The University of Melbourne
School of Computing and Information Systems

COMP30020 / COMP90048
Declarative Programming

Section 0

Subject Introduction

Welcome to Declarative Programming

Lecturer: Peter Schachte

Contact information is available from the LMS.

There will be two pre-recorded one-hour lectures per week, plus one live one-hour practical meeting for questions, discussion, and demonstrations.

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

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

Grok

We use Grok to provide added self-paced instructional material, exercises, and self-assessment for both Haskell and Prolog.

You can access Grok by following the link from the subject LMS page.

If you are unable to access Grok or find that it is not working correctly, please email

Grok University Support <uni-support@groklearning.com>

from your university email account and explain the problem.

If you have questions regarding the Grok lessons or exercises, please post a message to the subject LMS discussion forum.

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. You are encouraged to ask questions, discuss, and actively engage in workshops. The more you put into workshops, the more you will get out of them.

Workshop exerciese will be available through Grok, so they can be undertaken even if you are not present in Australia. Sample solutions for each set of workshop exercises will also be available through Grok.

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

Resources

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 be available online through the LMS.

The recommended text (which is available online) is

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

Other recommended resources are listed on the LMS.

Assessment

The subject has the following assessment components:

- **0%** short Prolog project, due in Week 4 (optional)
- **15%** larger Prolog project, due in Week 6 or 7
- **0%** short Haskell project, is due in Week 8 or 9 (optional)
- **15%** larger Haskell project, due in Week 11 or 12
- **70%** two-hour written final examination

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

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

Academic Integrity

All assessment for this subject is *individual*; what you submit for assessment must be your work and *your work alone*.

It is important to distinguish project work (which is assessed) from tutorials and other unassessed exercises.

We are well aware that there are many online sources of material for subjects like this one; you are encouraged to learn from any online sources, and from other students, but **do not submit** for assessment anything that is not your work alone.

Do not provide or show your project work to any other student.

Do not store your project work in a public Github or other repository.

We use sophisticated software to find code that is similar to other submissions this year or in past years. Students who submit another person's work as their own or provide their work for another student to submit in whole or in part will be subject to disciplinary action.

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 it)
- Email (if not of interest to everyone)
- Attend my consultation hours (see LMS for schedule)
- Email to schedule an appointment

Subject announcements will be made on the LMS.

Please monitor the LMS for announcements, and the discussion forum for detailed information. Read the discussion forum before asking questions; questions that have already been answered will not be answered again.

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 logic programming and Prolog
- Introduction to constraint programming
- Introduction to functional programming and Haskell
- Declarative programming techniques
- Tools for declarative programming, such as debuggers
- Interfacing to imperative language code

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

For logistical reasons, we will begin with Prolog.

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 logic vs functional programming

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

- Commands are executed.
- Commands have an effect, such as to update the computation state, and later code may depend on this update.

Logic programming languages are based on finding values that satisfy a set of *constraints*.

- Constraints may have multiple solutions or none at all.
- Constraints do not have an effect.

Functional languages are based on evaluating *expressions*.

- Expressions are evaluated.
- Expressions do not have an 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.

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.

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.

The University of Melbourne
School of Computing and Information Systems

COMP30020 / COMP90048
Declarative Programming

Section 1

Introduction to Logic Programming

Logic programming

Imperative programming languages are based on the machine architecture of John von Neumann, which executes a set of instructions step by step.

Functional programming languages are based on the lambda calculus of Alonzo Church, in which *functions* map inputs to outputs.

Logic programming languages are based on the predicate calculus of Gottlob Frege and the concept of a *relation*, which captures a 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.

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.

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 lower case 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.

?-
```

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 .

?- parent(prince_william, prince_charles).
false.
```

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.
```

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 ;
⋮
```

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 :- 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.

?- a = b.
false.

?- X = 7, X = a.
false.

?- 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 "\=". So `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.

?- 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.


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

Datalog Queries

What is the capital of France?

```
?- capital(france, Capital).
Capital = paris.
```

What are capitals of European countries?

```
?- continent(Country, europe), capital(Country, Capital).
Country = france,
Capital = paris.
```

What European countries have populations > 50,000,000?

```
?- continent(Country, europe), population(Country, Pop),
|  Pop > 50_000_000.
Country = france,
Pop = 65700000.
```

The University of Melbourne
School of Computing and Information Systems

COMP30020 / COMP90048
Declarative Programming

Section 2

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.

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.

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]`.

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.

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 $\{\texttt{X1} \mapsto \texttt{leaf}, \texttt{X2} \mapsto \texttt{1}, \texttt{X3} \mapsto \texttt{leaf}\}$ 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.

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 $\{\mathtt{X} \mapsto \mathtt{a}\}$ 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 $\{\mathtt{X} \mapsto \mathtt{a}, \mathtt{Y} \mapsto \mathtt{b}\}$ 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([]).                 append([], C, C).
proper_list([Head|Tail]) :-      append([A|B], C, [A|BC]) :-
        proper_list(Tail).               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/2` built-in predicate relates a list to its length:

```
?- length([a,b,c], Len).
Len = 3.

?- length(List, 3).
List = [_2956, _2962, _2968].
```


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

`[_2956, _2962, _2968]` 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 a predicate to `take` the front *n* elements of a list 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: think about *checking* if the result is correct rather than *computing* it. That is, *think of what* instead of *how*.

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).

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/2` (an infix operator) evaluates expressions.

```
?- X = 6 * 7.
X = 6*7.

?- 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/2 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)

The University of Melbourne
School of Computing and Information Systems

COMP30020 / COMP90048
Declarative Programming

Section 3

Logic and Resolution

Interpretations

In the mind of the person writing a logic program,

- each constant (atomic term) stands for an entity in the "*domain of discourse*" (world of the program);
- each functor (function symbol of arity n where $n > 0$) stands for a function from n entities to one entity in the domain of discourse; and
- each predicate of arity n stands for a particular relationship between n entities in the domain of discourse.

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.

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 a grandparent of C if there is a term B such that A is a parent of B and B is a parent of C.

In mathematical notation:

$$\forall A \forall C : grandparent(A, C) \leftarrow \exists B : parent(A, B) \wedge parent(B, C)$$

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.

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:

$$\forall A \forall B : parent(A, B) \leftarrow \\ \quad (A = queen\_elizabeth \wedge B = prince\_charles)$$

To represent the meaning of the predicate, create a disjunction of the bodies of all the clauses:

$$\begin{aligned} &\forall A \forall B : parent(A, B) \leftarrow \\ &\quad (A = queen\_elizabeth \wedge B = prince\_charles) \;\vee \\ &\quad (A = prince\_philip \wedge B = prince\_charles) \;\vee \\ &\quad (A = prince\_charles \wedge B = prince\_william) \;\vee \\ &\quad (A = prince\_charles \wedge B = prince\_harry) \;\vee \\ &\quad (A = princess\_diana \wedge B = prince\_william) \;\vee \\ &\quad (A = princess\_diana \wedge B = prince\_harry) \end{aligned}$$

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*):

$$
\begin{aligned}
&\forall A \forall B : parent(A, B) \leftrightarrow \\
&\quad (A = queen\_elizabeth \land B = prince\_charles) \ \lor \\
&\quad (A = prince\_philip \land B = prince\_charles) \ \lor \\
&\quad (A = prince\_charles \land B = prince\_william) \ \lor \\
&\quad (A = prince\_charles \land B = prince\_harry) \ \lor \\
&\quad (A = princess\_diana \land B = prince\_william) \ \lor \\
&\quad (A = princess\_diana \land B = prince\_harry)
\end{aligned}
$$

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.

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 $a + b = c$.

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 $T_P(T_P(T_P(\cdots(\emptyset)\cdots)))$ (T_P applied infinitely many times to the empty set).

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:


```
grandparent(queen_elizabeth, prince_harry).

parent(queen_elizabeth, Y), parent(Y, prince_harry).

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


success *failure*

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.


```
grandparent(queen_elizabeth, prince_harry).
                    |
parent(queen_elizabeth, Y), parent(Y, prince_harry).
                    |
parent(prince_charles, prince_harry).
                    |
                 success
```

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.

The University of Melbourne
School of Computing and Information Systems

COMP30020 / COMP90048
Declarative Programming

Section 4

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.

List Reverse

The *mode* of a Prolog goal says which arguments are bound (inputs) and which are unbound (outputs) when the predicate is called.

`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].

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

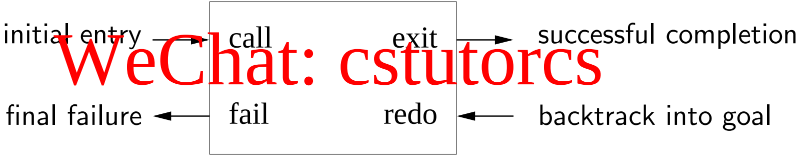

Prolog hangs at this point. We will use the Prolog debugger to understand why. For now, hit control-C and then 'a' to abort.

