Contents

1 Overview

Declarative programming is very different to imperative code. It requires a very different mindset to understand and code well in it.

Why declarative

- Grants a different perspective, focusing on what is to be done rather than how
- Work at a higher level of abstraction
- Easier to use more powerful programming techniques
- Cleaner semantics compared to conventional programs

Logical, Imperative, Functional

- Imperative based on commands, each command is executed with side effects
- Logical aims to find values that sat a set of constraints. Constraints have no side effects
- Functional based on expression, each being pure functions

Side effects

- Code has side effects if it modifies state or has an observable interaction with the outside world
- Examples: modify globals, modify arguments, raise exceptions, IO
- Declarative languages have functional updates that build ontop of existing data structures when updating. This requires special memory management to maintain performance. Benefits is that don't need to worry about mutability and allows undo, making parallel programming easier.
- Code without side effect can have stronger guarantees on: no modification of parameters, type safety and type systems, no side effects. This helps with debugging and parallelism

Blub paradox

- Understanding programming language differences requires understanding the more powerful ones
- Looking down at less powerful languages is easy, but looking up at more powerful ones and understanding why is hard. From the POV of the more powerful ones, this is also the same.

2 Prolog Introduction

Programming paradigm

- Imperative, execute instructions step by step, based on John von Neumann
- Functional, function and expressions mapping input to outputs, based on lambda calculus
- Logic, relations with predicates that relates them, based on predicate calculus

Prolog comments are written with a %.

Relation

- Specifies a relationship between arguments/columns
- The fact `parent(a, b)` specifies a part of the relationship
- The name of the relation is called a predicate (the function name)
- The arity of the relation is the number of columns it have.

Fact

- A statement with an entry/row to the relation
- Many facts define a relation

Rules

- An alternative way to define relations
- Has the form `Head :- Body`, where both are goals and body can be compound
- Means that if Body is true, Head is true
- A relation is defined with any number of clauses in any order
- Can be recursive

Prolog files

- File containing facts and rules have a name with: lowercase first, only letters/digits/underscores rest, end with pl
- Source file loaded by typing `[name].` which returns true if it is correct

Terms

- A term is the only datastructure in prolog
- Can be: atom, functor, variable

Clause

- Lines in the program file
- Can be: fact, rule

Variables

- Variables begin with capital letter (or underscores).
- Denotes an unassigned term. Variables with the same name must be assigned with the same term during resolution

Queries

- Queries are executed after loading facts/rules
- Queries look like terms, except that prolog tries to sat it
- Can consist of multiple goals making it a complex goal. Comma implies conjunctive goals, semicolon for disjunctive goals. Conjunction has higher precedence than disjunction like in Python

- Queries with a variable is existental: are there any values for the variable making the query true
- If more than one answer, prolog prints them all, semicolon to see next, enter to finish

Modes

- Mode denotes the arguments which are bounded (grounded) or unbounded (ungrounded) for a query
- Two similar queries with different modes implies that their bounded arguments are different

Equality

- Infix operator =. Used to assert that two terms are equal.

Negation

- Written as `\+` as a prefix operator. Higher precedence than both conjunction and disjunction. Must need space between it and parenthesis
- Disequality is an infix `\=`
- Prolog executes negative by trying to prove the expression. If sats, return false. Otherwise, return true.

Closed world assumption

- Assumes that all true things can be derived from the program, regardless of the semantic meaning of predicates
- Therefore we should bind all variables before negation, so that it performs as expected

Execution order

- Goals are executed left to right in a query or clause body.
- So place goals that binds the variables in a negation before the negation

Datalog

- The fragment of prolog with facts/rules/queries
- Generalization of what is provided by relational databases. Some modern databases provide datalog features as well or use its implementation techniques
- Can translate relations to tables in databases, and queries to SQL/relational-algebra, and vice versa

3 Terms and Arithmetic

Terms

- The only data structure in prolog
- Can be atomic, variable, or compound term
- Since prolog is dynamically typed, the arguments of a term (functor) can be any other term. No need to explicitly declare types

Atomic term

- integer, floats, atoms
- Begin with lowercase. Can also be written with single quotes if space is needed

Variable term

- variables written captialized, denoted a single unknown term
- the single underscore is special in that it specifies a different variable every time
- during resolution, variables are single-assignment in that they can only be bounded once

Compound term

- a functor (function symbol) followed by one or more arguments in parntheses comma separated.
- Functors have the same naming convention as atoms.
- Some functors can be applied infix

