Ys]) :-
          samelength(Xs, Ys).
```

```
?- rev3(X, [a,b]).
X = [b, a].
\par
\vspace2mm
?- rev3([a,b], Y).
Y = [b, a].
```

More on the Debuger

Some useful debugger commands:

h display debugger help

c creep to the next port (also space, enter)

s skip over goal; go straight to exit or fail port

r back to initial call port of goal, undoing all bindings done since starting it;

a abort whole debugging session

+ set spypoint (like breakpoint) on this pred

- remove spypoint from this predicate

l leap to the next spypoint

b pause this debugging session and enter a "break level," giving a new Prolog prompt; end of file reenters debugger

---

More on the Debuger

Built-in predicates for controlling the debugger:

`spy(Predspec)` Place a spypoint on Predspec, which can be a `Name/Arity` pair, or just a predicate name.

`nospy(Predspec)` Remove the spypoint from Predspec.

`trace` Turn on the debugger

`debug` Turn on the debugger and leap to first spypoint

`nodebug` Turn off the debugger

A "Predspec" is a predicate *name* or *name/arity*

Using the debugger

Note the `r` (retry) debugger command restarts a goal from the beginning, "time travelling" back to the time when starting to execute that goal.

The s (skip) command skips forward in time, over the whole execution of a goal, to its exit or fail port.

This leads to a quick way of tracking down most bugs:

1. When you arrive at a call or redo port: skip.
2. If you come to an exit port with the correct results (or a correct fail port): creep.
3. If you come to an incorrect exit or fail port: retry, then creep.

Eventually you will find a clause that has the right input and wrong output (or wrong failure); this is the bug. This will not help find infinite recursion, though.

– section 11 slide 12 –

Spypoints

For larger computations, it may take some time to get to the part of the computation where the bug lies. Usually, you will have a good idea, or at least a few good guesses, which predicates you suspect of being buggy (usually the predicates you have edited most recently). In cases of infinite recursion you may suspect certain predicates of being involved in the loop.

In these cases, spypoints will be helpful. Like a breakpoint in most debuggers, when Prolog reaches any port of a predicate with a spypoint set, Prolog stops and shows the port. The `l` (leap) command tells Prolog to run quietly until it reaches a spypoint. Use the `spy`(*pred*) goal at the Prolog prompt to set a spypoint on the named predicate, `nospy`(*pred*) to remove one. You can also add a spypoint on the predicate of the current debugger port with the `+` command, and remove it with `-`.

– section 11 slide 13 –

Managing nondeterminism

This is a common mistake in defining factorial:

```
fact(0, 1).
fact(N, F) :-
        N1 is N - 1,
        fact(N1, F1),
        F is N * F1.
```

```
?- fact(5, F).
F = 120 ;
ERROR: Out of local stack
```

`fact(5,F)` has only one solution, why was Prolog looking for another?

– section 11 slide 14 –

Correctness

The second clause promises that for all n, $n! = n \times (n-1)!$. This is wrong for $n < 1$.

Prolog is not like Haskell: even if one clause applies, later clauses are still tried. After finding $0! = 1$, Prolog thinks $0! = 0 \times -1!$; tries to compute $-1!, -2!, \ldots$

The simple solution is to ensure each clause is a correct (part of the) definition.

```
fact(0, 1).
fact(N, F) :-
        N > 0,
        N1 is N - 1,
        fact(N1, F1),
        F is F1 * N.
```

– section 11 slide 15 –

Avoiding the choicepoint

This definition is correct, but it could be more efficient.

Prolog's indexing scheme only works where the functors of the first arguments of the clause heads are distinct non-variables. Since the first argument of the second clause is a variable, Prolog will leave a choicepoint when selecting the first clause.

Backtracking to the second clause will fail unless $N > 0$. This test is quick, however, as long as the choicepoint exists, it inhibits the very important last call optimisation (discussed later). Therefore, where efficiency matters, it is important to make your recursive predicates not leave choicepoints when they should be deterministic.

In this case, $N = 0$ and $N > 0$ are mutually exclusive, so at most one clause can apply, so `fact/2` should not leave a choicepoint.

NOTE Actually, you will probably never take the factorial of a very large number, so the loss in efficiency from this definition is unlikely to make a detectable difference. For deeper arithmetic recursions or other recursions that are not structural inductions, however, this technique is useful.

If-then-else

We can avoid the choicepoint with Prolog's if-then-else construct:

```
fact(N, F) :-
        ( N =:= 0 ->
            F = 1
        ;   N > 0,
            N1 is N - 1,
            fact(N1, F1),
            F is F1 * N
        ).
```

The `->` is treated like a conjunction (`,`), except that when it is crossed, any alternative solutions of the goal before the `->`, as well as any alternatives following the `;` are forgotten. Conversely, if the goal before the `->` fails, then the goal after the `;` is tried. So this is deterministic whenever both the code between `->` and `;`, and the code after the `;`, are.

If-then-else caveats

However, you should prefer indexing and avoid if-then-else, when you have a choice. If-then-else usually leads to code that will not work smoothly in multiple modes. For example, append could be written with if-then-else:

```
ap(X, Y, Z) :-
        ( X = [] ->
            Z = Y
        ;   X = [U|V],
            ap(V, Y, W),
            Z = [U|W]
        ).
```

This may appear correct, and may follow the logic you would use to code it in another language, but it is not appropriate for Prolog.

If-then-else caveats

With that definition of `ap`:

```
?- ap([a,b,c], [d,e], L).
L = [a, b, c, d, e].
\par
\vspace2mm
?- ap(L, [d,e], [a,b,c,d,e]).
false.
\par
\vspace2mm
?- ap(L, M, [a,b,c,d,e]).
L = [],
M = [a, b, c, d, e].
```

Because the if-then-else commits to binding the first argument to `[]` when it can, this version of append will not work correctly unless the first argument is bound when append is called.

Section 12:
Logic and Resolution

Interpretations

In the mind of the person writing a logic program,

- each constant (function symbol of zero arity) stands for an entity in the world of the program;
- each function (function symbol of arity n where $n > 0$) stands for a function from n entities to one entity in the world of the program; and
- each predicate of arity n stands for a particular relationship between n entities in the world of the program.

This mapping from the symbols in the program to the world of the program (which may be the real world or some imagined world) is called an *interpretation*.

The obvious interpretation of the atomic formula `parent(queen_elizabeth, prince_charles)` is that Queen Elizabeth II is a parent of Prince Charles, but other interpretations are also possible.

NOTE Another interpretation would use the function symbol `queen_elizabeth` to refer to George W Bush, the function symbol `prince_charles` to refer to Barack Obama, and the predicate symbol `parent` to refer to the notion "succeeded by as US President". However, any programmer using this interpretation, or pretty much any interpretation other than the obvious one, would be guilty of using a horribly misleading programming style.

Terms using non-meaningful names such as `f(g, h)` do not lead readers to expect a particular interpretation, so these can have many different non-misleading interpretations.

Two views of predicates

As the name implies, the main focus of the predicate calculus is on *predicates*.

You can think of a predicate with n arguments in two equivalent ways.

- You can view the predicate as a function from all possible combinations of n terms to a truth value (i.e. true or false).
- You can view the predicate as a set of tuples of n terms. Every tuple in this set is implicitly mapped to true, while every tuple not in this set is implicitly mapped to false.

The task of a predicate definition is to define the mapping in the first view, or equivalently, to define the set of tuples in the second view.

The meaning of clauses

The meaning of the clause

```
grandparent(A, C) :-
    parent(A, B),
    parent(B, C).
```

is: for all the terms that `A` and `C` may stand for, `A` is the grandparent of `C` if there is a term `B` such that `A` is the parent of `B` and `B` is the parent of `C`.

In mathematical notation:

$$\forall A \forall C : grandparent(A, C) \leftarrow$$
$$\quad \exists B : parent(A, B) \wedge parent(B, C)$$

The variables appearing in the head are universally quantified over the entire clause, while variables appearing only in the body are existentially quantified over the body.

NOTE $\forall$ is "forall", the universal quantifier, while $\exists$ is "there exists", the existential quantifier. The sign $\wedge$ denotes the logical "and" operation, while the sign $\vee$ denotes the logical "or" operation, and the sign $\neg$ denotes the logical "not" operation.

– section 12 slide 3 –

The meaning of predicate definitions

A predicate is defined by a finite number of clauses, each of which is in the form of an implication. A fact such as `parent(queen_elizabeth, prince_charles)` represents this implication:

$$\forall A \forall B : parent(A, B) \leftarrow \\ (A = queen\_elizabeth \wedge B = prince\_charles)$$

To represent the meaning of the predicate, create a disjunction of the bodies of all the clauses:

$$\forall A \forall B : parent(A, B) \leftarrow \\ (A = queen\_elizabeth \wedge B = prince\_charles) \ \vee \\ (A = prince\_philip \wedge B = prince\_charles) \ \vee \\ (A = prince\_charles \wedge B = prince\_william) \ \vee \\ (A = prince\_charles \wedge B = prince\_harry) \ \vee \\ (A = princess\_diana \wedge B = prince\_william) \ \vee \\ (A = princess\_diana \wedge B = prince\_harry)$$

NOTE Obviously, this definition of the parent relationship is the correct one only if you restrict the universe of discourse to this small set of people.

– section 12 slide 4 –

The closed world assumption

To implement the *closed world assumption*, we only need to make the implication arrow go both ways (*if and only if*):

$$\forall A \forall B : parent(A, B) \leftrightarrow \\ (A = queen\_elizabeth \wedge B = prince\_charles) \ \vee \\ (A = prince\_philip \wedge B = prince\_charles) \ \vee \\ (A = prince\_charles \wedge B = prince\_william) \ \vee \\ (A = prince\_charles \wedge B = prince\_harry) \ \vee \\ (A = princess\_diana \wedge B = prince\_william) \ \vee \\ (A = princess\_diana \wedge B = prince\_harry)$$

This means that A is not a parent of B unless they are one of the listed cases.

Adding the reverse implication this way creates the *Clark completion* of the program.

– section 12 slide 5 –

Semantics of logic programs

A logic program P consists of a set of predicate definitions. The *semantics* of this program (its *meaning*) is the set of its *logical consequences* as ground atomic formulas.

A ground atomic formula a is a logical consequence of a program P if P makes a true.

A negated ground atomic formula $\neg a$, written in Prolog as `\+ a`, is a logical consequence of P if a is not a logical consequence of P.

For most logic programs, the set of ground atomic formulas it entails is infinite (as is the set it does not entail). As logicians, we do not worry about this any more than a mathematician worries that there are an infinite number of solutions to $a + b = c$.

– section 12 slide 6 –

Finding the semantics

You can find the semantics of a logic program by working backwards. Instead of reasoning from a query to find a satisfying substitution, you reason from the program to find what ground queries will succeed.

The immediate consequence operator T_P takes a set of ground unit clauses C and produces the set of ground unit clauses implied by C together with the program P.

This always includes all ground instances of all unit clauses in P. Also, for each clause $H :- G_1, \ldots, G_a.$ in P, if C contains instances of $G_1, \ldots G_n$, then the corresponding instance of H is also in the result.

Eg, if $P = \{q(X, Z) :- p(X, Y), p(Y, Z)\}$ and $C = \{p(a, b).p(b, c).p(c, d).\}$, then $T_P(C) = \{q(a, c).q(b, d).\}$.

The semantics of program P is always $T_P(T_P(T_P(\cdots(\varnothing)\cdots)))$
(T_P applied infinitely many times to the empty set).

– section 12 slide 7 –

Procedural Interpretation

The *logical* reading of the clause

```
grandparent(X, Z) :-
    parent(X, Y), parent(Y, Z).
```

says "*for all* X, Y, Z, if X is parent of Y and Y is parent of Z, then X is grandparent of Z".

The *procedural* reading says "to show that X is a grandparent of Z, it is sufficient to show that X is a parent of Y and Y is a parent of Z".

SLD resolution, used by Prolog, implements this strategy.

SLD Resolution

The consequences of a logic program are determined through a simple but powerful deduction strategy called *resolution*.

SLD resolution is an efficient version of resolution. The basic idea is: given this program, to show this goal is true

```
q :- b1a, b1b.
q :- b2a, b2b.
⋮
```

```
?- p, q, r.
```

it is sufficient to show any of

```
?- p, b1a, b1b, r.
?- p, b2a, b2b, r.
⋮
```


SLD resolution in action

E.g., to determine if Queen Elizabeth is Prince Harry's grandparent:

```
?- grandparent(queen_elizabeth, prince_harry).
```

with this program

```
grandparent(X, Z) :-
        parent(X, Y), parent(Y, Z).
```

we unify query goal `grandparent(queen_elizabeth, prince_harry)` with clause head `grandparent(X, Z)`, apply the resulting substitution to the clause, yielding the *resolvent*. Since the goal is identical to the resolvent head, we can replace it with the resolvent body, leaving:

```
?- parent(queen_elizabeth, Y), parent(Y, prince_harry).
```


SLD resolution can fail

Now we must pick one of these goals to resolve; we select the second.

The program has several clauses for `parent`, but only two can successfully resolve with `parent(Y, prince_harry)`:

```
parent(prince_charles, prince_harry).
parent(princess_diana, prince_harry).
```

We choose the second. After resolution, we are left with the query (note the unifying substitution is applied to both the selected clause and the query):

```
?- parent(queen_elizabeth, princess_diana).
```

No clause unifies with this query, so resolution fails. Sometimes, it may take many resolution steps to fail.

SLD resolution can succeed

Selecting the second of these matching clauses led to failure:

```
parent(prince_charles, prince_harry).
parent(princess_diana, prince_harry).
```

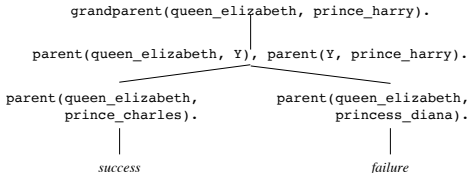
This does not mean we are through: we must *backtrack* and try the first matching clause. This leaves