The Prolog Debugger

To understand the debugger, you will need to understand the Byrd box model. Think of goal execution as a box with a 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:



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.


```
   Call: (7) rev1([a, b], _12717) ? creep
   Call: (8) rev1([b], _12834) ? creep
   Call: (9) rev1([], _12834) ? creep
   Exit: (9) rev1([], []) ? creep
   Call: (9) lists:append([], [b], _12838) ? creep
   Exit: (9) lists:append([], [b], [b]) ? creep
   Exit: (8) rev1([b], [b]) ? creep
   Call: (8) lists:append([b], [a], _12717) ? 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.


```
   Call: (7) rev1(_11553, [a]) ? creep
   Call: (8) rev1(_11661, _11671) ? creep
   Exit: (8) rev1([], []) ? creep
   Call: (8) lists:append([], [_11660], [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(_11661, _11671) ? creep
Call: (9) rev1(_11664, _11674) ? creep
Exit: (9) rev1([], []) ? creep
Call: (9) lists:append([], [_11663], _11678) ? creep
Exit: (9) lists:append([], [_11663], [_11663]) ? creep
Exit: (8) rev1([_11663], [_11663]) ? creep
Call: (8) lists:append([_11663], [_11660], [a]) ? creep
Fail: (8) lists:append([_11663], [_11660], [a]) ? creep
Redo: (9) rev1(_11664, _11674) ? creep
Call: (10) rev1(_11667, _11677) ? creep
Exit: (10) rev1([], []) ? creep
Call: (10) lists:append([], [_11666], _11681) ? creep
Exit: (10) lists:append([], [_11666], [_11666]) ? creep
Exit: (9) rev1([_11666], [_11666]) ? creep
Call: (9) lists:append([_11666], [_11663], _11684) ? creep
Exit: (9) lists:append([_11666], [_11663], [_11666, _11663]) ? creep
Exit: (8) rev1([_11663, _11666], [_11666, _11663]) ? creep
Call: (8) lists:append([_11666, _11663], [_11660], [a])
Fail: (8) lists:append([_11666, _11663], [_11660], [a])
.
.
.
```

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 $\{\texttt{BC} \mapsto \texttt{[]}, \texttt{CB} \mapsto \texttt{[]}\}$, $\{\texttt{BC} \mapsto \texttt{[Z]}, \texttt{CB} \mapsto \texttt{[Z]}\}$, $\{\texttt{BC} \mapsto \texttt{[Y,Z]}, \texttt{CB} \mapsto \texttt{[Z,Y]}\}, \ldots$. For each of these solutions, we call `append(CB, [A], [a])`.

`append([], [A], [a])` succeeds, with $\{\texttt{A} \mapsto \texttt{[a]}\}$. 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.

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

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 `same_length/2` succeeds, both arguments are bound to lists of the same fixed length.


```
rev3(ABC, CBA) :-
        same_length(ABC, CBA),
        rev1(ABC, CBA).
```



```
same_length([], []).
same_length([_|Xs], [_|Ys]) :-
        same_length(Xs, Ys).
```



```
?- rev3(X, [a,b]).
X = [b, a].

?- rev3([a,b], Y).
Y = [b, a].
```

More on the Debugger

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 Debugger

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.

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 `-`.

Documenting Prolog Code

Your code files should have two levels of documentation – file level documentation and predicate level comments. Each file should start with comments that outline: the purpose of the file; its author; the date at which the code was written; and a brief summary of what the code does and any underlying rationale.

Comments should be provided above all significant and non-trivial predicates in a consistent format. These comments should identify: the meaning of each argument; what the predicate does; and the modes in which the predicate is designed to operate.

An excellent resource on coding standards in Prolog is the paper "Coding guidelines for Prolog" by Covington et al. (2011).

Predicate level documentation

The following is an example of predicate level documentation from Covington et al. (2011). This predicate removes duplicates from a list.


```
%% remove_duplicates(+List, -Result)
%
% Removes the duplicates in List, giving Result.
% Elements are considered to match if they can be
% unified with each other; thus, a partly
% uninstantiated element may become further
% instantiated during testing.  If several elements
% match, the last of them is preserved.
```


Predicate arguments are prefaced with a: `+` to indicate that the argument is an `input` and must be instantiated to a term that is not an unbound variable; `-` if the argument is an `output` and may be an unbound variable; or a `?` to indicate that the argument can be either an input or an output.

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.
```

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?

Correctness

The second clause promises that for all n, $n! = n \times (n-1)!$. This is wrong for $n < 1$.

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.
```

Choicepoints

This definition is correct, but it could be more efficient.

When a clause succeeds but there are later clauses that could possibly succeed, Prolog will leave a choicepoint so it can later backtrack and try the later clause.

In this case, 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.

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 (discussed next time) 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].

?- ap(L, [d,e], [a,b,c,d,e]).
false.

?- 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.

The University of Melbourne
School of Computing and Information Systems

COMP30020 / COMP90048
Declarative Programming

Section 5

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).

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.

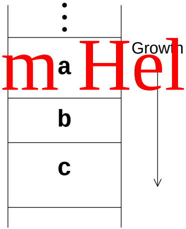
Note that TRO is more often directly applicable in Prolog than other languages because more Prolog code is tail recursive. For example, while `append/3` in Prolog is tail recursive, `++` in Haskell is not, because the last operation performed is `(:)`, not `++`.

```
[] ++ lst = lst
(h:t) ++ lst = h:(t++lst)
```

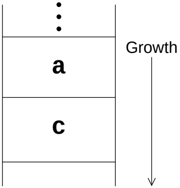
However another optimisation can permit TRO for this code.

The stack

To understand TRO, it is important to understand how programming languages (not just Prolog) implement call and return using a stack. 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 to the right.

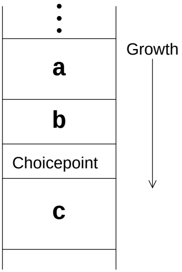


But if all b will do after calling c is return to a, then there is no need to preserve its local variables. Prolog can release b's frame before calling c, as shown to the right. When c is finished, it will return directly to a. This is called *last call optimisation*, and can save significant stack space.



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.


```
fact(N, F) :-
        ( N =:= 0 ->
            F = 1
        ;   N > 0,
            N1 is N - 1,
            fact(N1, F1),
            F is F1 * N
        ).
```

```
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,
            fact1(N1, A1, F)
        ).
```

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;
```

```
fact(N, F) :- fact1(N, 1, F).
fact1(N, A, F) :-
        ( N > 0 ->
            A1 is A * N,
            N1 is N - 1,
            fact1(N1, A1, F)
        ; N =:= 0 ->
            F = A
        ).
```

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
```

```
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.
```

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.
```

```
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
        ).
```

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
    ).
```

```
fact1(N, A, F) :-
    ( N =:= 0 ->
        F = A
    ; N > 0,
        N1 is N - 1,
        fact(N1, F1),
        F is (F1 * N) * A
    ).
```

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
        ).
```

```
fact1(N, A, F) :-
        ( N =:= 0 ->
            F = A
        ;   N > 0,
            N1 is N - 1,
            fact(N1, F1),
            F is F1 * (N * A)
        ).
```

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)
        ).
```

```
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
        ).
```

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 {`N` ↦ `N1`, `F2` ↦ `F1`, `A` ↦ `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
        ).
```

```
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.


```
flatten(empty, List, List).
flatten(node(L,E,R), List, List0) :-
        flatten(L, List, List1),
        List1 = [E|List2],
        flatten(R, List2, List0).
```

The University of Melbourne
School of Computing and Information Systems

COMP30020 / COMP90048
Declarative Programming

Section 6

Higher Order Programming and Impurity

Homoiconicity

Prolog is a *Homoiconic* language. This means that Prolog programs can manipulate Prolog programs as data.

The built-in predicate `clause(+Head,-Body)` allows a running program to access the clauses of the program.


```
?- clause(append(X,Y,Z), Body).
false.
```


Many SWI Prolog “built-ins”, such as `append/3`, are not actually built-in, but are auto-loaded. The first time you use them, Prolog detects that they are undefined, discovers that they can be auto-loaded, quietly loads the, and continues the computation.

Because `append/3` hasn't been auto-loaded yet, `clause/2` can't find its code.

clause/2

If we call append/3 ourselves, Prolog will load it, so we can access its definition.


```
?- append([a],[b],L).
L = [a, b].

?- clause(append(X,Y,Z), Body).
X = [],
Y = Z,
Body = true ;
X = [_7184|_7186],
Z = [_7184|_7192],
Body = append(_7186, Y, _7192).
```

A Prolog interpreter

This makes it very easy to write a Prolog interpreter:


```
interp(Goal) :-
        (   Goal = true
        ->  true
        ;   Goal = (G1,G2)
        ->  interp(G1),
            interp(G2)
        ;   clause(Goal,Body),
            interp(Body)
        ).
```


There is a more complete definition, supporting disjunction and negation, in `interp.pl` in the `examples` directory.

Higher Order Programming

The `call/1` built-in predicate executes a term as a goal, capitalising on Prolog's homoiconicity.