Grounded

- A term is ground if it contains no variables. Nonground if it contains at least one variable

Substitution

- A mapping from variables to terms
- Applying a subsitution to a term will consistently replace all occurrences of relevant variables with their mapped terms
- Applying θ to term t yields the term $t\theta$

Instances

- y is an instance of x iff we can apply a sub on x to get y, or $x\theta = y$
- Grounded terms only have one instance. Ungrounded have infinite

Unification

- A substitution θ unifies two terms t, u iff $t\theta = u\theta$. If so, θ is a unifier
- Essentially unification is to find a substitution such that the two terms are equal syntactically after substitution (allowed ungrounded equality)
- Two terms are syntactically equal if their variables are equal, atoms are equal, functors are equally named with equal terms
- If a query contains $t = u$ where both are terms, prolog will attempt to unify t and u with a substitution

Arithmetic

- Terms like $a * b$ are just data structure shorthand for $*(a, b)$, where the functor $*$ is infix
- The $=$ operator on arithmetic terms does not evaluate them but only unifies them

- The infix predicate *is* will evaluate arithmetic expressions on the RHS before unifying

$$X \quad is \quad 6 * 7 \implies X = 42$$

  Note that *is* requires the second argument to be grounded when it's unifying

- There are ways to do arithmetic in different modes, where we allow variables anywhere in the terms. Will talk about later

Arithmetic operators

- $+, -, *, /, //, mod, -x$ Arithmetic operators
- $integer, float$ coersions
- $<, <=, >, >=, =:=, = \ =$ Arithmetic predicates. Note that equality here are on numbers and returns a boolean atom

Lists

- Special functor syntax for lists
- Empty list is []. Non empty list is $[H|T]$ where H is the head element and T is the tail list
- If multiple head elements, use `[1,2,3|[4|[]]]`
- If all elements are known, use $[1, 2, 3]$

Proper list

- Predicate to check if a term is a list: empty or non-empty. Required since prolog has no type system
- `proper_list(List)` will be true if `List` is a list term

Singleton variables

- Prolog warns singleton variables, variables only used once that are irrelevant. Likely indicate a typo
- Can suppress the warning by replacing singleton variables with underscores
- Should always fix this

Append list

- `append(A, B, C)` checks if appending list B after A results into C
- Implementation similar to proper list, since both pattern match on the list data structure. In general, code to handle a term often pattern matches the data structure of the term
- Typical append mode of `append([1,2], [3,4], List)`
- Other modes where: only first is ungrounded, only second is ungrounded, only third is grounded

Length

- Predicate relating a list in argument 1 to its length in argument 2
- When used in a mode where first argument is unbound, prolog resolves to a list of (potentially distinct) unbounded variables starting with underscores followed by different numbers

List member

- `member(Ele, List)` checks if Ele is in List
- Can be implemented using append or using recursion. Recursion is a bit more efficient since it doesn't require building a list of elements before Ele

Combining list predicates

- Think about how to check if the result is correct rather then how to compute it. Think of what instead of how
- To implement `take(N, List, Front)`, do

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

  since it should hold when Front is length N and is the first N elements of List.

4 Resolution and Semantics

Syntax is the prolog code that is written. Semantics is the meaning of the prolog code that is written.

Interpretation

- Each atomic term stands for an entity in the domain of discourse (world)
- Each functor stands for a function from n entities (arity) to one entity in the domain of discourse
- Each predicate stands for a relation between n entities in the domain of discourse (arity)
- Interpretation is a mapping from the program syntax to the world that the program represents. `higher(a,b)` has an interpretation that a is higher than b

Predicate interpretation

- A function from all possible combinations of n terms to a truth value
- Set of n-tuples each with n terms. Every tuple in set is mapped to true and other tuples are false
- Defining a predicate can define either of the two interpretation

Predicate definition interpretation

- A predicate is defined by a finite number of clauses, each one being an implication statement
- The meaning of a predicate is: start with universial quantifier over variables, a disjunction of the body (with variable assignments) of all the clauses implies the predicate
- To incorporate the closed world assumption, the implication between the disjunction and the predicate is bidirectional (iff). This creates the Clark completion of the program

Clause interpretation

- The variables in the head are universially quantified over the entire clause. Variables only in the body are existentially quantified only over the body.