```
?- parent(queen_elizabeth, prince_charles).
```

There is one matching program clause, leaving nothing more to prove. The query succeeds.

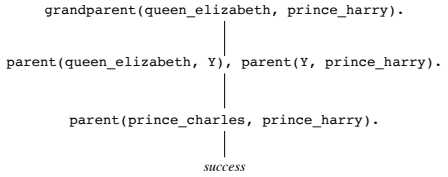
Resolution

This derivation can be shown as an SLD tree:



Order of execution

The order in which goals are resolved and the order in which clauses are tried does not matter for correctness (in pure Prolog), but it does matter for efficiency. In this example, resolving `parent(queen_elizabeth, Y)` before `parent(Y, prince_harry)` is more efficient, because there is only one clause matching the former, and two matching the latter.



SLD resolution in Prolog

At each resolution step we must make two decisions:

1. which goal to resolve
2. which clauses matching the selected goal to pursue

(though there may only be one choice for either or both).

Our procedure was somewhat haphazard when decisions needed to be made. For pure logic programming, this does not matter for correctness. All goals will need to be resolved eventually; which order they are resolved in does not change the answers. All matching clauses may need to be tried; the order in which we try them determines the order solutions are found, but not which solutions are found.

Prolog always selects the first goal to resolve, and always selects the first matching clause to pursue first. This gives the programmer more certainty, and control, over execution.

Backtracking

When there are multiple clauses matching a goal, Prolog must remember which one to go back to if necessary. It must be able to return the computation to the state it was in when the first matching clause was selected, so that it can return to that state and try the next matching clause. This is all done with a *choicepoint*.

When a goal fails, Prolog *backtracks* to the most recent choicepoint, removing all variable bindings made since the choicepoint was created, returning those variables to their unbound state. Then Prolog begins resolution with the next matching clause, repeating the process until Prolog detects that there are no more matching clauses, at which point it removes that choicepoint. Subsequent failures will then backtrack to the next most recent choicepoint.

Indexing

Indexing can greatly improve Prolog efficiency

Most Prolog systems will automatically create an index for a predicate such as `parent/2` (Prolog uses *name/arity* to refer to predicates) with multiple clauses the heads of which have distinct constants or functors. This means that, for a call with the first argument bound, Prolog will immediately jump to the first clause that matches. If backtracking occurs, the index allows Prolog to jump straight to the next clause that matches, and so on.

If the first argument is unbound, then all clauses will have to be tried.

Indexing

If some clauses have variables in the first argument of the head, those clauses will be tried at the appropriate time regardless of the call. Indexing changes performance, not behaviour. Consider:

```
p(a, z).
p(b, y).
p(X, X).
p(a, x).
```

For the call `p(I, J)`, all clauses will be tried, in order. For `p(a, J)`, the first clause will be tried, then the third, then fourth. For `p(b, J)`, the second, then third, clause will be tried. For `p(c, J)`, only the third clause will be tried.

Indexing

Some Prolog systems, such as SWI Prolog, will construct indices for arguments other than the first. For `parent/2`, SWI Prolog will index on both arguments, so finding the children of a parent or parents of a child both benefit from indexing.

Just as important as jumping directly to the first matching clause, indexing tells Prolog when no further clauses could possibly match the goal, allowing it to remove the choicepoint, or even to avoid creating the choicepoint in the first place. Even with only two clauses, such as for `append/3`, indexing can substantially improve performance.

Section 13: Tail Recursion

Tail recursion

A predicate (or function, or procedure, or method, or...) is *tail recursive* if the only recursive call on any execution of that predicate is the last code executed before returning to the caller. For example, the usual definition of `append/3` is tail recursive, but `rev1/2` is not:

```
append([], C, C).
append([A|B], C, [A|BC]) :-
        append(B, C, BC).
\par
\vspace2mm
rev1([], []).
rev1([A|BC], CBA) :-
        rev1(BC, CB),
        append(CB, [A], CBA).
```


Tail recursion optimisation

Like most declarative languages, Prolog performs *tail recursion optimisation (TRO)*. This is important for declarative languages, since they use recursion more than non-declarative languages. TRO makes recursive predicates behave as if they were loops.

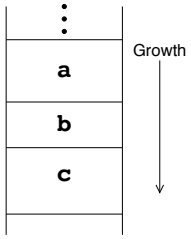
Note that TRO is more often directly applicable in Prolog than other languages because more Prolog code is tail recursive. For example, while `append/3` in Prolog is tail recursive, `++` in Haskell is not, because the last operation performed is `(:)`, not `++`.

```
[] ++ lst = lst
(h:t) ++ lst = h:(t++lst)
```

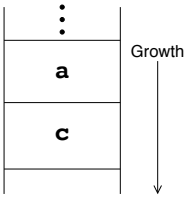
However another optimisation can permit TRO for this code.

The stack

To understand TRO, it is important to understand how programming languages (not just Prolog) implement call and return using a stack (Haskell is a rare exception). While `a` is executing, it stores its local variables, and where to return to when finished, in a *stack frame* or *activation record*. When `a` calls `b`, it creates a fresh stack frame for `b`'s local variables and return address, preserving `a`'s frame, and similarly when `b` calls `c`, as shown below.



But if all `b` will do after calling `c` is return to `a`, then there is no need to preserve its local variables. Prolog can release `b`'s frame before calling `c`, as shown below. When `c` is finished, it will return directly to `a`. This is called *last call optimisation*, and can save significant stack space.

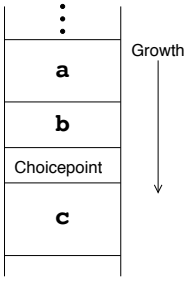


TRO and choicepoints

Tail recursion optimisation is a special case of last call optimisation where the last call is recursive. This is

especially beneficial, since recursion is used to replace looping. Without TRO, this would require a stack frame for each iteration, and would quickly exhaust the stack. With TRO, tail recursive predicates execute in constant stack space, just like a loop.

However, if `b` leaves a choicepoint, it sits on the stack above `b`'s frame, "freezing" that and all earlier frames so that they are not reclaimed. This is necessary because when Prolog backtracks to that choicepoint, `b`'s arguments must be ready to try the next matching clause for `b`. The same is true if `c` or any predicate called later leaves a choicepoint, but choicepoints before the call to `b` do not interfere.



Making code tail recursive

Our factorial predicate was not tail recursive, as the last thing it does is perform arithmetic.

```
fact(N, F) :-
        ( N =:= 0 ->
            F = 1
        ;   N > 0,
            N1 is N - 1,
            fact(N1, F1),
            F is F1 * N
        ).
```

Note that Prolog's if-then-else construct does not leave a choicepoint. A choicepoint is created, but is removed as soon as the condition succeeds or fails. So `fact` would be subject to TRO, if only it were tail recursive.

Adding an accumulator

We make factorial tail recursive by introducing an **accumulating parameter**, or just an **accumulator**. This is an extra parameter to the predicate that holds a partially computed result.

Usually the base case for the recursion will specify that the partially computed result is actually the result. The recursive clause usually computes more of the partially computed result, and passes this in the recursive goal.

The key to getting the implementation correct is specifying what the accumulator *means* and how it relates to the final result. To see how to add an accumulator, determine what is done after the recursive call, and then respecify the predicate so it performs this task, too.

Adding an accumulator

For factorial, we compute `fact(N1, F1), F is F1 * N` last, so the tail recursive version will need to perform the multiplication too. We must define a predicate `fact1(N, A, F)` so that `F` is `A` times the factorial of `N`. In most cases, it is not difficult to see how to transform the original definition to the tail recursive one.

```
\par
\vspace2mm
fact(N, F) :-
        ( N =:= 0 ->
            F = 1
        ;   N > 0,
            N1 is N - 1,
            fact(N1, F1),
            F is F1 * N
        ).
```

transforms to ⇓

```
fact(N, F) :- fact1(N, 1, F).
fact1(N, A, F) :-
        ( N =:= 0 ->
            F = A
        ;   N > 0,
```

```
            N1 is N - 1,
            A1 is A * N,
            fact(N1, A1, F)
        ).
```

– section 13 slide 7 –

Adding an accumulator

Finally, define the original predicate in terms of the new one. Again, it is usually easy to see how to do that.

Another way to think about writing a tail recursive implementation of a predicate is to realise that it will essentially be a loop, so think of how you would write it as a `while` loop, and then write that loop in Prolog.

```
A = 1;
while (N > 0) {
    A *= N;
    N--;
}
if (N == 0) return A;
else FAIL;
```

translates to ⇓

```
fact(N, F) :- fact1(N, 1, F).
fact1(N, A, F) :-
        ( N > 0,
            A1 is A * N,
            N1 is N - 1,
            fact(N1, A1, F)
        ; N =:= 0 ->
            F = A
        ).
```

– section 13 slide 8 –

Transformation

Another approach is to systematically transform the non-tail recursive version into an equivalent tail recursive predicate. Start by defining a predicate to do the work of the recursive call to `fact/2` and everything following it. Then replace the call to `fact(N, F2)` by the definition of `fact/2`. This is called *unfolding*.

```
fact1(N, A, F) :-
        fact(N, F2),
        F is F2 * A
```

transforms to ⇓

```
fact1(N, A, F) :-
        ( N =:= 0 ->
            F2 = 1
        ;   N > 0,
            N1 is N - 1,
            fact(N1, F1),
            F2 is F1 * N
        ),
        F is F2 * A.
```

– section 13 slide 9 –

Transformation

Next we move the final goal into both the *then* and *else* branches.

```
fact1(N, A, F) :-
        ( N =:= 0 ->
            F2 = 1
        ;   N > 0,
            N1 is N - 1,
            fact(N1, F1),
            F2 is F1 * N
        ),
        F is F2 * A.
```

transforms to ⇓

```
fact1(N, A, F) :-
        ( N =:= 0 ->
            F2 = 1,
            F is F2 * A
        ;   N > 0,
            N1 is N - 1,
            fact(N1, F1),
            F2 is F1 * N,
            F is F2 * A
        ).
```

– section 13 slide 10 –

Transformation

The next step is to simplify the arithmetic goals.

```
fact1(N, A, F) :-
        ( N =:= 0 ->
            F2 = 1,
            F is F2 * A
        ;   N > 0,
            N1 is N - 1,
            fact(N1, F1),
            F2 is F1 * N,
            F is F2 * A
        ).
```

transforms to ⇓

```
fact1(N, A, F) :-
        ( N =:= 0 ->
            F = A
        ;   N > 0,
            N1 is N - 1,
            fact(N1, F1),
            F is (F1 * N) * A
        ).
```

– section 13 slide 11 –

Transformation

Now we utilise the associativity of multiplication. This is the insightful step that is necessary to be able to make the next step.

```
fact1(N, A, F) :-
        ( N =:= 0 ->
            F = A
        ;   N > 0,
            N1 is N - 1,
            fact(N1, F1),
            F is (F1 * N) * A
        ).
```

transforms to ⇓

```
fact1(N, A, F) :-
        ( N =:= 0 ->
            F = A
        ;   N > 0,
            N1 is N - 1,
            fact(N1, F1),
            F is F1 * (N * A)
        ).
```

– section 13 slide 12 –

Transformation

Now part of the computation can be moved before the call to `fact/2`.

```
fact1(N, A, F) :-
        ( N =:= 0 ->
            F = A
        ;   N > 0,
            N1 is N - 1,
            fact(N1, F1),
            F is F1 * (N * A)
        ).
```

transforms to ⇓

```
fact1(N, A, F) :-
        ( N =:= 0 ->
            F = A
        ;   N > 0,
            N1 is N - 1,
            A1 is N * A,
            fact(N1, F1),
            F is F1 * A1
        ).
```

– section 13 slide 13 –

Transformation

The final step is to recognise that the last two goals look very much like the body of the original definition of `fact1/3`, with the substitution $\{\texttt{N} \mapsto \texttt{N1}, \texttt{F2} \mapsto \texttt{F1}, \texttt{A} \mapsto \texttt{A1}\}$. So we replace those two goals with the clause head with that substitution applied. This is called *folding*.

```
fact1(N, A, F) :-
        ( N =:= 0 ->
            F = A
        ;   N > 0,
            N1 is N - 1,
            A1 is N * A,
            fact(N1, F1),
            F is F1 * A1
        ).
```

transforms to ⇓

```
fact1(N, A, F) :-
        ( N =:= 0 ->
            F = A
        ;   N > 0,
            N1 is N - 1,
            A1 is N * A,
            fact1(N1, A1, F)
        ).
```

Accumulating Lists

The tail recursive version of `fact` is a constant factor more efficient, because it behaves like a loop. Sometimes accumulators can make an order difference, if it can replace an operation with a computation of lower asymptotic complexity, for example replacing `append/3` (linear time) with list construction (constant time).

```
rev1([], []).
rev1([A|BC], CBA) :-
        rev1(BC, CB),
        append(CB, [A], CBA).
```

This definition of `rev1/2` is of quadratic complexity, because for the n^{th} element from the end of the first argument, we append a list of length $n-1$ to a singleton list. Doing this for each of the n elements gives time proportional to $\frac{n(n-1)}{2}$.

Tail recursive `rev1/2`

The first step in making a tail recursive version of `rev1/2` is to specify the new predicate. It must combine the work of `rev1/2` with that of `append/3`. The specification is:

```
% rev(BCD, A, DCBA)
% DCBA is BCD reversed, with A appended
```

We could develop this by transformation as we did for `fact1/3`, but we implement it directly here. We begin with the base case, for `BCD` = []:

```
rev([], A, A).
```

Tail recursive `rev1/2`

For the recursive case, take `BCD` = `[B|CD]`:

```
rev([B|CD], A, DCBA) :-
```

the result, `DCBA`, must be the reverse of `CD`, with `[B]` appended to the end, and `A` appended after that. In Haskell notation this is

```
(rev cd ++ [b]) ++ a.
```

Because `append` is associative, this is the same as

```
rev cd ++ ([b] ++ a)   ≡   rev cd ++ (b:a).
```

We can use our `rev/3` predicate to compute that:

```
rev([B|CD], A, DCBA) :-
        rev(CD, [B|A], DCBA).
```

Tail recursive `rev1/2`

```
rev([], A, A).
rev([B|CD], A, DCBA) :-
        rev(CD, [B|A], DCBA).
```

At each recursive step, this code removes an element from the head of the input list and adds it to the head of the accumulator. The cost of each step is therefore a constant, so the overall cost is linear in the length of the list.

Accumulator lists work like a stack: the last element of the accumulator is the first element that was added to it, and so on. Thus at the end of the input list, the accumulator is the reverse of the original input.

Difference pairs

The trick used for a tail recursive `reverse` predicate is often used in Prolog: a predicate that generates a list takes an extra argument specifying what should come after the list. This avoids the need to append to the list.

In Prolog, if you do not know what will come after at the time you call the predicate, you can pass an unbound variable, and bind that variable when you do know what should come after. Thus many predicates intended to produce a list have two arguments, the first is the list produced, and the second is what comes after. This is called a *difference pair*, because the predicate generates the *difference* between the first and second list.