```
?- X=append(A, B, [1]), call(X).
X = append([], [1], [1]),
A = [],
B = [1] ;
X = append([1], [], [1]),
A = [1],
B = [] ;
false.
```


This allows you to write a predicate that takes a goal as an argument and call that goal.

This is called *higher order programming*. We will discuss it in some depth when we cover Haskell.

Currying

It is often useful to provide a goal omitting some arguments, which are supplied when the goal is called. This allows the same goal to be used many times with different arguments.

To support this, many Prologs, including SWI, support versions of `call` of higher arity. All arguments to `call/n` after the goal (first) argument are added as extra arguments at the end of the goal.


```
?- X=append([1,2],[3]), call(X, L).
X = append([1, 2], [3]),
L = [1, 2, 3].
```

When some arguments are supplied with the goal, as we have done here, they are said to be “curried”. We will cover this in greater depth later.

Writing higher-order code

It is fairly straightforward to write higher order predicates using `call/n`. For example, this predicate will apply a predicate to corresponding elements of two lists.


```
maplist(_, [], []).
maplist(P, [X|Xs], [Y|Ys]) :-
        call(P, X, Y),
        maplist(P, Xs, Ys).
```



```
?- maplist(length, [[a,b],[a],[a,b,c]], Lens).
Lens = [2, 1, 3].

?- length(List,2), maplist(same_length(List), List).
List = [[_9378, _9384], [_9390, _9396]].
```


This is defined in the SWI Prolog library. There are versions with arities 2–5, allowing 1–4 extra arguments to be passed to the goal argument.

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 `Template`.


```
?- 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.

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], _1274>_1276 wibble z).
?- write(p(x,[1,2],X>Y wibble z)).
p(x,[1,2],_1464>_1466 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)
    ).
```

The University of Melbourne
School of Computing and Information Systems

COMP30020 / COMP90048
Declarative Programming

Section 7

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 laziness, parallel execution, and various optimizations.

A logic program is in some ways more declarative, as it specifies 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.

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).

Herbrand Constraints

Herbrand constraints are just equality constraints over Herbrand terms — exactly what we have been using since we started with Prolog.

In Prolog, we can constrain variables to be equal, and Prolog will succeed if that is possible, and fail if not.


```
?- length(Word, 5), reverse(Word, Word).
Word = [_2056, _2062, _2068, _2062, _2056].

?- length(Word, 5), reverse(Word, Word), Word=[r,a,d,a,r].
Word = [r, a, d, a, r].

?- length(Word, 5), reverse(Word, Word), Word=[l,a,s,e,r].
false.
```

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 to the *labelling* step, which

- picks a not-yet-fixed variable,
- partitions its current domain (of size n) into k parts $d_1, \ldots, d_k$, 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.

Prolog Arithmetic Revisted

In earlier lectures we introduced a number of built-in arithmetic predicates, including `(is)/2`, `(=:=)/2`, `(=\=)/2` and `(=<)/2`. Recall that these predicates could only be used in certain modes. The predicate `(is)/2`, for example, only works when its second argument is ground.


```
?- 25 is X * X.
ERROR: Arguments are not sufficiently instantiated

?- X is 5 * 5.
X = 25.
```


The CLP(FD) library (constraint logic programming over finite domains) provides replacements for these lower-level arithmetic predicates. These new predicates are called *arithmetic constraints* and can be used in both directions (i.e., `in,out` and `out,in`).

CLP(FD) Arithmetic Constraints

CLP(FD) provides the following arithmetic constraints:

Expr_1 `#=` Expr_2	Expr_1 equals Expr_2
Expr_1 `#\=` Expr_2	Expr_1 is not equal to Expr_2
Expr_1 `#>` Expr_2	Expr_1 is greater than Expr_2
Expr_1 `#<` Expr_2	Expr_1 is less than Expr_2
Expr_1 `#>=` Expr_2	Expr_1 is greater than or equal to Expr_2
Expr_1 `#=<` Expr_2	Expr_1 is less than or equal to Expr_2
Var `in` Low..High	Low $\leq$ Var $\leq$ High
List `ins` Low..High	every Var in List is between Low and High


```
?- 25 #= X * X.
X in -5\/5.

?- 25 #= X * 5.
X = 5.
```

Propagation and Labelling with CLP(FD)

Recall that the domain of a CLP(FD) variable is the set of all integers. We reduce or restrict the domain of these variables with the use of CLP(FD) constraints. When a constraint is posted, the library automatically revises the domains of relevant variables if necessary. This is called propagation.

As we saw in the Sudoku example, sometimes propagation alone is enough to reduce the domains of each variable to a single element. In other cases, we need to tell Prolog to perform the labelling step.

`label/1` is an enumeration predicate that searches for an assignment to each variable in a list that satisfies all posted constraints.


```
?- 25 #= X * X, label([X]).
X = -5;
X = 5.
```

Propagation and Labelling with CLP(FD)

What we expressed in Prolog using generate-and-test


```
?- between(1,9,X), 0 =:= X mod 2, X =:= X * X mod 10.
```

would be expressed with constrain and generate as:


```
?- X in 1..9, 0 #= X mod 2, X #= X * X mod 10.
X in 2..8,
_12562 mod 10#=X,
X^2#=_12562,
X mod 2#=0,
_12562 in 4..64.
```


And with labelling:

```
?- X in 1..9, 0 #= X mod 2, X #= X * X mod 10, label([X]).
X = 6.
```

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 $\mathrm{all\_different}([r1c1, r1c2, r1c3, r2c1, r2c2, r2c3, r3c1, r3c2, r3c3])$.

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]]
```

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, and so on.

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.

The University of Melbourne
School of Computing and Information Systems

COMP30020 / COMP90048
Declarative Programming

Section 8

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.

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.

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`.

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.

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*.

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
```

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
```

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

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.

Order of evaluation: efficiency

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

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
```

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
```

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.

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.

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.

The University of Melbourne
School of Computing and Information Systems

COMP30020 / COMP90048
Declarative Programming

Section 9

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.

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 `LinkedList<Boolean>`, `LinkedList<Integer>`, and `LinkedList<LinkedList<Integer>>` 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.

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.

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 :: [a] -> Bool
```


A function type lists the types of all the arguments and the result, all separated by arrows. We'll see what the a means a bit later.

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

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.

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 p => p
Prelude> :t [1, 2]
[1, 2] :: Num a => [a]
```

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

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

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

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’
  ...
```

if-then-else

```haskell
-- 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.

Guards

```haskell
-- 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?

Structured definitions

Some Haskell equations 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.

Parametric polymorphism

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


```
len :: [t] -> Int
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.

The University of Melbourne
School of Computing and Information Systems

COMP30020 / COMP90048
Declarative Programming

Section 10

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.

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.

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

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.

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).

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"
...
```

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
```

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:

```haskell
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.

Maybe

In languages like C, if you have a value of type `*T` for some type T, or in languages like Java, if you have a value of some non-primitive type, can this value 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 it can be null.

And even if the value **must** not be null, there's no guarantee it won't be. This can lead to segfaults or `NullPointerException`s.

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`. This is a *polymorphic* type, like `[t]`.

The University of Melbourne
School of Computing and Information Systems

COMP30020 / COMP90048
Declarative Programming

Section 11

Using Haskell Types

Representing expressions in C

```
typedef enum {
    EXPR_NUM, EXPR_VAR, EXPR_BINOP, EXPR_UNOP
} ExprKind;

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 */
};
```

Representing expressions in Java

```
public abstract class Expr {
    ... abstract methods ...
}

public class NumExpr extends Expr {
    int value;
    ... implementation of abstract methods ...
}

public class VarExpr extends Expr {
    String name;
    ... implementation of abstract methods ...
}
```

Representing expressions in Java (2)


```
public class BinExpr extends Expr {
    Binop binop;
    Expr arg1;
    Expr arg2;
    ... implementation of abstract methods ...
}
```



```
public class UnExpr extends Expr {
    Unop unop;
    Expr arg;
    ... implementation of abstract methods ...
}
```

Representing expressions in Haskell

```
data Expr
    = Number Int
    | Variable String
    | Binop Binopr Expr Expr
    | Unop Unopr Expr

data Binopr = Plus | Minus | Times | Divide
data Unopr  = 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.

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.

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 represention: 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.

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.

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

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

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

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;

        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;
        }
    }

    return FALSE;
}
```

Searching a BST in Haskell

```
search_bst :: Tree -> String -> Maybe Int
search_bst Leaf _ = Nothing
search_bst (Node k v l r) sk
  | sk == k   = Just v
  | sk <  k   = search_bst l sk
  | otherwise = 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`.
- We could have used Haskell's if-then-else for this, but guards make the code look much nicer and easier to read.

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.

The University of Melbourne
School of Computing and Information Systems

COMP30020 / COMP90048
Declarative Programming

Section 12

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

loop2func base case
loop2func recursive case

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`.