```
generate_whole(X, L, L0) :-
        generate_first_part(X, L, L1),
        generate_second_part(X, L1, L0).
```

Section 14: All solutions and impurity

All solutions

Sometimes one would like to bring together all solutions to a goal. Prolog's *all solutions* predicates do exactly this.

`setof(Template, Goal, List)` binds `List` to sorted list of all distinct instances of `Template` satisfying `Goal`

`Template` can be any term, usually containing some of the variables appearing in `Goal`. On completion, `setof/3` binds its `List` argument, but does not further bind any variables in the `Goal`.

```
?- setof(P-C, parent(P, C), List).
List = [duchess_kate-prince_george,
prince_charles-prince_harry,
prince_charles-prince_william,
prince_philip-prince_charles,
prince_william-prince_george,
princess_diana-prince_harry,
princess_diana-prince_william,
queen_elizabeth-prince_charles].
```

All solutions

If `Goal` contains variables not appearing in `Template`, `setof/3` will backtrack over each distinct binding of these variables, for each of them binding `List` to the list of instances of `Template` for that binding.

```
?- setof(C, parent(P, C), List).
P = duchess_kate,
List = [prince_george] ;
P = prince_charles,
```

```
List = [prince_harry, prince_william] ;
P = prince_philip,
List = [prince_charles] ;
P = prince_william,
List = [prince_george] ;
P = princess_diana,
List = [prince_harry, prince_william] ;
P = queen_elizabeth,
List = [prince_charles].
```

Existential quantification

Use existential quantification, written with infix caret (^), to collect solutions for a template regardless of the bindings of some of the variables not in the `Template`.

E.g., to find all the people in the database who are parents of any child:

```
?- setof(P, C^parent(P, C), Parents).
Parents = [duchess_kate, prince_charles,
prince_philip, prince_william, princess_diana,
queen_elizabeth].
```

Unsorted solutions

The `bagof/3` predicate is just like `setof/3`, except that it does not sort the result or remove duplicates.

```
?- bagof(P, C^parent(P, C), Parents).
Parents = [queen_elizabeth, prince_philip,
prince_charles, prince_charles, princess_diana,
princess_diana, prince_william, duchess_kate].
```

Solutions are collected in the order they are produced. This is not purely logical, because the order of solutions should not matter, nor should the number of times a solution is produced.

Higher Order Programming

The `call/1` built-in predicate executes a term as a goal. This capitalises on the fact that Prolog terms look just like Prolog goals. Many Prologs, including SWI Prolog, will `call` a variable used as a goal.

```
?- X=append([1,2],[3],L), call(X).
X = append([1, 2], [3], [1, 2, 3]),
L = [1, 2, 3].
?- X=append(A, B, [1]), X.
X = append([], [1], [1]),
A = [],
B = [1] ;
X = append([1], [], [1]),
A = [1],
B = [] ;
false.
```

Currying

Currying in Prolog is done by simply omitting some final arguments of a higher-order goal. To support this, many Prologs, including SWI, support versions of `call` of higher arity. All arguments to `call/`n after the goal (first) are added as extra arguments to the goal.

```
?- X=append([1,2],[3]), call(X, L).
X = append([1, 2], [3]),
L = [1, 2, 3].
```

Writing higher-order code

It is fairly straightforward to write higher order predicates using `call/`N. For example, here is `map/3` in Prolog:

```
map(_, [], []).
map(P, [X|Xs], [Y|Ys]) :-
        call(P, X, Y),
        map(P, Xs, Ys).
```

```
?- map(plus(1), [3,4,5], L).
L = [4, 5, 6] ;
false.
?- map(plus(1), L, [3,4,5]).
L = [2, 3, 4] ;
false.
```

This is in the SWI Prolog library with the name `maplist`/2, 3, 4, and 5.

Input/Output

Prolog's Input/Output facility does not even try to be pure. I/O operations are executed when they are reached according to Prolog's simple execution order. I/O is not "undone" on backtracking.

Prolog has builtin predicates to read and write arbitrary Prolog terms. Prolog also allows users to define their own operators. This makes Prolog very convenient for applications involving structured I/O.

```
?- op(800, xfy, wibble).
true.
?- read(X).
|: p(x,[1,2],X>Y wibble z).
X = p(x, [1, 2], _G280>_G281 wibble z).
?- write(p(x,[1,2],X>Y wibble z)).
p(x,[1,2],_G207>_G208 wibble z)
true.
```

Input/Output

`write/1` is handy for printing messages:

```
?- write('hello '), write('world!').
hello world!
true.
?- write('world!'), write('hello ').
world!hello
true.
```

This demonstrates that Prolog's input/output predicates are non-logical. These should be equivalent, because conjunction *should* be commutative.

Code that performs I/O must be handled carefully — you must be aware of the modes. It is recommended to isolate I/O in a small part of the code, and keep the bulk of your code I/O-free. (This is a good idea in any language.)

Comparing terms

All Prolog terms can be compared for ordering using the built-in predicates `@<`, `@=<`, `@>`, and `@>=`. Prolog, somewhat arbitrarily, uses the ordering

$$\text{Variables} < \text{Numbers} < \text{Atoms} < \text{CompoundTerms}$$

but most usefully, within these classes, terms are ordered as one would expect: numbers by value and atoms are sorted alphabetically. Compound terms are ordered first by arity, then alphabetically by functor, and finally by arguments, left-to-right. It is best to use these only for ground terms.

```
?- hello @< hi.
true.
?- X @< 7, X = foo.
X = foo.
?- X = foo, X @< 7.
false.
```

Sorting

There are three SWI Prolog builtins for sorting ground lists according to the `@<` ordering: `sort/2` sorts a list, removing duplicates, `msort/2` sorts a list, without removing duplicates, and `keysort/2` stably sorts list of `X-Y` terms, only comparing `X` parts:

```
?- sort([h,e,l,l,o], L).
L = [e, h, l, o].
?- msort([h,e,l,l,o], L).
L = [e, h, l, l, o].
?- keysort([7-a, 3-b, 3-c, 8-d, 3-a], L).
L = [3-b, 3-c, 3-a, 7-a, 8-d].
```

Determining term types

`integer/1` holds for integers and fails for anything else. It also fails for variables.

```
?- integer(3).
true.
?- integer(a).
false.
?- integer(X).
false.
```

Similarly, `float/1` recognises floats, `number` recognises either kind of number, `atom/1` recognises atoms, and `compound/1` recognises compound terms. All of these fail for variables, so must be used with care.

Recognising variables

`var/1` holds for unbound variables, `nonvar/1` holds for any term other than an unbound variable, and `ground/1` holds for ground terms (this requires traversing the whole term). Using these or the predicates on the previous slide can make your code behave differently in different modes.

But they can also be used to write code that works in multiple modes.

Here is a tail-recursive version of `len/2` that works whenever the length is known:

```
len2(L, N) :-
    (   N =:= 0
    ->  L = []
    ;   N1 is N - 1,
        L = [_|L1],
        len2(L1, N1)
    ).
```

Recognising variables

This version works when the length is unknown:

```
len1([], N, N).
len1([_|L], N0, N) :-
    N1 is N0 + 1,
    len1(L, N1, N).
```

This code chooses between the two:

```
len(L, N) :-
    (   integer(N)
    ->  len2(L, N)
    ;   nonvar(N)
    ->  throw(error(type_error(integer, N),
                    context(len/2, '')))
    ;   len1(L, 0, N)
    ).
```

Mercury

The Mercury language was developed at The University of Melbourne as a purely declarative successor to Prolog. Mercury is strongly typed, with a type system very similar to Haskell's. It is also strongly moded, which means that the binding state of all variables is determined at compile-time. Mercury has a strong determinism system, as well, which allows the compiler to determine, for each predicate mode, whether it will always be deterministic.

All these properties, along with aggressive compiler optimisations, make Mercury among the highest-performing declarative languages. Although Mercury's learning curve is somewhat steeper than Prolog's, it is a more realistic choice for serious development.

Section 15: Constraint (logic) programming

Constraint (logic) programming

An imperative program specifies the exact sequence of actions to be executed by the computer.

A functional program specifies how the result of the program is to be computed at a more abstract level. One can read function definitions as suggesting an order of actions, but the language implementation can and sometimes will deviate from that order, due to lazyness, parallel execution, and various optimizations.

A logic program is in some ways more declarative, as it species a set of equality constraints that the terms of the solution must satisfy, and then searches for a solution.

A constraint program is more declarative still, as it allows more general constraints than just equality constraints. The search for a solution will typically follow an algorithm whose relationship to the specification can be recognized only by experts.

Problem specification

The specification of a constraint problem consists of

- a set of variables, each variable having a known domain,
- a set of constraints, with each constraint involving one or more variables, and
- an optional objective function.

The job of the constraint programming system is to find a *solution*, a set of assignments of values to variables (with the value of each variable being drawn from its domain) that satisfies all the constraints.

The objective function, if there is one, maps each solution to a number. If this number represents a cost, you want to pick the cheapest solution; if this number represents a profit, you want to pick the most profitable solution.

NOTE The objective function is optional because sometimes you care only about the existence of a solution, and sometimes all solutions are equally good.

The set of constraints may be given in advance, or it may be discovered piecemeal, as you go along. The latter occurs reasonably often in planning problems.

Kinds of constraint problems

There are several kinds of constraints, of which the following four are the most common. Most CP systems handle only one or two kinds.

In *Herbrand* constraint systems, the variables represent terms, and the basic constraints are unifications, i.e. they have the form $term_1 = term_2$. This is the constraint domain implemented by Prolog.

In *finite domain* or *FD* constraint systems, each variable's domain has a finite number of elements.

In *boolean satisfiability* or `SAT` systems, the variables represent booleans, and each constraint asserts the truth of an expression constructed using logical operations such as AND, OR, NOT and implication.

In *linear inequality* constraint systems, the variables represent real numbers (or sometimes integers), and the constraints are of the form $ax + by \leq c$ (where x and y are variables, and a, b and c are constants).

NOTE You can view boolean satisfiability (SAT) problems as a subclass of finite domain problems, but there are specialized algorithms that work only on booleans and not on finite domains with more than two values, Research into solving SAT problems has made great progress over the last 20 years, and modern SAT solvers are surprisingly efficient at solving NP complete problems. So in practice, the two classes of problems should be handled differently.

Type inference

One application of constraint programming using Herbrand terms is type inference for the Hindley-Milner type system (the Haskell type system). The process of inferring the types of variables inside functions can be viewed as a matter of solving a set of constraints over variable types. Herbrand terms can be used to neatly represent types. and (if not declared) the types of the functions and predicates themselves can be viewed as a Herbrand constraint problem.

This approach represents the type of each variable in the program as a Herbrand term, and applies rules to produce equality constraints for each subexpression in the function definitions. To simplify the process, we assume the program has been "normalised" into an equivalent definition expressed in a restricted form. In particular, all nested expressions have been replaced by let-bound temporary variables, and deconstructions have been replaced by `case` constructs.

Type inference

For example, the rule for handling function application states that if the type of f is $a \to b$ and the type of x is t then the type of $f\,x$ is b, where $t = a$. The `case` construct is handled by unifying the type of the expression with the type of all of the cases.

Consider the normalised definition of the Haskell `map` function:

```
map f l = case l of
              []     -> []
              (e:es) -> let t = map f es
                            h = f e
                        in  h:t
```

The form of the equation tells you that the type of `map`, t_{map}, is a type of the form $t_1 \to t_2 \to t_r$.

t_{map}, t_1, t_2 and t_r are all constraint variables whose domain is the set of possible Haskell types, and $t_{map} = t_1 \to t_2 \to t_r$ is a constraint on those variables.

Type inference: an example

We can use Prolog's Herbrand terms to infer types:

```
    map f l = case l of
                  []     -> []
                  (e:es) -> let t = map f es
                                h = f e
                            in  h:t
```

```
?- Map = F => L => Result,
|    L = [_], Result = [_],
|    L = [E], Es = [E],
|    Map = F => Es => T,
|    F = E => H,
|    Result = [H], Result = T.
Map =  (E=>H)=>[E]=>[H],
F = E=>H,
L = Es, Es = [E],
Result = T, T = [H].
```

Linear inequality constraints

Suppose you want to make banana and/or chocolate cakes for a bake sale, and you have 10 kg of flour, 30 bananas, 1.2 kg of sugar, 1.5 kg of butter, and 700 grams of cocoa on hand. You can charge $4.00 for a banana cake and $6.50 for a chocolate one. Each kind of cake requires a certain quantity of each ingredient. How do you determine how many of each cake to make so as to maximise your profit?

To solve such a problem, you need to set up a system of constraints saying, for example, that the number of each kind of cake times the amount of flour needed for that kind of cake must add to no more than the amount of flour you have.

You also need to specify that the number of each kind of cake must be non-negative. Finally, you need to define your revenue as the sum of the number of each kind of cake times its price, and specify that you would like to maximise revenue.

Linear inequality constraints

We can use SWI Prolog's `library(clpr)` to solve such problems. This library requires constraints to be enclosed in curly braces.