Example: C

```
int f(int *a, int size)
{
    int i;
    int target;
    int first_gt_target;

    i = 0;
    while (i < size && a[i] <= 0) {
        i++;
    }

    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

```
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

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)

find_gt :: [Int] -> Int -> Int
find_gt (x:xs) target =
    if x <= target then find_gt xs target else x
```

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. Any loop can be implemented with recursion, but some recursions are difficult to implement with iteration.

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?

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`, Haskell 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.

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.

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.

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.

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).

Sublists

Suppose we want to write a Haskell function


```
sublists :: [a] -> [[a]]
```

that returns a list of all the "sublists" of a list. A list *a* is a sublist of a list *b* iff every element of *a* appears in *b* in the same order, though some elements of *b* may be omitted from *a*. It does not matter in what order the sublists appear in the resulting list.

For example:


```
sublists "ABC" = ["ABC","AB","AC","A","BC","B","C",""]
sublists "BC" =  ["BC","B","C",""]
```

How would you implement this?

Declarative thinking

Some problems are difficult to approach imperatively, and are much easier to think about declaratively.

The imperative approach is procedural: we devise a way to solve the problem step by step. As an afterthought we may think about grouping the steps into chunks (methods, procedures, functions, etc.)

The declarative approach breaks down the problem into chunks (functions), assembling the results of the chunks to construct the result.

You must be careful in imperative languages, because the chunks may not compose due to side-effects. The chunks always compose in purely declarative languages.

Recursive thinking

One especially useful approach is recursive thinking: use the function you are defining as one of the chunks.

To take this approach:

1. Determine how to produce the result for the whole problem from the results for the parts of the problem (recursive case);
2. Determine the solution for the smallest part of the input (base case).

Keep in mind the specification of the problem, but it also helps to think of concrete examples.

For lists, (1) usually means generating the result for the whole list from the list head and the result for the tail; (2) usually means the result for the empty list.

This works perfectly well in most imperative languages, if you're careful to ensure your function composes. But it takes practice to think this way.

Sublists again

Write a Haskell function:

```
sublists :: [a] -> [[a]]
sublists "ABC" = ["ABC","AB","AC","A","BC","B","C",""]
sublists "BC" =  ["BC","B","C",""]
```


How can we produce `["ABC","AB","AC","A","BC","B","C",""]` from `["BC","B","C",""]` and `A`?

It is just `["BC","B","C",""]` with `A` added to the front of each string, followed by `["BC","B","C",""]` itself.

For the base case, the only sublist of `[]` is `[]` itself, so the list of sublists of `[]` is `[[]]`.

Sublists again

The problem becomes quite simple when we think about it declaratively (recursively). The sublists of a list is the sublists of its tail both with and without the head of the list added to the front of each sublist.


```
sublists :: [a] -> [[a]]
sublists [] = [[]]
sublists (e:es) = addToEach e restSeqs ++ restSeqs
    where restSeqs = sublists es
```



```
addToEach :: a -> [[a]] -> [[a]]
addToEach h [] = []
addToEach h (t:ts) = (h:t):addToEach h ts
```

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

- because both old and new are needed, as in `sublists`
- to implement undo
- to gather statistics, e.g. about how the size of a data structure changes over time

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
    | ik == k   = Node ik iv l r
    | ik < k    = Node k v (insert_bst l ik iv) r
    | otherwise = 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.

The University of Melbourne
School of Computing and Information Systems

COMP30020 / COMP90048
Declarative Programming

Section 13

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.

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.

Data.Map

The polymorphic Tree type described above is defined in the standard library with the name Map in the module Data.Map. You can import it with the declaration:


```
import Data.Map as Map
```

The key functions defined for Maps include:


```
    insert :: Ord k => k -> a -> Map k a -> Map k a
Map.lookup :: Ord k => k -> Map k a -> Maybe a
    (!)    :: Ord k => Map k a -> k -> a   -- infix operator
    size   ::          Map k a -> Int
```


... and many, many more functions; see the documentation.

Deriving membership automatically

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


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.

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.

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.

Defining multiple names

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

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

or

```
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.

The University of Melbourne
School of Computing and Information Systems

COMP30020 / COMP90048
Declarative Programming

Section 14

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;

    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;
        }
    }
}
```

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

is_pos :: Int -> Bool
is_pos x = if x > 0 then True else False

is_long :: String -> Bool
is_long x = if length x > 3 then True else False
```

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 $\lambda a.\lambda b.a + b$.

Map

(Not to be confused with Data.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


... 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.

Calling a closure: an example

```
filter f (x:xs) =
    if f x then x:fxs else fxs
    where fxs = filter f xs
```



```
... 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)
```

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 "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)`.

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:


```
minimum = head . sort
```

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


```
maximum = 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.

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.

The University of Melbourne
School of Computing and Information Systems

COMP30020 / COMP90048
Declarative Programming

Section 15

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.

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	$((((I \odot X_1) \odot X_2) \ldots) \odot X_n)$
right	$(X_1 \odot (X_2 \odot (\ldots (X_n \odot I))))$
balanced	$((X_1 \odot X_2) \odot (X_3 \odot X_4)) \odot \ldots$

Here $\odot$ denotes a binary function, 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

suml :: Num a => [a] -> a
suml = foldl (+) 0

productl :: Num a => [a] -> a
productl = foldl (*) 1

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

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.

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 (len `div` 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.

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

$$\textbf{foldl1} \;\; o \; [X_1, X_2, \ldots, X_n] = ((X_1 \odot X_2) \ldots \odot X_n)$$
$$\textbf{foldr1} \;\; o \; [X_1, X_2, \ldots, X_n] = (X_1 \odot (X_2 \odot \ldots X_n))$$

```
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
```



```
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:

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

```
snoc :: [a] -> a -> [a]
snoc tl hd = hd:tl

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

reverse = foldl (flip (:)) []
```

Foldable

But what about types other than lists? Can we fold over them?

Actually, we can:


```
Prelude> sum (Just 7)
7
Prelude> sum Nothing
0
```


In fact we can fold over any type in the type class `Foldable`


```
Prelude> :t foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

We can declare our own types to be instances of `Foldable` by defining `foldr` for our type; then many standard functions, such `length`, `sum`, *etc.* will work on that type, too.

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

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]

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

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
```

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

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

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

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

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.

The University of Melbourne
School of Computing and Information Systems

COMP30020 / COMP90048
Declarative Programming

Section 16

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.

Representation of if-then-elses

To represent if-then-else expressions such as C's ternary operator


```
(x > y) ? x : y
```

the relevant union field is a structure that has an array of operands, which should have exactly three elements (the condition, the then part and the else part). 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.

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

data Binop = Add | Sub | ...
data Unop = Uminus | ...
data Const = IntConst Int | FloatConst Double | ...
```

Generic lists in C

```
typedef struct generic_list List;
struct generic_list {
    void    *head;
    List    *tail;
};

...
List    *int_list;
List    *p;
int     item;

for (p = int_list; p != NULL; p = p->tail) {
    item = (int) p->head;
    ... do something with item ...
}
```

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.

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

meters_to_length :: Double -> Length
meters_to_length m = Meters m

feet_to_length :: Double -> Length
feet_to_length f = Meters (f * 0.3048)

add_lengths :: Length -> Length -> Length
add_lengths (Meters a) (Meters b) = Meters (a+b)
```

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

add_durations :: Duration -> Duration -> Duration
add_durations (Seconds a) (Seconds b) = Seconds (a+b)

add_duration_to_time :: Time -> Duration -> Time
add_duration_to_time (SecondsSinceEpoch sse) (Seconds t) =
    SecondsSinceEpoch (sse + t)
```

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.

Mapping over a `Maybe`

Suppose we have a type

```
type Marks = Map String [Int]
```


giving a list of all the marks for each student.

(A `type` declaration like this declares that `Marks` is an alias for `Map String [Int]`, just the way `String` is an alias for `[Char]`.)

We want to write a function

```
studentTotal :: Marks -> String -> Maybe Int
```


that returns `Just` the total mark for the specified student, or `Nothing` if the specified student is unknown. `Nothing` means something different from `Just 0`.

Mapping over a Maybe

This definition will work, but it's a bit verbose:


```
studentTotal marks student =
    case Map.lookup student marks of
        Nothing -> Nothing
        Just ms -> Just (sum ms)
```


We'd like to do something like this:

```
studentTotal marks student =
    sum (Map.lookup student marks)
```


but it's a type error:

```
Couldn't match expected type 'Maybe Int'
with actual type 'Int'
```

The `Functor` class

If we think of a `Maybe a` as a "box" that holds an `a` (or maybe not), we want to apply a function *inside* the box, leaving the box in place but replacing its content with the result of the function.

That's actually what the `map` function does: it applies a function to the contents of a list, returning a list of the results.

What we want is to apply a function `a -> b` to the contents of a `Maybe a`, returning a `Maybe b`. We want to `map` over a `Maybe`.

The `Functor` type class is the class of all types that can be "mapped" over. This includes `[a]`, but also `Maybe a`.

The Functor class

You can map over a Functor type with the fmap function:


```
fmap :: Functor f => (a -> b) -> f a -> f b
```

This gives us a much more succinct definition:


```
studentTotal marks student =
    fmap sum (Map.lookup student marks)
```

Functor is defined in the standard prelude, so you can use fmap over Maybes (and lists) without importing anything.

The standard prelude also defines a function <$> as an infix operator alias for fmap, so the following code will also work:

```
sum <$> Map.lookup student marks
```

Beyond Functors

Suppose our students work in pairs, with either teammate submitting the pair's work. We want to write a function


```
pairTotal :: Marks -> String -> String -> Maybe Int
```

to return the total of the assessments of two students, or `Nothing` if either or both of the students are not enrolled.

This code works, but is disappointingly verbose:

```
pairTotal marks student1 student2 =
    case studentTotal marks student1 of
        Nothing -> Nothing
        Just t1 -> case studentTotal marks student2 of
                        Nothing -> Nothing
                        Just t2 -> Just (t1 + t2)
```

Putting functions in `Functors`

`Functor` works nicely for unary functions, but not for greater arities. If we try to use `fmap` on a binary function and a `Maybe`, we wind up with a function in a `Maybe`.

Remembering the type of `fmap`,

```
fmap :: Functor f => (a -> b) -> f a -> f b
```


if we take `f` to be `Maybe`, `a` to be `Int` and `b` to be `Int -> Int`, then we see that

```
fmap (+) (studentTotal marks student1)
```


returns a value of type `Maybe (Int -> Int)`.

So all we need is a way to extract a function from inside a functor and `fmap` that over another functor. We want to apply one functor to another.

Applicative functors

Enter applicative functors. These are functors that can contain functions, which can be applied to other functors, defined by the Applicative class.

The most important function of the Applicative class is

```
(<*>) :: f (a -> b) -> f a -> f b
```

which does exactly what we need.

Happily, Maybe is in the Applicative class, so the following definition works:


```
pairTotal marks student1 student2 =
    let mark = studentTotal marks
    in  (+) <$> mark student1 <*> mark student2
```

Applicative

The second function defined for every Applicative type is


```
pure :: a -> f a
```

which just inserts a value into the applicative functor. For the Maybe class, pure = Just. For lists, pure = (:[]) (creating a singleton list).

For example, if we wanted to subtract a student's score from 100, we could do:

```
(-) <$> pure 100 <*> studentTotal marks student
```


but a simpler way to do the same thing is:

```
(100-) <$> studentTotal marks student
```

In fact, every Applicative must also be a Functor, just as every Ord type must be Eq.

Lists are Applicative

<*> gets even more interesting for lists:


```
...> (++) <$> ["do","something","good"] <*> pure "!"
["do!","something!","good!"]
...> (++) <$> ["a","b","c"] <*> ["1","2"]
["a1","a2","b1","b2","c1","c2"]
```


You can think of <*> as being like a Cartesian product, hence the "*".

The University of Melbourne
School of Computing and Information Systems

COMP30020 / COMP90048
Declarative Programming

Section 17

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", "elephant", "legion"]
True
```


To check if a word contains any vowels:

```
Prelude> any (\x -> elem x "aeiou") "sky"
False
```

`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.

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`.

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.

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

return x = Just x

(Just x) >>= f = f x
Nothing >>= _ = Nothing
```

```
-- monad ops for MaybeOK
data MaybeOK t = OK t
               | Error String

return x         = OK x

(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
```



```
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?

Simplifying code with monads


```
maybe_head :: [a] -> MaybeOK a
maybe_sqrt :: Int -> MaybeOK Double
maybe_sqrt_of_head :: [Int] -> MaybeOK Double

-- definition not using monads
maybe_sqrt_of_head l =
    case maybe_head l of
        Error msg -> Error msg
        OK h      -> maybe_sqrt h

-- simpler definition using monads
maybe_sqrt_of_head l =
    maybe_head l >>= maybe_sqrt
```

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 `()`.

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.

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
  )
```

`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"
```

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
```

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
```

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.

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

readlen :: IO Int
readlen = do
    str <- getLine
    return (length str)
```

The University of Melbourne
School of Computing and Information Systems

COMP30020 / COMP90048
Declarative Programming

Section 18

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

print_table :: Int -> Int -> IO ()
print_table cur max
    | cur > max = return ()
    | otherwise = do
        putStrLn (table_entry cur)
        print_table (cur+1) max

table_entry :: Int -> String
table_entry n = (show n) ++ "^2 = " ++ (show (n*n))
```

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)

table_entry :: Int -> String
table_entry n = (show n) ++ "^2 = " ++ (show (n*n))

show_entry :: Int -> IO ()
show_entry n = do putStrLn (table_entry n)

execute_actions :: [IO ()] -> IO ()
execute_actions [] = return ()
execute_actions (x:xs) = do
    x
    execute_actions xs
```

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 printfs

One standard approach for debugging a program written in C is to edit your code to insert debugging printfs to show you what input your buggy function is called with and what results it computes.

In a program written in Haskell, you can't just insert printing code into functions not already in the IO monad.

Debugging printfs are only used for debugging, so you're not concerned with where the output from debugging printfs appears relative to other output. This is where the function `unsafePerformIO` comes in: it allows you to perform IO anywhere, but the order of output will probably be wrong.

Do not use `unsafePerformIO` in real code, but it is useful for debugging.

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)
```

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 code adds 1 to each element of a tree, and does not need a monad:

```
data Tree a = Empty | Node (Tree a) a (Tree a)
              deriving Show
type IntTree = Tree Int

incTree :: IntTree -> IntTree
incTree Empty = Empty
incTree (Node l e r) =
    Node (incTree l) (e + 1) (incTree r)
```

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)

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)
```

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)

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

withCounter :: Int -> Counter a -> a
withCounter init f = fst (runState f init)

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)

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)
```

The University of Melbourne
School of Computing and Information Systems

COMP30020 / COMP90048 Declarative Programming

Section 19

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.

The sieve of Eratosthenes


```
-- returns the (infinite) list of all primes
all_primes :: [Integer]
all_primes = prime_filter [2..]

prime_filter :: [Integer] -> [Integer]
prime_filter [] = []
prime_filter (x:xs) =
    x:prime_filter (filter (not . (`divisibleBy` x)) xs)

-- n `divisibleBy` d means n is divisible by d
divisibleBy n d = n `mod` d == 0
```

Using `all_primes`

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.

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.

Evaluating lazy values only once

Many functions use the values of some variables more than once. This includes `takeWhile`, which uses `x` twice:


```
takeWhile _ []  = []
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)); ...
```



```
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) ...
```



```
ite :: Bool -> a -> a -> a
ite c t e = if c then t else e
```

In Haskell, (c) will execute a call to only one of `f` and g, and thanks to lazyness, this is also true for (d).

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 :: (Ord a, Num a) => a -> b -> b -> b -> b
ite3 x lt eq gt
    | x < 0  = lt
    | x == 0 = eq
    | x > 0  = gt
```

Using lazyness to avoid unnecessary work

```
minimum = 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 $O(n\ log\, n)$ 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.

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 fs)
    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.

The University of Melbourne
School of Computing and Information Systems

COMP30020 / COMP90048
Declarative Programming

Section 20

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 the last lecture 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.

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.

Memory efficiency

(Revised) BST insertion code:


```
insert_bst :: Ord k => Tree k v -> k -> v -> Tree k v
insert_bst Leaf ik iv = Node ik iv Leaf Leaf
insert_bst (Node k v l r) ik iv
  | k == ik   = Node ik iv l r
  | k > ik    = Node k v (insert_bst l ik iv) r
  | otherwise = Node k v l (insert_bst r ik iv)
```


As discussed earlier, this creates new data structures instead of destructively modifying the old structure.

The advantage of this is that the old structure can still be used.

The disadvantage is new memory is allocated and written. This takes time, and creates garbage that must be collected.

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 $log_2(n)$.

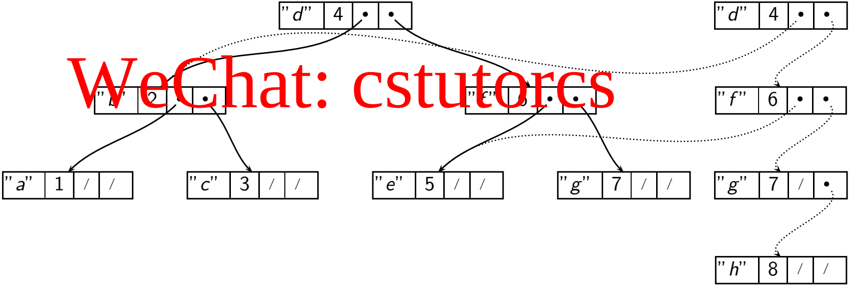
Therefore the number of nodes allocated during an insertion tends to be logarithmic in the size of the tree.

- 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.

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".



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*.

Simple Deforestation

In some cases, you can deforest your own code with minimal effort. For example, you can always deforest two calls to `map`:

```
map (+1) $ map (2*) list
```

is equivalent to

```
map ((+1) . (2*)) list
```

The second one is more succinct, more elegant, and more efficient.