```
?- {250*B + 200*C =< 10000},
|    {2*B =< 30},
|    {75*B + 150*C =< 1200},
|    {100*B + 150*C =< 1500},
|    {75*C =< 700},
|    {B>=0}, {C>=0},
|    {Revenue = 4*B + 6.5*C}, maximize(Revenue).
B = 12.0,
C = 2.0,
Revenue = 61.0
```

So we can make $61.00 by making 12 Banana and 2 Chocolate cakes.

MiniZinc

The MiniZinc language is a *modelling* language, allowing concise specification of constraint satisfaction problems. This same problem could be expressed in MiniZinc as:

```
% b is the number of banana cakes to make
% c is the number of chocolate cakes to make

constraint 250*b + 200*c <= 10000;
constraint   2*b         <=    30;
constraint  75*b + 150*c <=  1200;
constraint 100*b + 150*c <=  1500;
constraint          75*c <=   700;
constraint b >= 0;
constraint c >= 0;

solve maximize 400*b + 650*c;
```

Sudoku

Sudoku is a class of puzzles played on a 9x9 grid. Each grid position should hold a number between 1 and 9. The same integer may not appear twice

- in a single row,
- in a single column, or
- in one of the nine 3x3 boxes.

The puzzle creator provides some of the numbers; the challenge is to fill in the rest.

This is a classic finite domain constraint satisfaction problem.

	c1	c2	c3	c4	c5	c6	c7	c8	c9
r1	5	3			7				
r2	6			1	9	5			
r3		9	8					6	
r4	8				6				3
r5	4			8		3			1
r6	7				2				6
r7		6					2	8	
r8				4	1	9			5
r9					8			7	9

Sudoku as finite domain constraints

You can represent the rules of sudoku as a set of 81 constraint variables ($r1c1$, $r1c2$ etc) each with the domain 1..9, and 27 all-different constraints: one for each row, one for each column, and one for each box. For example, the constraint for the top left box would be all_different($[r1c1, r1c2, r1c3, r2c1, r2c2, r2c3, r3c1, r3c2, r3c3]$).

Initially, the domain of each variable is [1..9]. If you fix the value of a variable e.g. by setting $r1c1 = 5$, this means that the other variables that share a row, column or box with $r1c1$ (and that therefore appear in an all-different constraint with it) cannot be 5, so their domain can be reduced to $[1..4, 6..9]$.

This is how the variables fixed by our example puzzle reduce the domain of $r3c1$ to only [1..2], and the domain of $r5c5$ to only [5].

Fixing $r5c5$ to be 5 gives us a chance to further reduce the domains of the other variables linked to $r5c5$ by constraints, e.g. $r7c5$.

Sudoku in SWI Prolog

Using SWI's `library(clpfd)`, Sudoku problems can be solved:

```
sudoku(Rows) :-
        length(Rows, 9), maplist(same_length(Rows), Rows),
        append(Rows, Vs), Vs ins 1..9,
        maplist(all_distinct, Rows),
        transpose(Rows, Columns),
        maplist(all_distinct, Columns),
        Rows = [A,B,C,D,E,F,G,H,I],
        blocks(A, B, C), blocks(D, E, F), blocks(G, H, I).

blocks([], [], []).
blocks([A,B,C|Bs1], [D,E,F|Bs2], [G,H,I|Bs3]) :-
        all_distinct([A,B,C,D,E,F,G,H,I]),
        blocks(Bs1, Bs2, Bs3).
```


Sudoku in SWI Prolog

```
?- Puzzle=[[5,3,_, _,7,_, _,_,_],
|          [6,_,_, 1,9,5, _,_,_],
|          [_,9,8, _,_,_, _,6,_],
|
|          [8,_,_, _,6,_, _,_,3],
|          [4,_,_, 8,_,3, _,_,1],
|          [7,_,_, _,2,_, _,_,6],
|
|          [_,6,_, _,_,_, 2,8,_],
|          [_,_,_, 4,1,9, _,_,5],
|          [_,_,_, _,8,_, _,7,9]],
|    sudoku(Puzzle),
|    write(Puzzle).
```

Sudoku solution

In less than $\frac{1}{20}$ of a second, this produces the solution:

```
Puzzle=[[5,3,4, 6,7,8, 9,1,2],
        [6,7,2, 1,9,5, 3,4,8],
        [1,9,8, 3,4,2, 5,6,7],

        [8,5,9, 7,6,1, 4,2,3],
        [4,2,6, 8,5,3, 7,9,1],
        [7,1,3, 9,2,4, 8,5,6],

        [9,6,1, 5,3,7, 2,8,4],
        [2,8,7, 4,1,9, 6,3,5],
        [3,4,5, 2,8,6, 1,7,9]]
```

Search

Prolog normally employs a strategy known as *"generate and test"* to search for variable bindings that satisfy constraints. Nondeterministic goals *generate* potential solutions; later goals *test* those solutions, imposing further constraints and rejecting some candidate solutions.

For example, in

```
?- between(1,9,X), 0 =:= X mod 2, X =:= X * X mod 10.
```

The first goal generates single-digit numbers, the second tests that it is even, and the third that its square ends in the same digit.

Constraint logic programming uses the more efficient *"constrain and generate"* strategy. In this approach, constraints on variables can be more sophisticated than simply binding to a Herbrand term. This is generally accomplished in Prolog systems with *attributed variables*, which allow constraint domains to control unification of constrained variables.

Propagation

The usual algorithm for solving a set of FD constraints consists of two steps: *propagation* and *labelling*.

In the propagation step, we try to reduce the domain of each variable as much as possible.

For each constraint, we check whether the constraint rules out any values in the current domains of any of the variables in that constraint. If it does, then we remove that value from the domain of that variable, and schedule the constraints involving that variable to be looked at again.

The propagation step ends

- if every variable has a domain of size one, which represents a fixed value, since this represents a solution;
- if some variable has an empty domain, since this represents failure; or
- if there are no more constraints to look at, in which case propagation can do no more.

Labelling

If propagation cannot do any more, we go on the *labelling* step, which

- picks a not-yet-fixed variable,
- partitions its current domain (of size n) into k parts $d_1, ..., d_k$, where usually $k = 2$ but may be any value satisfying $2 \leq k \leq n$, and
- recursively invokes the whole constraint solving algorithm k times, with invocation i restricting the domain of the chosen variable to d_i.

Each recursive invocation also consists of a propagation step and (if needed) a labelling step. The whole computation therefore consists of alternating propagation and labelling steps.

The labelling steps generate a search tree. The size of the tree depends on the effectiveness of propagation: the more effective propagation is at removing values from domains, the smaller the tree will be, and the less time searching it will take.

NOTE The root node of the tree represents the computation for solving the whole constraint problem. Every other node represents the choice within the computation represented by its parent node. If it is the ith child of its parent node, then it represents the choice in the labelling step to set the domain of the selected variable to d_i.

If the current domain of the chosen variable is e.g. [1..5], then the selected partition maybe contain two to five parts. The only partition that divides [1..5] into five parts is of course [1], [2], [3], [4], [5], but there are many partitions that divide [1..5] into two parts, with [1..2], [3..5] and [1, 3, 5], [2, 4] being two of them.

Note that one can view the labelling step as imposing an extra *constraint* on each branch of the search tree, with the extra constraint for branch i requiring the chosen variable to have a value in d_i of its current domain.

Timetabling

Timetabling is a finite domain problem of great practical importance.

To timetable university lectures, you need two variables for every lecture:

- The domain of $timeCOMP30020L1$ represents all the available lecture time slots from $mon8am$ to $fri7pm$.
- The domain of $locCOMP30020L1$ represents all the available lecture theatres that are big enough for COMP30020's enrolment.

There are many kinds of constraints, including:

- all the lectures for a subject must be at different times
- all the lectures taught by a single lecturer must be at different times
- all the rooms of lectures taught at the same time must be different
- the room in which each lecture is taught must be big enough for its anticipated enrolment

---

The search space

Timetabling happens to be very hard to do well, partly because it generates a huge search space if you impose only absolutely necessary constraints such as the previous two.

Imposing the constraint that $timeCOMP30020L1$ must be earlier in the week than $timeCOMP30020L2$ breaks a symmetry and thus halves the number of solutions. Imposing such a constraint on each of say 1000 subjects reduces the number of solutions by a factor of 2^{1000}.

Despite this *astronomical* reduction, the search space is still *very* large.

Most constraint problems are NP-complete, so in the worst case, their solution requires time that is exponential in the size of the problem.

Nevertheless, heuristics can often reduce the solution time to manageable levels, and new and better heuristics are being developed all the time.

NOTE Another essential constraint is that you can schedule at most one lecture in a particular theatre in

a particular time slot. The university has violated this constraint in the past, which in one case led to a theatre overflowing with three sets of students, and three lecturers at the lectern energetically discussing whose subject the theatre was *really* reserved for.

One example of another kind of essential constraint applies to masters subjects intended for part-time students. For these subjects, the usual arrangement is that the two lectures in a week are delivered back to back, from 5pm to 7pm, in the same lecture theatre, with the tutorial following from 7 to 8pm. This allows people who work during the day to take this subject without either having to take (much) time off work or having to commute to campus more than once a week.

A solution in which COMP30020 has

- a lecture called COMP30020L1 on tuesdays at 3pm and
- a lecture called COMP30020L2 on thursdays at 3pm,

and a solution in which COMP30020 has

- a lecture called COMP30020L2 on tuesdays at 3pm and
- a lecture called COMP30020L1 on thursdays at 3pm,

does not make any difference in practice, but to a constraint system, these are different solutions, and it will try to find them both, unless you add a constraint to rule out one of them.

Optional constraints

Another reason why timetabling is very hard to do well is because there are many objectives that are desirable but not essential:

- Different lectures in the same subject should not be on the same day.
- Schedule as few lectures for 8am, 1pm, 6pm and 7pm as possible.
- If a lecturer or a cohort of students have lectures in consecutive time slots, it should be possible to walk from the first lecture theatre to the second in less than ten minutes. (This *should* be essential, but apparently isn't.)
- If a lecturer or a cohort of students spend most of their time in a building, then their lectures should be in that building.

Requiring all these constraints to be fulfilled would result in a horribly overconstrained problem and thus have no solution.

However, if you have several solutions that satisfy the essential constraints, you can choose the solution that satisfies the most optional constraints.

Section 16: Monads

`any` and `all`

Other useful higher-order functions are

```
any :: (a -> Bool) -> [a] -> Bool
all :: (a -> Bool) -> [a] -> Bool
```

For example, to see if every word in a list contains the letter 'e':

```
Prelude> all (elem 'e') ["eclectic", "ele
phant", "legion"]
True
```

To check if a word contains any vowels:

```
Prelude> any (\x -> elem x "aeiou") "sky"
False
```

– section 16 slide 1 –

`flip`

If the order of arguments of `elem` were reversed, we could have used currying rather than the bulky `\x -> elem x "aeiou"`. The `flip` function takes a function and returns a function with the order of arguments flipped.

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f x y = f y x
```

```
Prelude> any (flip elem "aeiou") "hmmmm"
False
```

The ability to write functions to construct other functions is one of Haskell's strengths.

– section 16 slide 2 –

Monads

Monads build on this strength. A *monad* is a type constructor that represents a computation. These computations can then be composed to create other computations, and so on. The power of monads lies in the programmer's ability to determine *how* the computations are composed. Phil Wadler, who introduced monads to Haskell, describes them as "programmable semicolons".

A monad `M` is a type constructor that supports two operations:

- A sequencing operation, denoted `>>=`, whose type is `M a -> (a -> M b) -> M b`.
- An identity operation, denoted `return`, whose type is `a -> M a`.

– section 16 slide 3 –

Monads

Think of the type `M a` as denoting a computation that produces an `a`, and possibly carries something extra. For example, if `M` is the `Maybe` type constructor, that something extra is an indication of whether an error has occurred so far.

- You can take a value of type `a` and use the (misnamed) identity operation to wrap it in the monad's type constructor.
- Once you have such a wrapped value, you can use the sequencing operation to perform an operation on it. The `>>=` operation will unwrap its first argument, and then typically it will invoke the function given to it as its second argument, which will return a wrapped up result.

NOTE You can apply the sequencing operation to any value wrapped up in the monad's type constructor; it does not have to have been generated by the monad's identity function.

The Maybe and MaybeOK monads

The obvious ways to define the monad operations for the `Maybe` and `MaybeOK` type constructors are these (`MaybeOK` is not in the library):

```
-- monad ops for Maybe
data Maybe t = Just t
             | Nothing
\par
\vspace2mm
return x = Just x
\par
\vspace2mm
(Just x) >>= f = f x
Nothing >>= _  = Nothing
```

```
-- monad ops for MaybeOK
data MaybeOK t = OK t
               | Error String
\par
\vspace2mm
return x        = OK x
\par
\vspace2mm
(OK x) >>= f    = f x
(Error m) >>= _ = Error m
```

In a sequence of calls to `>>=`, as long as all invocations of `f` succeed, (returning `Just x` or `OK x`), you keep going.

Once you get a failure indication, (returning `Nothing` or `Error m`), you keep that failure indication and perform no further operations.

Why you may want these monads

Suppose you want to encode a sequence of operations that each may fail. Here are two such operations:

```
maybe_head :: [a] -> MaybeOK a
maybe_head [] = Error "head of empty list"
maybe_head (x:_) = OK x
\par
\vspace2mm
maybe_sqrt :: Int -> MaybeOK Double
maybe_sqrt x =
    if x >= 0 then
        OK (sqrt (fromIntegral x))
    else
        Error "sqrt of negative number"
```

How can you encode a sequence of operations such as taking the head of a list and computing its square root?

NOTE The `fromIntegral` function can convert an integer to any other numeric type.

Simplifying code with monads

```
maybe_head :: [a] -> MaybeOK a
maybe_sqrt :: Int -> MaybeOK Double
maybe_sqrt_of_head ::
    [Int] -> MaybeOK Double
\par
\vspace2mm
-- definition not using monads
maybe_sqrt_of_head l =
    let mh = maybe_head l in
        case mh of
            Error msg -> Error msg
            OK h -> maybe_sqrt h
\par
\vspace2mm
-- simpler definition using monads
maybe_sqrt_of_head l =
    maybe_head l >>= maybe_sqrt
```

NOTE The monadic version is simpler because in that version, the work of checking for failure and handling it if found is done once, in the `MaybeOK` monad's sequence operator.