You can combine two calls to filter in a similar way:

```
filter (>=0) $ filter (<10) list
```

is always the same as

```
filter (\x -> x >= 0 & x < 10) list
```

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

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

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

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
```

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)
```



```
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 []

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

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.

The University of Melbourne
School of Computing and Information Systems

COMP30020 / COMP90048
Declarative Programming

Section 21

Interfacing with foreign languages

Foreign language interface

Many applications involve code written in a number of different languages; declarative languages are no different in this respect. There are many reasons for this:

- to interface to existing code (especially libraries) written in another language;
- to write performance-critical code in a lower-level language (typically C or C++);
- to write each part of an application in the most appropriate language;
- as a way to gracefully translate an application from one language to another, by replacing one piece at a time.

Any language that hopes to be successful must be able to work with other languages. This is generally done through what is called a *foreign language interface* or *foreign function interface*.

Application binary interfaces

In computer science, a *platform* is a combination of an instruction set architecture (ISA) and an operating system, such as x86/Windows 10, x86/Linux or SPARC/Solaris.

Each platform typically has an *application binary interface*, or ABI, which dictates such things as where the callers of functions should put the function parameters and where the callee function should put the result.

By compiling different files to the same ABI, functions in one file can call functions in a separately compiled file, even if compiled with different compilers.

The traditional way to interface two languages, such as C and Fortran, or Ada and Java, is for the compilers of both languages to generate code that follows the ABI.

Beyond C

ABIs are typically designed around C's simple calling pattern, where each function is compiled to machine language, and each function call passes some number of inputs, calls another known function, possibly returns one result, and is then finished.

This model does not work for lazy languages like Haskell, languages like Prolog or Mercury that support nondeterminism, languages like Prolog, Python, and Java that are implemented through an abstract machine, or even languages like C++ where function (method) calls may invoke different code each time they are executed.

In such languages, code is not compiled to the normal ABI. Then it becomes necessary to provide a mechanism to call code written in other languages. Typically, calling C code through the normal ABI is supported, but interfacing to other languages may also be supported.

Boolean functions

One application of a foreign interface is to use specialised data structures and algorithms that would be difficult or inefficient to implement in the host language.

Some applicatations need to be able to efficiently manipulate Boolean formulas (Boolean functions). This includes the following primitive values and operations:

- *true*, *false*
- Boolean variables: *eg:* a, b, c, ...
- Operations: and ($\wedge$), or ($\vee$), not ($\neg$), implies ($\rightarrow$), iff ($\leftrightarrow$), *etc.*
- Tests: satisfiability (is there any binding for the variables that makes a formula true?), equivalence (are two formulas the same for every set of variable bindings?)

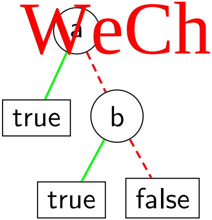
For example, is $a \leftrightarrow b$ equivalent to $\neg((a \wedge \neg b) \vee (\neg a \wedge b))$?

Binary Decision Diagrams

Deciding satisfiability or equivalence of Boolean functions is NP-complete, so we need an efficient implementation.

BDDs are *decision graphs*, based on *if-then-else* (*ite*) nodes, where each node is labeled by a boolean variable and each leaf is a truth value.

With a *truth assignment* for each variable, the value of the formula can be determined by traversing from the root, following *then* branch for true variables and *else* branch for false variables.



a	b	the BDD
T	T	T
T	F	T
F	T	T
F	F	F

BDDs in Haskell

We could represent BDDs in Haskell with this type:


```
data BDD label = BTrue | BFalse
               | Ite label (BDD label) (BDD label)
```

The meaning of a BDD is given by:

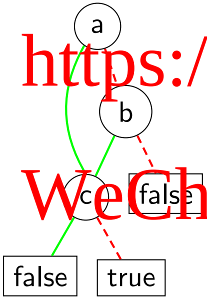
$$
\begin{aligned}
\mathit{meaning}\ \texttt{BTrue} &= \mathit{true} \\
\mathit{meaning}\ \texttt{BFalse} &= \mathit{false} \\
\mathit{meaning}\ (\texttt{Ite}\ v\ t\ e) &= (v \wedge \mathit{meaning}\ t) \vee (\neg v \wedge \mathit{meaning}\ e)
\end{aligned}
$$

So for example, *meaning* (Ite a BTrue (Ite b BTrue BFalse))

$$
\begin{aligned}
&= (\mathtt{a} \wedge \mathit{true}) \vee (\neg \mathtt{a} \wedge (\mathtt{b} \wedge \mathit{true} \vee (\neg \mathtt{b} \wedge \mathit{false}))) \\
&= \mathtt{a} \vee (\neg \mathtt{a} \wedge \mathtt{b} \vee \mathit{false}) \\
&= \mathtt{a} \vee \mathtt{b}
\end{aligned}
$$

ROBDDs

Reduced Ordered Binary Decision Diagrams (*ROBDDs*) are BDDs where labels are in increasing order from root to leaf, no node has two identical children, and no two distinct nodes have the same semantics.



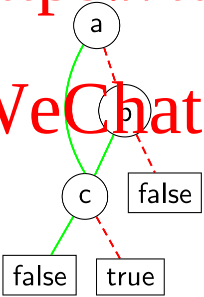
a	**b**	**c**	the BDD
T	T	T	F
T	T	F	T
T	F	T	F
T	F	F	T
F	T	T	F
F	T	F	T
F	F	T	F
F	F	F	F

By sharing the c node, the ROBDD is smaller than it would be if it were a tree. For larger ROBDDs, this can be a big savings.

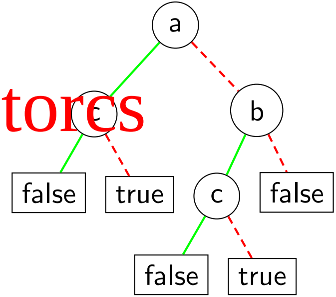
Object Identity

ROBDD algorithms traverse DAGs, and often meet the same subgraphs repeatedly. They greatly benefit from *caching*: recording results of past operations to reuse without repeating the computation.

Caching requires efficiently recognizing when a node is seen again. Haskell does not have the concept of *object identity*, so it cannot distinguish



from



Structural Hashing

Building a new ROBDD node $ite(v, thn, els)$ must ensure that:

1. the two children of every node are different; and
2. for any Boolean formula, there is only one ROBDD node with that semantics.

The first is achieved by checking if $thn = els$, and if so, returning thn

The second is achieved by *structural hashing* (AKA *hash-consing*): maintaining a hash table of past calls $ite(v, thn, els)$ and their results, and always returning the past result when a call is repeated.

Because of point 1 above, satisfiability of an ROBDD can be tested in constant time ($meaning\ r$ is satisfiable iff $r \neq false$)

Because of point 2 above, equality of two ROBDDs is also a constant time test ($meaning\ r_1 = meaning\ r_2$ iff $r_1 = r_2$)

Impure implementation of pure operations

(We haven't cheated NP-completeness: we have only shifted the cost to building ROBDDs.)

Yet ROBDD operations are purely declarative. Constructing ROBDDs, conjunction, disjunction, negation, implication, checking satisfiability and equality, etc., are all pure.

This is an example of using impurity to build purely declarative code.

In fact, all declarative code running on a commodity computer does that: these CPUs work through impurity. Even adding two numbers on a modern CPU works by destructively adding one register to another.

If you are required to work in an imperative or object-oriented language, you can still use such languages to build declarative abstractions, and work with them instead of working directly with impure constructs.

`robdd.h`: C interface for ROBDD implementation

```
extern robdd *true_rep(void);        /* ROBDD true */
extern robdd *false_rep(void);       /* ROBDD false */
extern robdd *variable_rep(int var); /* ROBDD for var */
extern robdd *conjoin(robdd *a, robdd *b);
extern robdd *disjoin(robdd *a, robdd *b);
extern robdd *negate(robdd *a);
extern robdd *implies(robdd *a, robdd *b);

extern int is_true(robdd *f);       /* is ROBDD == true? */
extern int is_false(robdd *f);      /* is ROBDD == false? */

extern int robdd_label(robdd *f);   /* label of f */
extern robdd *robdd_then(robdd *f); /* then branch of f */
extern robdd *robdd_else(robdd *f); /* else branch of f */
```

Interfacing Haskell to C

For simple cases, the Haskell foreign function interface is fairly simple. You can interface to a C function with a declaration of the form:

`foreign import ccall "`C *name*`"` *Haskell name* `::` *Haskell type*

But how shall we represent an ROBDD in Haskell?

C primitive types convert to and from natural Haskell types, eg,

C `int` $\longleftrightarrow$ Haskell `Int`

In Haskell, we want to treat ROBDDs as an *opaque type*: a type we cannot peer inside, we can only pass it to, and receive it as output from, foreign functions.

The Haskell `Word` type represents a word of memory, much like an `Int`. However `Word` is not opaque, as we can confuse it with an integer, or any `Word` type.

newtype

Declaring `type BoolFn = Word` would not make `BoolFn` opaque; it would just be an alias for `Word` and could be passed as a `Word`.

We can make it opaque with a `data BoolFn = BoolFn Word` declaration. We could convert a `Word w` to a `BoolFn` with `BoolFn w`, and convert a `BoolFn b` to a `Word` with


```
let BoolFn w = b in ...
```

But this would *box* the word, adding an extra indirection to operations. Instead we declare:


```
newtype BoolFn = BoolFn Word deriving Eq
```

We can only use `newtype` to declare types with only one constructor, with exactly one argument. This avoids the indirection, makes the type opaque, and allows it to be used in the foreign interface.

The interface

```
foreign import ccall "true_rep"     true     :: BoolFn
foreign import ccall "false_rep"    false    :: BoolFn
foreign import ccall "variable_rep" variable :: Int->BoolFn

foreign import ccall "is_true"      isTrue   :: BoolFn->Bool
foreign import ccall "is_false"     isFalse  :: BoolFn->Bool
foreign import ccall "robdd_label"  minVar   :: BoolFn->Int
foreign import ccall "robdd_then"   minThen  :: BoolFn->BoolFn
foreign import ccall "robdd_else"   minElse  :: BoolFn->BoolFn

type BoolBinOp = BoolFn -> BoolFn -> BoolFn
foreign import ccall "conjoin"      conjoin  :: BoolBinOp
foreign import ccall "disjoin"      disjoin  :: BoolBinOp
foreign import ccall "negate"       negation:: BoolFn->BoolFn
foreign import ccall "implies"      implies  :: BoolBinOp
```

Using it

To make C code available, compile it and pass the object file on the ghc or ghci command line.

(There is also code to show BoolFns in disjunctive normal form; all code is in BoolFn.hs, robdd.c and robdd.h in the examples directory.)


```
nomad% gcc -c -Wall robdd.c
nomad% ghci robdd.o
GHCi, version 8.4.3: http://www.haskell.org/ghc/
Prelude> :l BoolFn.hs
[1 of 1] Compiling BoolFn           ( BoolFn.hs, interpreted )
Ok, one module loaded.
*BoolFn> (variable 1) `disjoin` (variable 2)
((1) | (~1 & 2))
*BoolFn> it `conjoin` (negation $ variable 3)
((1 & ~3) | (~1 & 2 & ~3))
```

Interfacing to Prolog

The Prolog standard does not standardise a foreign language interface. Each Prolog system has its own approach.

The SWI Prolog approach does most of the work on the C side, rather than in Prolog. This is powerful, but inconvenient.

In an SWI Prolog source file, the declaration


```
:- use_foreign_library(swi_robdd).
```

will load a compiled C library file that links the code in `swi_robdd.c`, which forms the interface to Prolog, and the `robdd.c` file.

These are compiled and linked with the shell command:

```
swipl-ld -shared -o swi_robdd swi_robdd.c robdd.c
```

Connecting C code to Prolog

The `swi_robdd.c` file contains C code to interface to Prolog:


```
install_t install_swi_robdd() {
  PL_register_foreign("boolfn_node", 4, pl_bdd_node, 0);
  PL_register_foreign("boolfn_true", 1, pl_bdd_true, 0);
  PL_register_foreign("boolfn_false", 1, pl_bdd_false, 0);
  PL_register_foreign("boolfn_conjoin", 3, pl_bdd_and, 0);
  PL_register_foreign("boolfn_disjoin", 3, pl_bdd_or, 0);
  PL_register_foreign("boolfn_negation", 2, pl_bdd_negate, 0);
  PL_register_foreign("boolfn_implies", 3, pl_bdd_implies, 0);
}
```

This tells Prolog that a call to `boolfn_node`/4 is implemented as a call to the C function `pl_bdd_node`, *etc.*

Marshalling data

A C function that implements a Prolog predicate needs to convert between Prolog terms and C data structures. This is called *marshalling* data.


```
static foreign_t
pl_bdd_and(term_t f, term_t g, term_t result_term) {
  void *f_nd, *g_nd;
  if (PL_is_integer(f)
      && PL_is_integer(g)
      && PL_get_pointer(f, &f_nd)
      && PL_get_pointer(g, &g_nd)) {
    robdd *result = conjoin((robdd *)f_nd, (robdd *)g_nd);
    return PL_unify_pointer(result_term, (void *)result);
  } else {
    PL_fail;
  }
}
```

Making Boolean functions abstract in Prolog

To keep Prolog code from confusing an ROBDD (address) from a number, we wrap the address in a `boolfn/1` term, much like we did in Haskell. We must do this manually; it is most easily done in Prolog code.

```
%  conjoin(+BFn1, +BFn2, -BFn)
%  BFn is the conjunction of BFn1 and BFn2.
conjoin(boolfn(F), boolfn(G), boolfn(FG)) :-
        boolfn_conjoin(F, G, FG).
```

We can make Prolog print BDDs (or anything) nicely by adding a clause for `user:portray/1`:

```
:- multifile(user:portray/1).
user:portray(boolfn(BDD)) :- !,
        % definition is in boolfn.pl ...
```

Using it

```
nomad% swipl-ld -shared -o swi_robdd swi_robdd.c robdd.c
nomad% pl
Welcome to SWI-Prolog (threaded, 64 bits, version 7.7.19)
1 ?- [boolfn].
true.

2 ?- variable(1,A), variable(2,B), variable(3,C),
|    disjoin(A,B,AB), negation(C,NotC), conjoin(AB,NotC,X).
A = ((1)),
B = ((2)),
C = ((3)),
AB = ((1) | (~1 & 2)),
NotC = ((~3)),
X = ((1 & ~3) | (~1 & 2 & ~3)).
```

Impedance mismatch

Declarative languages like Haskell and Prolog typically use different representations for similar data. For example, what would be represented as a list in Haskell or Prolog would most likely be represented as an array in C or Java.

The consequence of this is that in each language (declarative and imperative) it is difficult to write code that works on data structures defined in the other language.

This problem, usually called *impedance mismatch*, is the reason why most cross-language interfaces are low level, and operate only or mostly on values of primitive types.

Comparative strengths of declarative languages

- Programmers can be significantly more productive because they can work at a significantly higher level of abstraction. They can focus on the big picture without getting lost in details, such as whose responsibility it is to free a data structure.
- Processing of symbolic data is significantly easier due to the presence of algebraic data types and parametric polymorphism.
- Programs can be significantly more reliable, because
  - you cannot make a mistake in an aspect of programming that the language automates (e.g. memory allocation), and
  - the compiler can catch many more kinds of errors.
- What debugging is still needed is easier because you can jump backward in time.
- Maintenance is significantly easier, because
  - the type system helps to locate what needs to be changed, and
  - the typeclass system helps avoid unwanted coupling in the first place.
- You can automatically parallelize declarative programs.

Comparative strengths of imperative languages

- If you are willing to put in the programming time, you can make the final program significantly faster.
- Most existing software libraries are written in imperative languages. Using them in declarative languages is harder than using them in another imperative language (due to dissimilarity of basic concepts), while using them is easiest in the language they are written in. If the bulk of a program interfaces to an existing library, this argues for writing the program in the language of the library:
  - Java for Swing
  - Visual Basic.NET or C# for ASP.NET
- There is a much greater variety of programming tools to choose from (debuggers, profilers, IDEs etc).
- It is much easier to find programmers who know or can quickly learn the language.

The University of Melbourne
School of Computing and Information Systems

COMP30020 / COMP90048
Declarative Programming

Section 22

Parsing

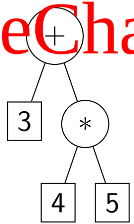
Parsing

A *parser* is a program that extracts a structure from a linear sequence of elements.

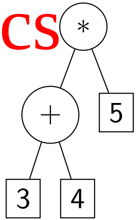
For example, a parser is responsible for taking input like:

$$3 + 4 * 5$$

and producing a data structure representing the intended structure of the input:



as opposed to



Using an existing parser

The simplest parsing technique is to use an existing parser. Since every programming language must have a parser to parse the program, it may also give the programmer access to its parser.

A *Domain Specific Language* (DSL) is a small programming language intended for a narrow domain. Often these are embedded in existing languages, extending the host language with new capabilities, but using the host language for functionality outside that domain.

If a DSL can be parsed by extending the host language parser, that makes the DSL more convenient to use, since it just adds new constructs to the language.

Prolog handles that quite nicely, as we saw earlier. The `read`/1 built-in predicate reads a term. You can use `op`/3 declarations to extend the language.

Operator precedence

Operator precedence parsing is a simple technique based on operator's:

precedence which operators bind tightest;

associativity whether repeated infix operators associate to the left, right, or neither (eg, whether $a - b - c$ is $(a - b) - c$ or $a - (b - c)$ or an error); and

fixity whether an operator is infix, prefix, or postfix.

In Prolog, the `op`/3 predicate declares an operator:

$$:-\ \texttt{op}(\textit{precedence}, \textit{fixity}, \textit{operator})$$

where *precedence* is a precedence number (larger number is *lower* precedence; 1000 is precedence of goals), *fixity* is a two or three letter symbol giving fixity and associativity (`f` indicates the operator, `x` indicates subterm at *lower* precedence, `y` indicates subterm at *same* precedence), and *operator* is the operator to declare.

Example: Prolog imperative for loop

```
:- op(950, fx, for).    :- op(940, xfx, in).
:- op(600, xfx, '..').  :- op(1050, xfy, do).

for Generator do Body :-
        (   call(Generator),
            call(Body),
            fail
        ;   true
        ).