In the version without monads, this code would have to be repeated for every step in a sequence of possibly-failing operations. If this sequence has ten operations, you will be grow bored of writing the pattern-match code way before writing the tenth copy. The steady increase in indentation required by the offside rule also makes the code ugly, since the code of the tenth iteration would have to be squashed up against the right margin. Longer sequences of operations may even require the use of more columns than your screen actually has.

Note that the two occurrences of `Error m` in the first definition are of different types; the first is type `MaybeOK Int`, while the second is of type `MaybeOK Double`. The two occurrences of `Error m` in the definition of the sequence operation for the `MaybeOK` monad are similarly of different types, `MaybeOK a` for the first and `MaybeOK b` for the second.

I/O actions in Haskell

Haskell has a type constructor called `IO`. A function that returns a value of type `IO t` for some `t` will return a value of type `t`, but can also do input and/or output. Such functions can be called I/O functions or I/O actions.

Haskell has several functions for reading input, including

```
getChar :: IO Char
getLine :: IO String
```

Haskell has several functions for writing output, including

```
putChar  :: Char -> IO ()
putStr   :: String -> IO ()
putStrLn :: String -> IO ()
print    :: (Show a) => a -> IO ()
```

The type `()`, called `unit`, is the type of 0-tuples (tuples containing zero values). This is similar to the `void` type in C or Java. There is only one value of this type, the empty tuple, which is also denoted `()`.

NOTE The notion that a function that returns a value of type `IO t` for some `t` actually does input and/or output is only approximately correct; we will get to the fully correct notion in a few slides.

Since there is only one value of type `()`, variables of this type carry no information.

Haskell represents a computation that takes a value of type `a` and computes a value of type `b` as a function of type `a -> b`.

Haskell represents a computation that takes a value of type `a` and computes a value of type `b` while also possibly performing input and/or output (subject to the caveat above) as a function of type `a -> IO b`.

All the functions listed on this slide are defined in the prelude.

`getChar` returns the next character in the input.

`getLine` returns the next line in the input, without the final `'\n'`.

`putChar` prints the given character.

`putStr` prints the given string.

`putStrLn` prints the given string, and appends a newline character.

`print` uses the `show` function (which is defined for every type in the `Show` type class) to turn the given value into a string; it then prints that string and appends a newline character.

Operations of the I/O monad

The type constructor `IO` is a monad.

The identity operation: `return val` just returns `val` (inside `IO`) without doing any I/O.

The sequencing operation: `f >>= g`

1. calls `f`, which may do I/O, and which will return a value `rf` that may be meaningful or may be `()`,
2. calls `g rf` (passing the return value of `f` to `g`), which may do I/O, and which will return a value `rg` that also may be meaningful or may be `()`,

3. returns `rg` (inside `IO`) as the result of `f >>= g`.

You can use the sequencing operation to create a chain of any number of I/O actions.

Example of monadic I/O: hello world

```
hello :: IO ()
hello =
    putStr "Hello, "
    >>=
    \_ -> putStrLn "world!"
```

This code has two I/O actions connected with `>>=`.

1. The first is a call to `putStr`. This prints the first half of the message, and returns `()`.
2. The second is an anonymous function. It takes as argument and ignores the result of the first action, and then calls `putStrLn` to print the second half of the message, adding a newline at the end.

The result of the action sequence is the result of the last action.

NOTE In this case, the resulf of the last action will always be `()`.

Actually, there is a third monad operation besides `return` and `>>=`: the operation `>>`. This is identical to `>>=`, with the exception that it ignores the return value of its first operand, and does not pass it to its second operand. This is useful when the second operand would otherwise have to explicitly ignore its argument, as in this case.

With `>>`, the code of this function could be slightly simpler:

```
hello :: IO ()
hello =
    putStr "Hello, "
    >>
    putStrLn "world!"
```

Example of monadic I/O: greetings

```
greet :: IO ()
greet =
  putStr "Greetings! What is your name? "
  >>=
  \_ -> getLine
  >>=
  \name -> (
      putStr "Where are you from? "
      >>=
      \_ -> getLine
      >>=
      \town ->
        let msg = "Welcome, " ++ name ++
                  " from " ++ town
        in  putStrLn msg
  )
```

NOTE This example shows how each call to `getLine` is followed by a lambda expression that captures the value of the string returned `getLine`: the name for the first call, and the town for the second. In the case of the first call, the return value is not used immediately; we do not want to pass it to the next call `putStr`. That is why we need to make sure that the scope of the lambda expression that takes `name` as its argument includes all the following actions, or at least all the following actions that need access to the value of `name` (which in this case is the same thing).

`do` blocks

Code written using monad operations is often ugly, and writing it is usually tedious. To address both concerns, Haskell provides `do` blocks. These are merely syntactic sugar for sequences of monad operations, but they make the code much more readable and easier to write.

A *do block* starts with the keyword `do`, like this:

```
hello = do
    putStr "Hello, "
    putStrLn "world"
```

– section 16 slide 12 –

`do` block components

Each element of a `do` block can be

- an I/O action that returns an ignored value, usually of type `()`, such as the calls to `putStr` and `putStrLn` below (just call the function);
- an I/O action whose return value is used to bind a variable, (use *var* `<-` *expr* to bind the variable);
- bind a variable to a non-monadic value (use `let` *var* `=` *expr* (no `in`)).

```
greet :: IO ()
greet = do
    putStr
        "Greetings! What is your name? "
    name <- getLine
    putStr "Where are you from? "
    town <- getLine
    let msg = "Welcome, " ++ name ++
        " from " ++ town
    putStrLn msg
```

NOTE Each do block element has access to all the variables defined by previous actions but not later ones.

Let clauses in do blocks do not end with the keyword `in`, since the variable defined by such a let clause is available in `all` later operations in the block, not just the immediately following opeation.

This notation is so much more convenient to use than raw monad operations that raw monad operations occur in real Haskell programs only rarely.

– section 16 slide 13 –

Operator priority problem

Unfortunately, the following line of code does not work:

```
    putStrLn "Welcome, " ++ name ++
        " from " ++ town
```

The reason is that due to its system of operator priorities, Haskell thinks that the main function being invoked here is not `putStrLn` but `++`, with its left argument being `putStrLn "Welcome, "`.

This is also the reason why Haskell accepts only the second of the following equations. It parses the left hand side of the first equation as `(len x):xs`, not as `len (x:xs)`.

```
len x:xs = 1 + len xs
len (x:xs) = 1 + len xs
```

– section 16 slide 14 –

Working around the operator priority problem

There are two main ways to fix this problem:

```
putStrLn ("Welcome, " ++ name ++
    " from " ++ town)
putStrLn $ "Welcome, " ++ name ++
    " from " ++ town
```

The first simply uses parentheses to delimit the possible scope of the `++` operator.

The second uses another operator, `$`, which has lower priority than `++`, and thus binds less tightly.

The main function invoked on the line is thus `$`. Its first argument is its left operand: the function `putStrLn`, which is of type `String -> IO ()`. Its second argument is its right operand: the expression `"Welcome, " ++ name ++ " from " ++ town`, which is of type `String`.

`$` is of type `(a -> b) -> a -> b`. It applies its first argument to its second argument, so in this case it invokes `putStrLn` with the result of the concatenation.

– section 16 slide 15 –

`return`

If a function does I/O and returns a value, and the code that computes the return value does not do I/O, you will need to invoke the `return` monad operation as the last operation in the `do` block.

```
main :: IO ()
main = do
    putStrLn "Please input a string"
    len <- readlen
    putStrLn $
        "The length of that string is "
        ++ show len
\par
\vspace2mm
readlen :: IO Int
readlen = do
    str <- getLine
    return (length str)
```


Section 17: More on monads

I/O actions as descriptions

Haskell programmers usually think of functions that return values of type `IO t` as doing I/O as well as returning a value of type `t`. While this is usually correct, there are some situations in which it is not accurate enough.

The correct way to think about such functions is that they return two things:

- a value of type `t`, and
- a *description* of an I/O operation.

The monadic operator `>>=` can then be understood as taking descriptions of two I/O operations, and returning a description of those two operations being executed in order.

The monadic operator `return` simply associates a description of a do-nothing I/O operation with a value.

Description to execution: theory

Every complete Haskell program must have a function named `main`, whose signature should be

```
main :: IO ()
```

As in C, this is where the program starts execution. Conceptually,

- the OS starts the program by invoking the Haskell runtime system;
- the runtime system calls `main`, which returns a description of a sequence of I/O operations; and

- the runtime system executes the described sequence of I/O operations.

Description to execution: practice

In actuality, the compiler and the runtime system together ensure that each I/O operation is executed as soon as its description has been computed,

- *provided* that the description is created in a context which guarantees that the description will end up in the list of operation descriptions returned by `main`, and
- *provided* that all the previous operations in that list have also been executed.

The provisions are necessary since

- you don't want to execute an I/O operation that the program does not actually call for, and
- you don't want to execute I/O operations out of order.

Example: printing a table of squares directly

```
main ::IO ()
main = do
    putStrLn "Table of squares:"
    print_table 1 10
\par
\vspace2mm
print_table :: Int -> Int -> IO ()
print_table cur max
    | cur > max = return ()
    | otherwise = do
        putStrLn (table_entry cur)
        print_table (cur+1) max
\par
\vspace2mm
table_entry :: Int -> String
table_entry n = (show n) ++ "^2 = "
    ++ (show (n*n))
```

NOTE The definition of `print_table` uses *guards*. If `cur > max`, then the applicable right hand side is the one that follows that guard expression; if `cur <= max`, then the applicable right hand side is the one that follows the keyword `otherwise`.

Non-immediate execution of I/O actions

Just because you have created a description of an I/O action, does not mean that this I/O action will eventually be executed.

Haskell programs can pass around descriptions of I/O operations. They cannot peer into a description of an I/O operation, but they can nevertheless do things with them, such as

- build up lists of I/O actions, and
- put I/O actions into binary search trees as values.

Those lists and trees can then be processed further, and programmers can, if they wish, take the descriptions of I/O actions out of those data structures, and have them executed by including them in the list of actions returned by `main`.

Example: printing a table of squares indirectly

```
main = do
    putStrLn "Table of squares:"
    let row_actions =
        map show_entry [1..15]
    execute_actions (take 10 row_actions)
\par
\vspace2mm
table_entry :: Int -> String
table_entry n = (show n) ++ "^2 = "
    ++ (show (n*n))
\par
\vspace2mm
show_entry :: Int -> IO ()
show_entry n = do putStrLn (table_entry n)
```

```
\par
\vspace2mm
execute_actions :: [IO ()] -> IO ()
execute_actions [] = return ()
execute_actions (x:xs) = do
    x
    execute_actions xs
```

NOTE The `take` function from the Haskell prelude returns the first few elements of a list; the number of elements it should return is specified by the value of its first argument.

Input, process, output

A typical batch program reads in its input, does the required processing, and prints its output.

A typical interactive program goes through the same three stages once for each interaction.

In most programs, the vast majority of the code is in the middle (processing) stage.

In programs written in imperative languages like C, Java, and Python, the type of a function (or procedure, subroutine or method) does not tell you whether the function does I/O.

In Haskell, it does.

I/O in Haskell programs

In most Haskell programs, the vast majority of the functions are not I/O functions and they do no input or output. They merely build, access and transform data structures, and do calculations. The code that does I/O is a thin veneer on top of this bulk.

This approach has several advantages.

- A unit test for a non-IO function is a record of the values of the arguments and the expected value of the result. The test driver can read in those values, invoke the function, and check whether the result matches. The test driver can be a human.
- Code that does no I/O can be rearranged. Several optimizations exploit this fact.
- Calls to functions that do no I/O can be done in parallel. Selecting the best calls to parallelize is an active research area.

Debugging Haskell functions

Suppose you are working on a function `f`, and tests reveal the function to have a bug.

Ideally, you would have unit tested all the other functions `f` calls and found them to be correct, which would narrow the potential scope of the bug down to `f` itself. In practice, you probably didn't do this.

You can try to debug `f` by calling the functions `f` calls with the arguments that the failed test case would have invoked them with, and see what happens. This is effectively unit testing on demand.

Unfortunately, sometimes you need lots of information before you can select a promising debugging approach.

Debugging printfs

In a program written in C, you could get this information by adding debugging printfs to `f` and/or the functions it calls.

In a program written in Haskell, things are not always so simple.

- If `f` does I/O, which means that it returns a description of an I/O operation that you know will be executed, then you can simply print the diagnostics you need as part of that operation.
- If `f` does not do I/O, or if the I/O operation whose description it returns is not executed, this will not work.

This is where the function `unsafePerformIO` comes in.

– section 17 slide 10 –

`unsafePerformIO`

The type of `unsafePerformIO` is `IO t -> t`.

- You give it as argument an I/O operation, which means a function of type `IO t`. `unsafePerformIO` calls this function.
- The function will return a value of type `t` and a description of an I/O operation.
- `unsafePerformIO` executes the described I/O operation and returns the value.

Here is an example:

```
sum :: Int -> Int -> Int
sum x y = unsafePerformIO $ do
    putStrLn ("summing " ++
        (show x) ++ " and " ++ (show y))
    return (x + y)
```

NOTE As its name indicates, the use of `unsafePerformIO` is unsafe in general. This is because you the program does not indicate how the I/O operations performed by an invocation of `unsafePerformIO` fit into the sequence of operations executed by `main`. This is why `unsafePerformIO` is *not* defined in the prelude. This means that your code will not be able to call it unless you import the module that defines it, which is `GHC.IOBase`.

– section 17 slide 11 –

The State monad

The `State` monad is useful for computations that need to thread information throughout the computation. It allows such information to be transparently passed around a computation, and accessed and replaced when needed. That is, it allows an imperative style of programming without losing Haskell's declarative semantics.

This simple code, which adds 1 to each element of a tree, does not need a monad:

```
data Tree a
    = Empty
    | Node (Tree a) a (Tree a)
    deriving Show
type IntTree = Tree Int
\par
\vspace2mm
incTree :: IntTree -> IntTree
incTree Empty = Empty
incTree (Node l e r) =
    Node (incTree l) (e + 1) (incTree r)
```

– section 17 slide 12 –

Threading state

If we instead wanted to add 1 to the leftmost element, 2 to the next element, and so on, we would need to pass an integer into our function saying what to add, but also we need to pass an integer out, saying what to add to the *next* element. This requires more complex code:

```
incTree1 :: IntTree -> IntTree
incTree1 tree = fst (incTree1' tree 1)
\par
\vspace2mm
incTree1' :: IntTree -> Int -> (IntTree, Int)
incTree1' Empty n = (Empty, n)
incTree1' (Node l e r) n =
    let (newl, n1) = incTree1' l n
        (newr, n2) = incTree1' r (n1 + 1)
    in  (Node newl (e+n1) newr, n2)
```

NOTE Given a tuple of two elements, the `fst` builtin function returns its first element, and the `snd` builtin function returns its second element.

– section 17 slide 13 –

Introducing the `State` monad

The `State` monad abstracts the type `s -> (v,s)`, hiding away the `s` part. Haskell's `do` notation allows us to focus on the `v` part of the computation while ignoring the `s` part where not relevant.

```
incTree2 :: IntTree -> IntTree
incTree2 tree =
    fst (runState (incTree2' tree) 1)
\par
\vspace2mm
incTree2' :: IntTree -> State Int IntTree
incTree2' Empty = return Empty
incTree2' (Node l e r) = do
    newl <- incTree2' l
    n <- get    -- gets the current state
    put (n + 1) -- sets the current state
    newr <- incTree2' r
    return (Node newl (e+n) newr)
```

Abstracting the state operations

In this case, we do not need the full generality of being able to update the integer state in arbitrary ways; the only update operation we need is an increment. We can therefore provide a version of the state monad that is specialized for this task. Such specialization provides useful documentation, and makes the code more robust.

```
type Counter = State Int
\par
\vspace2mm
withCounter :: Int -> Counter a -> a
withCounter init f = fst (runState f init)
\par
\vspace2mm
nextCount :: Counter Int
nextCount = do
    n <- get
    put (n + 1)
    return n
```

Using the counter

Now the code that uses the monad is even simpler:

```
incTree3 :: IntTree -> IntTree
incTree3 tree = withCounter 1 (incTree3' tree)
\par
\vspace2mm
incTree3' :: IntTree -> Counter IntTree
incTree3' Empty = return Empty
incTree3' (Node l e r) = do
    newl <- incTree3' l
    n <- nextCount
    newr <- incTree3' r
    return (Node newl (e+n) newr)
```

Section 18: Lazyness

Eager vs lazy evaluation

In a programming language that uses *eager evaluation*, each expression is evaluated as soon as it gets bound to a variable, either explicitly in an assignment statement, or implicitly during a call. (A call implicitly assigns each actual parameter expression to the corresponding formal parameter variable.)

In a programming language that uses *lazy evaluation*, an expression is not evaluated until its value is actually needed. Typically, this will be when

- the program wants the value as input to an arithmetic operation, or
- the program wants to match the value against a pattern, or
- the program wants to output the value.

Almost all programming languages use eager evaluation. Haskell uses lazy evaluation.

Lazyness and infinite data structures

Lazyness allows a program to work with data structures that are conceptually infinite, as long as the program looks at only a finite part of the infinite data structure.

For example, `[1..]` is a list of all the positive numbers. If you attempt to print it out, the printout will be infinite, and will take infinite time, unless you interrupt it.

On the other hand, if you want to print only the first `n` positive numbers, you can do that with `take n [1..]`.

Even though the second argument of the call to `take` is infinite in size, the call takes finite time to execute.

NOTE The expression `[1..]` has the same value as `all_ints_from 1`, given the definition

```
all_ints_from :: Integer -> [Integer]
all_ints_from n = n:(all_ints_from (n+1))
```


The sieve of Eratosthenes

```
-- returns the (infinite) list of all primes
all_primes :: [Integer]
all_primes = prime_filter [2..]
\par
\vspace2mm
prime_filter :: [Integer] -> [Integer]
prime_filter [] = []
prime_filter (x:xs) =
    x:prime_filter (filter (not . (`divisibleBy` x)) xs)
\par
\vspace2mm
-- n `divisibleBy` d means n is evenly divisible by d
divisibleBy n d = n `mod` d == 0
```

NOTE The input to `prime_filter` is a list of integers which share the property that they are not evenly divisible by any prime that is smaller than the first element of the list.

The invariant is trivially true for the list [2..], since there are no primes smaller than 2.

Since the smallest element of such a list cannot be evenly divisible by any number smaller than itself, it must be a prime. Therefore filtering multiples of the first element out of the input sequence before giving the filtered sequence as input to the recursive call to `prime_filter` maintains the invariant.

Using `all`$_{p}$*rimes*

To find the first n primes:

```
take n all_primes
```

To find all primes up to n:

```
takeWhile (<= n) all_primes
```

Lazyness allows the programmer of `all_primes` to concentrate on the function's task, *without* having to also pay attention to exactly *how* the program wants to decide how many primes are enough.

Haskell automatically interleaves the computation of the primes with the code that determines how many primes to compute.

Representing unevaluated expressions

In a lazy programming language, expressions are not evaluated until you need their value. However, until then, you do need to remember the code whose execution will compute that value.

In Haskell implementations that compile Haskell to C (this includes GHC), the data structure you need for that is a pointer to a C function, together with all the arguments you will need to give to that C function.

This representation is sometimes called a *suspension*, since it represents a computation whose evaluation is temporarily suspended.

It can also be called a *promise*, since it also represents a promise to carry out a computation if its result is needed.

Historically inclined people can also call it a *thunk*, because that was the name of this construct in the first programming language implementation that used it. That language was Algol-60.

NOTE The word "thunk" can actually refer to several programming language implementation constructs, of which this is only one. Think of "thunk" as the compiler writer's equivalent of the mechanical engineer's word "gadget": they can both be used to refer to anything small and clever.

Parametric polymorphism

Parametric polymorphism is the name for the form of polymorphism in which types like `[a]` and `Tree k v`, and functions like `length` and `insert_bst`, include type variables, and the types and functions work identically regardless of what types the type variables stand for.

The implementation of parametric polymorphism requires that the values of all types be representable in the same amount of memory. Without this, the code of e.g. `length` wouldn't be able to handle lists with elements of all types.

That "same amount of memory" will typically be the word size of the machine, which is the size of a pointer. Anything that does not fit into one word is represented by a pointer to a chunk of memory on the heap.

Given this fact, the arguments of the function in a suspension can be stored in an array of words, and we can arrange for all functions in suspensions to take their arguments from a single array of words.

NOTE Parametric polymorphism is the form of polymorphism on which the type systems of languages like Haskell are based. Object-oriented languages like Java are based on a different form of polymorphism, which is usually called "inclusion polymorphism" or "subtype polymorphism".

The "same amount of memory" obviously does not include the memory needed by the pointed-to heap cells, or the cells *they* point to directly or indirectly. A value such as `[1, 2, 3, 4, 5]` will require several heap cells; in this case, these will be five cons cells and (depending on the details of the implementation) maybe one cell for the nil at the end. (The nonempty list constructor `:` is usually pronounced "cons", while the empty list constructor `[]` is usually pronounced "nil".) However, the whole value can be represented by a pointer to the first cons cell.

Evaluating lazy values only once

Many functions use the values of some variables more

than once. This includes `takeWhile`, which uses `x` twice:

```
takeWhile p (x:xs)
    | p x       = x : takeWhile p xs
    | otherwise = []
```

You need to know the value of `x` to do the test `p x`, which requires calling the function in the suspension representing `x`.

If the test succeeds, you will again need to know the value of `x` to put it at the front of the output list.

To avoid redundant work, you want the first call to `x`'s suspension to record the result of the call, and you want all references to `x` after the first to get its value from this record.

Therefore once you know the result of the call, you don't need the function and its arguments anymore.

Call by need

Operations such as printing, arithmetic and pattern matching start by ensuring their argument is at least partially evaluated.

They will make sure that at least the top level data constructor of the value is determined. However, the arguments of that data constructor may remain suspensions.

For example, consider the match of the second argument of `takeWhile` against the patterns `[]` and `(p:ps)`. If the original second argument is a suspension, it must be evaluated enough to ensure its top-level constructor is determined. If it is `x:xs`, then the first argument must be applied to `x`. Whether `x` needs to be evaluated will depend on what the first argument (function) does.

This is called "call by need", because function arguments (and other expressions) are evaluated only when their value is needed.

Control structures and functions

```
(a) ... if (x < y) f(x); else g(y); ...
(b) ... ite(x < y, f(x), g(y)); ...
\par
\vspace{2mm}
int ite(bool c, int t, int e)
{ if (c) then return t; else return e; }
```

In C, (a) will generate a call to only one of `f` and g, but (b) will generate a call to both.

```
(c) ... if x < y then f x else g y ...
(d) ... ite (x < y) (f x) (g y) ...
\par
\vspace2mm
ite :: Bool -> a -> a -> a
ite c t e = if c then t else e
```

In Haskell, (c) will generate a call to only one of `f` and g, and thanks to lazyness, this is also true for (d).

NOTE The Haskell implementation of if-then-else calls `evaluate_suspension` on the suspension representing the condition. If the condition's value is `True`, it will then call `evaluate_suspension` on the suspension representing then part, and return its value; if the condition's value is `False`, it will call `evaluate_suspension` on the suspension representing the else part, and return its value. In each case, the suspension for the other part will remain unevaluated. Roughly speaking, both (c) and (d) are implemented this way.

Implementing control structures as functions

Without lazyness, using a function instead of explicit code such as a sequence of if-then-elses could get unnecessary non-termination or at least unnecessary slowdowns.

Lazyness' guarantee that an expression will not be evaluated if its value is not needed allows programmers to define their own control structures as functions.

For example, you can define a control structure that returns the value of one of three expressions, with the expression chosen based on whether an expression is less than, equal to or greater than zero like this:

```
ite3 :: (Num a) a -> b -> b -> b -> b
ite3 x lt eq gt
    | x < 0  = lt
    | x == 0 = eq
    | x > 0  = gt
```

NOTE This ability to define new control structures can come in handy if you find yourself repeatedly writing code with the same nontrivial decision-making structure. However, such situations are pretty rare.

The kind of three-way branch shown by this example was the main way to implement choice in the first widely-used high level programming language, the original dialect of Fortran. That version of the if statement specified three labels; which one control jumped to next depended on the value of the control expression.

Using lazyness to avoid unnecessary work

```
min = head . sort
```

On the surface, this looks like a very wasteful method for computing the minimum, since sorting is usually done with an O(n^2) or O(n *logn*) algorithm, and `min` should be doable with an O(n) algorithm.

However, in this case, the evaluation of the sorted list can stop after the materialization of the first element.

If `sort` is implemented using selection sort, this is just a somewhat higher overhead version of the direct code for `min`.

Multiple passes

```
output_prog chars = do
    let anno_chars =
        annotate_chars 1 1 chars
    let tokens = scan anno_chars
    let prog = parse tokens
    let prog_str = show prog
    putStrLn prog_str
```

This function takes as input one data structure (`chars`) and calls for the construction of four more (`anno_chars`, `tokens`, `prog` and `prog_str`).

This kind of pass structure occurs frequently in real programs.

The effect of lazyness on multiple passes

With eager evaluation, you would completely construct each data structure before starting construction of the next.

The maximum memory needed at any one time will be the size of the largest data structure (say pass n), plus the size of any part of the previous data structure (pass $n-1$) needed to compute the last part of pass n. All other memory can be garbage collected before then.

With lazy evaluation, execution is driven by `putStrLn`, which needs to know what the next character to print (if any) should be. For each character to be printed, the program will materialize the parts of those data structures needed to figure that out.

The memory demand at a given time will be given by the tree of suspensions from earlier passes that you need to materialize the rest of the string to be printed. The maximum memory demand can be significantly less than with eager evaluation.

NOTE The memory requirements analysis above assumes that the code that constructs pass n's data structure uses only data from pass $n-1$, and does not need access to data from pass $n-2$, $n-3$ etc.

If the last pass builds a data structure instead of printing out the data structure built by the second-last pass, then you will need to add the

memory needed by the part of the last data structure constructed so far to the memory needed at any point in time. This will increase the maximum memory demand with lazy evaluation, but it can still be less than with eager evaluation.

Lazy input

In Haskell, even input is implemented lazily.

Given a filename, `readFile` returns the contents of the file as a string, but it returns the string lazily: it reads the next character from the file only when the rest of the program needs that character.

```
parse_prog_file filename = do
    fs <- readFile filename
    let tokens =
        scan (annotate_chars 1 1 chars)
    return (parse_prog [] tokens)
```

When the main module calls `parse_prog_file`, it gets back a tree of suspensions.

Only when those suspensions start being forced will the input file be read, and each call to `evaluate_suspension` on that tree will cause only as much to be read as is needed to figure out the value of the forced data constructor.

NOTE The `readFile` function is defined in the prelude.

Section 19: Performance

Effect of lazyness on performance

Lazyness adds two sorts of overhead that slow down programs.

- The execution of a Haskell program creates a lot of suspensions, and most of them are evaluated, so eventually they also need to be unpacked.
- Every access to a value must first check whether the value has been materialized yet.

However, lazyness can also speed up programs by avoiding the execution of computations that take a long time, or do not terminate at all.

Whether the dominant effect is the slowdown or the speedup will depend on the program and what kind of input it typically gets.

The usual effect is something like lotto: in most cases you lose a bit, but sometimes you win a little, and in some rare cases you win a lot.

Strictness

Theory calls the value of an expression whose evaluation loops infinitely or throws an exception "bottom", denoted by the symbol $\perp$.