Var in Low .. High :-
        between(Low, High, Var).

Var in [H|T] :-
        member(Var, [H|T]).
```

Example: Prolog imperative for loop


```
?- for X in 1 .. 4 do format('~t~d ~6|^ 2 = ~d~n', [X, X^2]).
    1 ^ 2 = 1
    2 ^ 2 = 4
    3 ^ 2 = 9
    4 ^ 2 = 16
true.
?- for X in [2,3,5,7,11] do
|     ( X mod 2 =:= 0 -> Parity = even
|     ; Parity = odd
|     ),
|     format('~t~d~6| is ~w~n', [X,Parity]).
     2 is even
     3 is odd
     5 is odd
     7 is odd
    11 is odd
true.
```

Haskell operators

Haskell operators are simpler, but more limited. Haskell does not support prefix or postfix operators, only infix.

Declare an infix operator with:

associativity precedence operator

where *associativity* is one of:

- `infixl` left associative infix operator
- `infixr` right associative infix operator
- `infix` non-associative infix operator

and *precedence* is an integer 1–9, where lower numbers are *lower* (looser) precedence.

Haskell example

This code defines % as a synonym for mod in Haskell:


```
infixl 7 %

(%) :: Integral a => a -> a -> a
a % b = a `mod` b
```

Grammars

Parsing is based on a *grammar* that specifies the language to be parsed.

Grammars are defined in terms of *terminals*, which are the symbols of the language, and *non-terminals*, which each specify a linguistic category.

Grammars are defined by a set of rules of the form:

$$(nonterminal \cup terminal)* \rightarrow (nonterminal \cup terminal)*$$

where $\cup$ denotes set union, $*$ (*Kleene star*) denotes a sequence of zero or more repetitions, and the part on the left of the arrow must contain at least one *non-terminal*. Most commonly, the left side of the arrow is just a single *non-terminal*:

$$\begin{aligned}
expression &\rightarrow expression\ \texttt{'+'}\ expression \\
expression &\rightarrow expression\ \texttt{'-'}\ expression \\
expression &\rightarrow expression\ \texttt{'*'}\ expression \\
expression &\rightarrow expression\ \texttt{'/'}\ expression \\
expression &\rightarrow number
\end{aligned}$$

Here we denote terminals by enclosing them in quotes.

Definite Clause Grammars

Prolog directly supports grammars, called *definite clause grammars* (DCG), which are written using a very similar syntax:

- Nonterminals are written using a syntax like ordinary Prolog goals.
- Terminals are written between backquotes.
- The left and right sides are separated with `-->` (instead of `:-`).
- Parts on the right side are separated with commas.
- Empty terminal is written as `[]` or `‘‘`

For example, the above grammar can be written as a Prolog DCG:

```
expr --> expr, ‘+‘, expr.
expr --> expr, ‘-‘, expr.
expr --> expr, ‘*‘, expr.
expr --> expr, ‘/‘, expr.
expr --> number.
```

Producing a parse tree

A grammar like this one can only be used to test if a string is in the defined language; usually we want to produce a data structure (a parse tree) that represents the linguistic structure of the input.

This is done very easily in a DCG by adding arguments, ordinary Prolog terms, to the nonterminals.


```
expr(E1+E2) --> expr(E1), ‘+‘, expr(E2).
expr(E1-E2) --> expr(E1), ‘-‘, expr(E2).
expr(E1*E2) --> expr(E1), ‘*‘, expr(E2).
expr(E1/E2) --> expr(E1), ‘/‘, expr(E2).
expr(N)     --> number(N).
```


We will see a little later how to define the `number` nonterminal.

Recursive descent parsing

DCGs map each nonterminal to a Prolog predicate that nondeterministically parses one instance of that nonterminal. This is called *recursive descent parsing*.

To use a grammar in Prolog, use the built-in `phrase`/2 predicate: `phrase(`*nonterminal*,*string*`)`. For example:


```
?- phrase(expr(Expr), `3+4*5`).
ERROR: Stack limit (1.0Gb) exceeded
```


This exposes a weakness of recursive descent parsing: it cannot handle left recursion.

Left recursion

A grammar rule like


```
expr(E1+E2) --> expr(E1), '+', expr(E2).
```

is *left recursive*, meaning that the first thing it does, before parsing any terminals, is to call itself recursively. Since DCGs are transformed into similar ordinary Prolog code, this turns into a clause that calls itself recursively without consuming any input, so it is an infinite recursion.

But we can transform our grammar to remove left recursion:

1. Rename left recursive rules to *A*_`rest` and remove their first nonterminal.
2. Add a rule for *A*_`rest` that matches the empty input.
3. Then add *A*_`rest` to the end of the non-left recursive rules.

Left recursion removal

This is a little harder for DCGs with arguments: you also need to transform the arguments.

Replace the argument of non-left recursive rules with a fresh variable, use the original argument of the rule as the first argument of the _rest added nonterminal, and that fresh variable as the second. So:


```
expr(N)      --> number(N).
```

would be transformed to:


```
expr(E) --> number(N), expr_rest(N, E).
```

Left recursion removal

For non-recursive rules, use the argument of the left-recursive nonterminal as the first head argument and a fresh variable as the second. Then use the original argument of the head as the first argument of the `_tail` call and a fresh variable as the second argument of the head and of the `_tail` call. So:


```
expr(E1+E2) --> expr(E1), '+', expr(E2).
```


would be transformed to:

```
expr_rest(E1, R) --> '+', expr(E2), expr_rest(E1+E2, R).
```

Ambiguity

With left recursion removed, this grammar no longer loops:


```
?- phrase(expr3(Expr), '3-4-5').
Expr = 3-(4-5) ;
Expr = 3-4-5 ;
false.

?- phrase(expr3(Expr), '3+4*5').
Expr = 3+4*5 ;
Expr = (3+4)*5 ;
false.
```


Unfortunately, this grammar is ambiguous: negation can be either left- or right-associative, and it's ambiguous whether + or ∗ has higher precedence.

Disambiguating a grammar

The ambiguity originated in the original grammar: a rule like

```
expr(E1-E2) --> expr(E1), '-', expr(E2).
```


applied to input "3-4-5" allows the first `expr` to match "3-4", or the second to match "4-5".

The solution is to ensure only the desired one is possible by splitting the ambiguous nonterminal into separate nonterminals, one for each precedence level. The above rule should become:

```
expr(E-F) --> expr(E), '-', factor(F)
```


before eliminating left recursion.

Disambiguating a grammar

This finally gives us:


```
expr(E) --> factor(F), expr_rest(F,E).

expr_rest(F1,E) --> '+', factor(F2), expr_rest(F1+F2,E).
expr_rest(F1,E) --> '-', factor(F2), expr_rest(F1-F2,E).
expr_rest(F,F) --> [].

factor(F) --> number(N), factor_rest(N,F).

factor_rest(N1,F) --> '*', number(N2), factor_rest(N1*N2,F).
factor_rest(N1,F) --> '/', number(N2), factor_rest(N1/N2,F).
factor_rest(N,N) --> [].
```

Handling terminals

The "terminals" in a grammar can be whatever you like. Traditionally, syntax analysis is divided into *lexical analysis* (also called *tokenising*) and parsing.

Lexical analysis uses a simpler class of grammar to group characters into tokens, while eliminating meaningless text, like whitespace and comments.

Tools are available for writing tokenisers, but you can write them by hand or use the same grammar tool as you used for parsing, such as a DCG.

We will take that approach.

DCG for parsing numbers

In addition to allowing literal ‘strings‘ as terminals DCGs allow you to write lists as terminals (in fact, ‘strings‘ are just lists of ASCII codes).

DCGs also allow you to write ordinary Prolog code enclosed in {curley braces}. If this code fails, the rule will fail. We can also use if->then;else in DCGs.


```
number(N) -->
        [C], { ‘0‘ =< C, C =< ‘9‘ },
        { N0 is C -‘0‘ },
        number_rest(N0,N).
number_rest(N0,N) -->
        (   [C], { ‘0‘ =< C, C =< ‘9‘ }
        ->  { N1 is N0 * 10 + C - ‘0‘ },
            number_rest(N1,N)
        ;   { N = N0 }
        ).
```

Demo

Finally, we have a working parser.

```
?- phrase(expr(E), ‘3-4-5‘), Value is E.
E = 3-4-5,
Value = -6 ;
false.

?- phrase(expr(E), ‘3+4*5‘), Value is E.
E = 3+4*5,
Value = 23 ;
false.

?- phrase(expr(E), ‘3*4+5‘), Value is E.
E = 3*4+5,
Value = 17 ;
false.
```

Going beyond

This is just the beginning. Take Programming Language Implementation to go further with parsing and compilation. A few final comments:

- DCGs can run backwards, generating text from structure:

```
flatten(empty) --> []
flatten(node(L,E,R)) -->
        flatten(L),
        [E],
        flatten(R).
```


- Haskell has several ways to build parsers:

`Read`	type class for parsing Haskell expressions; opposite of `Show`.
`ReadP`	More general, more efficient string parser.
`Parsec`	Full-fledged file parsing.