A function is *strict* if it always needs the values of all its arguments. In formal terms, this means that if any of its arguments is $\perp$, then its result will also be $\perp$.

The addition function `+` is strict. The function `ite` from earlier in this section is nonstrict.

Some Haskell compilers including GHC include *strictness analysis*, which is a compiler pass whose job is to analyze the code of the program and figure out which of its functions are strict and which are nonstrict.

When the Haskell code generator sees a call to a strict function, instead of generating code that creates a suspension, it can generate the code that an imperative language compiler would generate: code that evaluates all the arguments, and then calls the function.

NOTE Eager evaluation is also called strict evaluation, while lazy evaluation is also called nonstrict evaluation.

Unpredictability

Besides generating a slowdown for most programs, lazyness also makes it harder for the programmer to understand where the program is spending most of its time and what parts of the program allocate most of its memory.

This is because small changes in exactly where and when the program demands a particular value can cause great changes in what parts of a suspension tree are evaluated, and can therefore cause great changes in the time and space complexity of the program. (Lazy evaluation is also called *demand driven* computation.)

The main problem is that it is very hard for programmers to be simultaneous aware of *all* the relevant details in the program.

Modern Haskell implementations come with sophisticated profilers to help programmers understand the behavior of their programs. There are profilers for both time and for memory consumption.

Deforestation

As we discussed earlier, many Haskell programs have code that follows this pattern:

- You start with the first data structure, ds1.
- You traverse ds1, generating another data structure, ds2.
- You traverse ds2, generating yet another data structure, ds3.

If the programmer can restructure the code to compute ds3 directly from ds1, this should speed up the program, for two reasons:

- the new version does not need to create ds2, and
- the new version does one traversal instead of two.

Since the eliminated intermediate data structures are often trees of one kind or another, this optimization idea is usually called *deforestation*.

`filter_map`

```
filter_map :: (a -> Bool) -> (a -> b)
    -> [a] -> [b]
filter_map _ _ [] = []
filter_map f m (x:xs) =
    let newxs = filter_map f m xs in
    if f x then (m x):newxs else newxs
\par
\vspace2mm
one_pass xs = filter_map is_even triple xs
two_pass xs = map triple (filter is_even xs)
```

The `one_pass` function performs exactly the same task as the `two_pass` function, but it does the job with one list traversal, not two, and does not create an intermediate list.

One can also write similarly deforested combinations of many other pairs of higher order functions, such as map and foldl.

Computing standard deviations

```
four_pass_stddev :: [Double] -> Double
four_pass_stddev xs =
  let
      count = fromIntegral (length xs)
      sum = foldl (+) 0 xs
      sumsq = foldl (+) 0
        (map square xs)
  in
    (sqrt (count * sumsq - sum * sum)) /
        count
\par
\vspace2mm
square :: Double -> Double
square x = x * x
```

This is the simplest approach to writing code that computes the standard deviation of a list. However, it traverses the input list three times, and it also traverses a list of that same length (the list of squares) once.

Computing standard deviations in one pass

```
data StddevData =
  SD Double Double Double
\par
\vspace2mm
one_pass_stddev :: [Double] -> Double
one_pass_stddev xs =
  let
    init_sd = SD 0.0 0.0 0.0
    update_sd (SD c s sq) x =
        SD (c + 1.0) (s + x) (sq + x*x)
    SD count sum sumsq =
      foldl update_sd init_sd xs
  in
    (sqrt (count * sumsq - sum * sum)) /
        count
```

NOTE This is an example of a call to `foldl` in which the base is of one type (`StddevData`) while the list elements are of another type (`Double`).

It is also an example of a let clause that defines an auxiliary function, in this case `update_sd`, and one in which the last part picks up the values of the arguments of the `SD` data constructor in three variables.

Cords

Repeated appends to the end of a list take time that is quadratic in the final length of the list.

In imperative languages, you would avoid this quadratic behavior by keeping a pointer to the tail of the list, and destructively updating that tail.

In declarative languages, the usual solution is to switch from lists to a data structure that supports appends in constant time. These are usually called *cords*. This is one possible cord design; there are several.

```
data Cord a
    = Nil
    | Leaf a
    | Branch (Cord a) (Cord a)
\par
\vspace2mm
append_cords :: Cord a -> Cord a -> Cord a
append_cords a b = Branch a b
```

Converting cords to lists

The obvious algorithm to convert a cord to a list is

```
cord_to_list :: Cord a -> [a]
cord_to_list Nil = []
cord_to_list (Leaf x) = [x]
cord_to_list (Branch a b) =
    (cord_to_list a) ++ (cord_to_list b)
```

Unfortunately, it suffers from the exact same performance problem that cords were designed to avoid.

The cord `Branch (Leaf 1) (Leaf 2)` that the last equation converts to a list may itself be one branch of a bigger cord, such as `Branch (Branch (Leaf 1) (Leaf 2)) (Leaf 3)`.

The list `[1]`, converted from `Leaf 1`, will be copied twice by `++`, once for each `Branch` data constructor in whose first operand it appears.

Accumulators

With one exception, all leaves in a cord are followed by another item, but the second equation puts an empty list behind all leaves, which is why all but one of the lists it creates will have to be copied again. The other two equations make the same mistake for empty and branch cords.

Fixing the performance problem requires telling the conversion function what list of items follows the cord currently being converted. This is easy to arrange using an accumulator.

```
cord_to_list :: Cord a -> [a]
cord_to_list c = cord_to_list' c []
\par
\vspace2mm
cord_to_list' :: Cord a -> [a] -> [a]
cord_to_list' Nil rest = rest
cord_to_list' (Leaf x) rest = x:rest
cord_to_list' (Branch a b) rest =
    cord_to_list' a (cord_to_list' b rest)
```


Sortedness check

The obvious way to write code that checks whether a list is sorted:

```
sorted1 :: (Ord a) => [a] -> Bool
sorted1 [] = True
sorted1 (_:[]) = True
sorted1 (x1:x2:xs) = x1 <= x2 && sorted1 (x2:xs)
```

However, the code that looks at each list element handles three alternatives (lists of length zero, one and more).

It does this because each sortedness comparison needs *two* list elements, not one.

A better sortedness check

```
sorted2 :: (Ord a) => [a] -> Bool
sorted2 [] = True
sorted2 (x:xs) = sorted_lag x xs
\par
\vspace2mm
sorted_lag :: (Ord a) => a -> [a] -> Bool
sorted_lag _ [] = True
sorted_lag x1 (x2:xs) = x1 <= x2 && sorted_lag x2 xs
```

In this version, the code that looks at each list element handles only two alternatives. The value of the previous element, the element that the current element should be compared with, is supplied separately.

Optimisation

You can use `:set +s` in GHCi to time execution.

```
Prelude> :l sorted
[1 of 1] Compiling Sorted      ( sorted.hs, interpreted )
Ok, modules loaded: Sorted.
*Sorted> :set +s
*Sorted> sorted1 [1..100000000]
True
(50.11 secs, 32,811,594,352 bytes)
*Sorted> sorted2 [1..100000000]
True
(40.76 secs, 25,602,349,392 bytes)
```

The `sorted2` version is about 20% faster and uses 22% less memory.

Optimisation

However, the Haskell compiler is very sophisticated. After doing
`ghc -dynamic -c -O3 sorted.hs`, we get this:

```
Prelude> :l sorted
Ok, modules loaded: Sorted.
Prelude Sorted> :set +s
Prelude Sorted> sorted1 [1..100000000]
True
(2.89 secs, 8,015,369,944 bytes)
Prelude Sorted> sorted2 [1..100000000]
True
(2.91 secs, 8,002,262,840 bytes)
```

Compilation gives a *factor* of 17 speedup and a factor of 3 memory savings. It also removes the difference between `sorted1` and `sorted2`. *Always* benchmark your compiled code when trying to speed it up.

Storing functions in data structures

One reason why functional programmers can do things with plain old libraries that object-oriented programmers typically do not is that in functional programs, it is easy to store functions in data structures.

For example, a query on an XML document may want to return the URLs of all anchor tags whose associated text is "here".

In a functional language, one can implement this simply by passing a closure for a function that tests the text of the tag for equality with a given value.

The code needed for this in an object-oriented language would be significantly more complicated.

Autocomplete

Modern Unix shells implement file name completion. After typing the first few characters of a file name, the user can type the completion character (which is usually set to be TAB), and the shell will type the rest of the filename. If there are two or more filenames with the given prefix, the shell will extend the current text to their common prefix.

One way to implement this would be using a function that takes

- a set of strings giving the set of filenames in the current directory, which could be "compose.hs", "compose.hi", "cord.hs", and "countnodes.hs", and
- a string that says what the user has typed so far, which could be "com".

It would find all the filename strings that agree with the characters typed so far ("compose.hs" and "compose.hi"), and return the longest common prefix of these strings ("compose.h" in this case).

Autocomplete functions

Consider a system which allows users to type in several windows. In one window the system expects email addresses, in another the names of people, in a third filenames, and in a fourth arbitrary text.

Autocomplete is applicable in all these windows, just with different sets of strings.

- For the window expecting email addresses, the strings could come from the contacts list of your email client.
- For the window expecting names, the strings could come from there as well, or they could come from the list of your friends on Facebook, or ...
- For the window expecting arbitrary text, the strings could come from any of the previous sources, and they could include the words you have already typed in that window.

Autocomplete functions

You could implement all this by having a dictionary that maps the ids of windows to a list of functions that each return a set of strings.

```
type SourceMap =
  Map WindowId [ProgState -> Set String]
\par
\vspace2mm
get_sources ::
  ProgState -> SourceMap -> WindowId
    -> Set String
get_sources ps source_map win =
  let maybe_funcs =
    Data.Map.lookup win source_map in
  case maybe_funcs of
    Nothing ->
      Data.Set.empty
    Just funcs ->
      let sources =
        apply_prog_state ps funcs in
      union_list sources
```

NOTE The functions called by this code could be coded like this:

```
apply_prog_state :: ProgState ->
    [ProgState -> a] -> [a]
apply_prog_state _ [] = []
apply_prog_state ps (f:fs) =
    (f ps):(apply_prog_state ps fs)
\par
\vspace2mm
union_list :: Ord a => [Set a] -> Set a
union_list [] = Data.Set.empty
union_list (s:ss) =
    Data.Set.union s (union_list ss)
```

Storing functions in data structures in C

The Mercury debugger is implemented partly in C, partly in Mercury. The part that handles debugger commands is implemented in C. It has a table mapping the name of a command to the C function implementing the command and the C function providing autocompletion for its arguments.

```
typedef struct {
    const char       *MR_cmd_name;
    MR_TraceCmdFunc  *MR_cmd_function;
    MR_MakeCompleter *MR_cmd_arg_completer;
} MR_TraceCmdInfo;
\par
\vspace{2mm}
static const MR_TraceCmdInfo
    MR_trace_command_infos[] =
{
    { "print", MR_trace_cmd_print,
        MR_var_completer },
    ...
};
```

NOTE This shows that one can store functions in data structures in C as well.

The type `MR_MakeCompleter` is defined elsewhere to be a function type.

All the identifiers in the Mercury runtime system start with the prefix "`MR_`", since this minimizes the risk of accidental collision between the names of types, functions, variables and structure fields defined by the Mercury system and the names of types, functions, variables and structure fields defined by C code linked into the Mercury program.

The part of the Mercury debugger that handles debugger commands is implemented in C and not in Mercury because some of the commands perform operations (such as jumping back in time to when a function was called) that cannot be expressed in Mercury.

Section 20: Exploiting the type system

Representation of C programs in `gcc`

The gcc compiler has one main data type to represent the code being compiled. The node type is a giant union which has different fields for different kinds of entities. A node can represent, amongst other things,

- a data type,
- a variable,
- an expression or
- a statement.

Every link to another part of a program (such as the operands of an operator) is a pointer to a tree node of this can-represent-everything type.

When Stallman chose this design in the 1980s, he was a Lisp programmer. Lisp does not have a static type system, so the Blub paradox applies here in reverse: even C has a better static type system than Lisp. It's up to the programmer to design types to exploit the type system.

NOTE The giant union is definitely very big: in gcc 4.4.1, it has 40 alternatives, many more than the four listed above.

Like Python, Lisp does have a type system, but it operates only at runtime.

Representation of if-then-elses

For if-then-else expressions such as `(x > y) ? x : y`, the relevant union field is a structure that has an array of operands.

For if-then-else expressions, the array should have exactly three elements (the condition, the then part and the else part), and all three should be expressions.

This representation is subject to two main kinds of error.

- The array of operands could have the wrong number of operands.
- Any operand in the array could point to the wrong kind of tree node.

`gcc` has extensive infrastructure designed to detect these kinds of errors, but this infrastructure itself has three problems:

- it makes the source code harder to read and write;
- if enabled, it slows down gcc by about 5 to 15%; and
- it detects violations only at runtime.

NOTE When trying to compiler some unusual C programs, usually those generated by a program rather than a programmer, you may gcc to abort with a message talking about an "internal compiler error". The usual reason for such aborts is that the compiler expected to find a node of a particular kind in a given position in the tree, but found a node of a different kind. In other words, the problem is the failure of a runtime type check. A better representation that took advantage of the type system would allow such bugs to be caught at compile time.

Exploiting the type system

A well designed representation using algebraic types is not vulnerable to either kind of error, and is not subject any of those three kinds of problems.

```
data Expr
    = Const Const
    | Var String
    | Binop Binop Expr Expr
    | Unop Unop Expr
    | Call String [Expr]
```

```
    | ITE Expr Expr Expr
\par
\vspace2mm
data Binop = Add | Sub | ...
data Unop = Uminus | ...
data Const
    = IntConst Int
    | FloatConst Double
    | ...
```

NOTE You can do much better than gcc's design even in C. A native C programmer would have chosen to have represent these four very different kinds of entities using four different main types, together with more types representing their components. This representation could start from something like the following.

```
typedef enum {
    EXPR_CONST, EXPR_VAR,
    EXPR_BINOP, EXPR_UNOP,
    EXPR_CALL, EXPR_ITE
} ExprKind;
\par
\vspace{2mm}
typedef struct expr_struct  Expr;
typedef union expr_union    ExprUnion;
typedef struct exprs_struct Exprs;
typedef struct const_struct Const;
typedef struct var_struct   Var;
\par
\vspace{2mm}
struct expr_struct {
    ExprKind    expr_kind;
    ExprUnion   *expr_union;
};
\par
\vspace{2mm}
union expr_union {
    Const                    *expr_const;
    Var                      *expr_var;
    struct unop_expr_struct  *expr_unop;
    struct binop_expr_struct *expr_binop;
    struct call_expr_struct  *expr_call;
    struct ite_expr_struct   *expr_ite;
};
\par
\vspace{2mm}
typedef enum {
    ADD, SUB, ...
} Binop;
\par
\vspace{2mm}
typedef enum {
    UMINUS, ...
} Unop;
\par
\vspace{2mm}
struct binop_expr_struct {
    Binop    *binop;
    Expr     binop_arg_1;
    Stmt     binop_arg_2;
};
\par
\vspace{2mm}
struct unop_expr_struct {
    Unop     *unop;
    Expr     unop_arg_1;
};
\par
\vspace{2mm}
struct call_expr_struct {
    char     *funcname;
    Exprs    func_args;
};
\par
\vspace{2mm}
struct ite_expr_struct {
    Expr     *cond;
    Stmt     then_part;
    Stmt     else_part;
};
\par
\vspace{2mm}
struct exprs_struct {
    Expr    *head;
    Exprs   *tail;
};
```

There is a paper titled "Statically typed trees in GCC" by Sidwell and Weinberg that describes the drawbacks of gcc's current program representation and proposes a roadmap for switching to a more type-safe representation, something like the representation shown above. Unfortunately, that roadmap has not been implemented, and probably never will be, partly because its implementation interfere considerably with the usual development of gcc.

Generic lists in C

```
typedef struct generic_list List;
struct generic_list {
    void    *head;
    List    *tail;
};
\par
\vspace{2mm}
...
List    *int_list;
List    *p;
int     item;
\par
\vspace{2mm}
for (p = int_list; p != NULL; p = p->tail) {
    item = (int) p->head;
    ... do something with item ...
}
```

NOTE Despite the name of the variable holding the list being `int_list`, C's type system cannot guarantee that the list elements are in fact integers.

Type system expressiveness

Programmers who choose to use generic lists in a C program need only one list type and therefore one set of functions operating on lists.

The downside is that every loop over lists needs to cast the list element to the right type, and this cast is a frequent source of bugs.

The other alternative in a C program is to define and use a separate list type for every different type of item that the program wants to put into a list. This is type safe, but requires repeated duplication of the functions that operate on lists. Any bugs in those those functions must be fixed in each copy.

Haskell has a very expressive type system that is increasingly being copied by other languages. Some OO/procedural languages now support generics. A few such languages (Rust, Swift, Java 8) support *option types*, like Haskell's `Maybe`. No well-known such languages support full algebraic types.

NOTE However, even in Haskell, some programming practices help the compiler to catch problems early (at compile time), and some do not.

Encoding semantic information in types

When designing data structures, programmers sometimes know that a given list of items may not be empty. For example, multimaps map a key to a *list* of values. If this list is empty, the key shouldn't be in the map.

```
-- obvious definition
type MultiMap k v = Map k [v]
\par
\vspace2mm
-- semantically tighter definition
type MultiMap k v = Map k (OneOrMore v)
data OneOrMore v = OneOrMore v [v]
```

NOTE `sorted_lag` checks whether its second argument is the empty list, but the second argument forms just the *tail* of the list represented by the two arguments together.

Using encoded semantic information

Suppose `map` is a `MultiMap`. Using the first definition, finding the maximum value associated with `key` requires code like this:

```
case lookup key map of
    [] -> error "empty item list"
    (head:tail) -> foldr max head tail
```

You need the test for `[]`, because sooner or later someone *will* create an empty value list, even if the documentation of `MultiMap` says they shouldn't.

If you are using the second definition, then you do not need to consider error handling at all.

```
let OneOrMore head tail = lookup key map in
foldr max head tail
```

NOTE The first definition of `MultiMap` has two ways for a key to have no associated value: the key may not be in the map, or it may be in the map with an empty list. A comment may say that an empty value list should never be used, but such comments cannot be enforced by the compiler. The only way to have the semantic restriction enforced by the compiler is to encode it in the type, as the second definition does.

Finding the maximum (or minimum) element of a list works intuitively only for nonempty lists. There are two main justifiable definitions of e.g. what the maximum of an empty list of integers should be: zero, for lists in which the integers are intended to be all nonnegative, and the most negative integer (which is $-2^{31} - 1$ for 32 bit platforms), for lists in which the integers have no such restriction. However, whichever definition your function for computing the maximum of a list uses, there will be times when a programmer using it will expect the other.

Units

One typical bug type in programs that manipulate physical measurements is unit confusion, such as adding 2 meters and 3 feet, and thinking the result is 5 meters. Mars Climate Orbiter was lost because of such a bug.

Such bugs can be prevented by wrapping the number representing the length in a data constructor giving its unit.

```
data Length = Meters Double
\par
\vspace2mm
meters_to_length :: Double -> Length
meters_to_length m = Meters m
\par
\vspace2mm
feet_to_length :: Double -> Length
feet_to_length f = Meters (f * 0.3048)
\par
\vspace2mm
```

```
add_lengths :: Length -> Length -> Length
add_lengths (Meters a) (Meters b) =
    Meters (a+b)
```

NOTE Making `Length` an abstract type, and exporting only type-safe operations to the rest of the program, improves safety even further.

Wrapping a data constructor around a number looks like adding overhead, since normally, each data constructor requires a cell of its own on the heap for itself and its arguments. However, types that have exactly one data constructor with exactly one argument can be implemented as if the data constructor were not there, eliminating the overhead. The Mercury language uses this representation scheme for types like this, and in some circumstances GHC can avoid the indirection.

For Mars Climate Orbiter, the Jet Propulsion Laboratory expected a contractor to provide thruster control data expressed in newtons, the SI unit of force, but the contractor provided the data in pounds-force, the usual unit of force in the old English system of measurement. Since one pound of force is 4.45 newtons, the thruster calculations were off by a factor of 4.45. The error was not caught because the data files sent from the contractor to JPL contained only numbers and not units, and because other kinds of tests that could have caught the error were eliminated in an effort to save money. The result was that Mars Climate Orbiter dived too steeply into the Martian atmosphere and burned up.

Different uses of one unit

Sometimes, you want to prevent confusion even between two kinds of quantities measured in the same units.

For example, many operating systems represent time as the number of seconds elapsed since a fixed *epoch*. For Unix, the epoch is 0:00am on 1 Jan 1970.

```
data Duration = Seconds Int
data Time = SecondsSinceEpoch Int
\par
```

```
\vspace2mm
add_durations ::
    Duration -> Duration -> Duration
add_durations (Seconds a) (Seconds b) =
    Seconds (a+b)
\par
\vspace2mm
add_duration_to_time ::
    Time -> Duration -> Time
add_duration_to_time
    (SecondsSinceEpoch sse) (Seconds t) =
        SecondsSinceEpoch (sse + t)
```

NOTE It makes sense to add together two durations or a time and a duration, but it does *not* make sense to add two times.

Different units in one type

Sometimes, you cannot apply a fixed conversion rate between different units. In such applications, each operation may need to do conversion on demand at whatever rate is applicable at the time of its execution.

```
data Money
    = USD_dollars Double
    | AUD_dollars Double
    | GBP_pounds Double
```

For financial applications, using `Double`s would not be a good idea, since accounting rules that precede the use of computers specify rounding methods (e.g. for interest calculations) that binary floating point numbers do not satisfy.

One workaround is to use fixed-point numbers, such as integers in which 1 represents not one dollar, but one one-thousandth of one cent.

NOTE Those accounting rules specified rounding algorithms for human accountants working with decimal numbers, not for automatic computers working with binary numbers. The difference is significant, since floating-point numbers cannot even represent exactly such simple but important fractions as one tenth and one percent.

Fat lists

A potential problem with lists is their memory consumption, which is typically two words (each of 32 or 64 bits) per element: one for the head element, and one for the tail, which is typically represented as a pointer to the next node, if any. The second word is overhead.

Using *fat lists*, in which a node can contain more than one item, can amortize this overhead over those items. In this example, we store three elements for each next pointer, and keep the leftover elements at the *front* of the fat list. This allows an element added to the front of the fat list to be combined with the left over elements.

Fat lists

The fat list type:

```
data FatList a
    = FCons0 (FatRest a)
    | FCons1 a (FatRest a)
    | FCons2 a a (FatRest a)
    deriving (Show, Eq)
\par
\vspace2mm
data FatRest a
    = FatNil
    | FatRest a a a (FatRest a)
    deriving (Show, Eq)
```

This type ensures that all but one nodes in a fat list will contain exactly three elements.

Using fat lists

One price for using fat lists is that functions defined for fat lists are much more complex than for standard lists.

```
length :: [a] -> Int
length [] = 0
length (_:lst) = 1 + length lst
\par
\vspace2mm
fatLength :: FatList a -> Int
fatLength (FCons0 fr)
  = fatLength' fr
fatLength (FCons1 x1 fr)
  = 1 + fatLength' fr
fatLength (FCons2 x1 x2 fr)
  = 2 + fatLength' fr
\par
\vspace2mm
fatLength' :: FatRest a -> Int
fatLength' FatNil
  = 0
fatLength' (FatRest e1 e2 e3 fr)
  = 3 + fatLength' fr
```

Converting fat lists to normal lists

```
fromFatList :: FatList a -> [a]
fromFatList (FCons0 fr)
  = fromFatList' fr
fromFatList (FCons1 x1 fr)
  = x1:fromFatList' fr
fromFatList (FCons2 x1 x2 fr)
  = x1:x2:fromFatList' fr
\par
\vspace2mm
fromFatList' :: FatRest a -> [a]
fromFatList' FatNil
  = []
fromFatList' (FatRest e1 e2 e3 fr)
  = e1:e2:e3:fromFatList' fr
```

Converting normal lists to fat lists

```
toFatList :: [a] -> FatList a
toFatList (x1:x2:x3:xt) = fcons3 x1 x2 x3 (toFatList xt)
toFatList [x1,x2]       = FCons2 x1 x2 FatNil
toFatList [x1]          = FCons1 x1 FatNil
toFatList []            = FCons0 FatNil
\par
\vspace2mm
fcons3 :: a -> a -> a -> FatList a -> FatList a
fcons3 e1 e2 e3 (FCons0 fr)
  = FCons0 (FatRest e1 e2 e3 fr)
fcons3 e1 e2 e3 (FCons1 e4 fr)
  = FCons1 e1 $ FatRest e2 e3 e4 fr
fcons3 e1 e2 e3 (FCons2 e4 e5 fr)
  = FCons2 e1 e2 $ FatRest e3 e4 e5 fr
```

Appending fat lists

```
fatAppend :: FatList a -> FatList a -> FatList a
fatAppend (FCons0 xr) ys
  = fatAppend' xr ys
fatAppend (FCons1 x1 xr) ys
  = fcons x1 $ fatAppend' xr ys
fatAppend (FCons2 x1 x2 xr) ys
  = fcons x1 $ fcons x2 $ fatAppend' xr ys
\par
\vspace2mm
fatAppend' FatNil ys = ys
fatAppend' (FatRest x1 x2 x3 xr) ys
  = fcons3 x1 x2 x3 $ fatAppend' xr ys
\par
\vspace2mm
fcons :: a -> FatList a -> FatList a
fcons x1 (FCons0 xr)
  = FCons1 x1 xr
fcons x1 (FCons1 x2 xr)
  = FCons2 x1 x2 xr
fcons x1 (FCons2 x2 x3 xr)
  = FCons0 $ FatRest x1 x2 x3 xr
```

Defining a typeclass

A second shortcoming of fat lists is that none of the standard list manipulation functions work for them. This cannot be helped.

However, it is possible to arrange for newly defined functions to work for both ordinary and fat lists. This is done by defining a typeclass for lists.

However, because list is a type constructor, not a type, we must use the `-XMultiParamTypeClasses` and `-XFlexibleInstances` GHC extensions.

This can be specified by adding the special comment form:

```
{-# OPTIONS_GHC -XMultiParamTypeClasses
                -XFlexibleInstances #-}
```

near the top of the file

Declaring a type class

A type class is declared by specifying all the functions that must be defined for all instances of that class. Our list typeclass will need functions to construct and deconstruct a list. We shall use the names given to these operations by Lisp: `nil`, `cons`, `car`, and `cdr`. We'll also specify functions to append and compute length, and to convert to and from ordinary lists.

```
class List l a where
  nil      :: l a
  cons     :: a -> l a -> l a
  car      :: l a -> a
  cdr      :: l a -> l a
  len      :: l a -> Int
  append   :: l a -> l a -> l a
  fromList :: [a] -> l a
  toList   :: l a -> [a]
```

Declaring an instance

Then we can declare our fat list type to be an instance of the List class just by specifying the definition of these functions for fat lists.

```
instance List FatList a where
  nil      = FCons0 FatNil
  cons     = fcons
  len      = fatLength
  append   = fatAppend
  fromList = toFatList
  toList   = fromFatList
\par
\vspace2mm
  car (FCons0 (FatRest x1 _ _ _)) = x1
  car (FCons1 x1 _)               = x1
  car (FCons2 x1 _ _)             = x1
\par
\vspace2mm
  cdr (FCons0 (FatRest _ x1 x2 xr)) = FCons2 x1 x2 xr
  cdr (FCons1 _ xr)                 = FCons0 xr
  cdr (FCons2 _ x1 xr)              = FCons1 x1 xr
```

Declaring an instance

Similarly to make ordinary list types instances of this class:

```
instance List [] a where
  nil      = []
  cons     = (:)
  len      = length
  append   = (++)
  fromList = id
  toList   = id
  car      = head
  cdr      = tail
```

With these declarations, functions defined to operate on instances of `List` can be applied to both kinds of lists (and any other instances).