- For all possible terms A, B, $head(A, B)$ only when there exists a term C where $body(A, C) \wedge body(C, B)$
- This is equivalent to the interpretation of all variables being universially quantified at the start

Semantics

- A logic program P contains a set of clauses. The semantics/meaning of the program is its set of logical consequences as ground clauses
- A ground clause c is a logical consequence (entailed by) of a program P if P makes c true. Ofc a negated ground clause $\neg c$ is a logical consequence if c is not a logical consequence of P
- For most logical programs, the semantics set is infinite

Finding semantics

- The immediate consequence operator T_P takes a set of ground clauses C and outputs the set of ground clauses that are immediate logical consequences of clauses in P using C
- This will always include all ground instances of clauses in P.
- For each clause $H \leftarrow G_1, G_2$ in P, if C contains instances of G_1, G_2, then said instances of H is also in the output set
- The semantics of a program P is $T_P(T_P(\dots(\emptyset)\dots)$, basically T_P applied infinitely times. The iteration can terminate until a fixed point where $T_P(C) = C$

Procedural clause interpretation

- Instead of a universal quantifier followed by implication used in logical reading, procedural reading says that: to show the predicate for some variable assignments, it is sufficient to show the body is true
- Used in SLD resolution algorithm in Prolog

SLD Resolution algorithm

- Have a goal G and program P
- Output an instance of G that is a logical consequence of P, or false
- Basic algorithm

```
resolvents = G
while non_empty(resolvents)
 choose goal A from resolvents
 choose clause A' <- B1, B2 from P such that A unifies with A' using theta
 if no clause exists
  exit
 replace A with B1, B2 in resolvents
 apply theta to resolvent
 apply theta to G

if resolvents empty
 return G
```

```
else
 return false
```

- Backtracking algorithm. Resolution can fail. If so, backtrack until the last choicepoint by selecting a different clause to unify
- A choicepoint is created when branches on clauses for goal exists. Prolog backtracks to the most recent choicepoint by removing all variable bindings made after the choicepoint and unbound them, then begin resolution with the next matching clause until all none matching clauses failed. It then deletes the choicepoint and backtrack
- Note that prolog doesn't try another goal if the choicepoint clauses are exhausted
- The resolution/goal will succeed if just one resolution search branch succeeds

Resolution search tree

- Nodes are current resolvents
- Edges between nodes if a clause is unified with a resolvent while applying the substitution to the entire resolvent

Order of execution

- The order in which the goals are resolved and clauses are tried does not matter for correctness (which a goal can be resolved), but does impact efficiency if we first the best goal/clause first
- The order will determine the ordering of solutions found, but not which solutions are found since all goals need to be resolved and all matching clauses need to be tried
- During SLD resolution, Prolog selects the first goal to resolve, and first matching clause to unify

Indexing

- Used by prolog to efficiently select relevant rules/facts when unifying a query, reducing complexity of searching clauses
- By default it will sequential linear search clauses
- Indexing will optimize this using a lookup table (ground clause partial head to list of clauses), often based on predicate name/arity plus ground terms in the first argument
- SWI uses JIT indexing, which dynamically adjusts indexes based on call/resolving patterns
- Indexing will only change performance not semantics
- To optimize indexing, place the most selective/restrictive argument (argument that narrows down the applicable clauses the most) first in the predicate

5 Debugging Prolog

Certain prolog rules will only work in certain modes

- A query hangs if prolog attempts to search for another solution, but never reaching one. This is an infinite backtracking loop.

- The recursive reverse rule will hang in the mode where one argument is unground and the other is ground
- Note that a query with infinite solutions is not hanging. It needs to execute forever without outputing more solutions

To prevent infinite backtracking on reverse

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

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

where `same_length` checks if two lists are equal length. This breaks the backtracking in `rev1` so it doesn't try lists with length longer than the input.

Debugger

- Prolog uses the byrd box model
- Each goal execution is represented as a box with ports for entering and exiting the box.
- Prolog has four ports to each box:
- Call for initial entry
- Exit for successful completion
- Fail for final failure
- Redo for backtracking into the goal
- An example port usage is: call, exit, redo, exit, redo, fail

Debugger usage

- trace/0 will enable the debugger
- nodebug/0 will disable the debugger
- Debugger pauses on each port and prints: current port, resolution depth, goal with variable bindings
- Type c/enter/creep to continue execution after each pause
- Type r/retry to time travel back to the time when the goal is first executed
- Type s/skip to time travel forward over the execution of a goal until it exits/fails
- To track down a bug: skip/creep when call/redo/exit with correct results, retry then creep when seeing an incorrect exit or fail.
- Type b to pause and entry a break level
- Type a/abort to abort

Spypoints

- use the spy(predicate) predicate to set a spypoint at a named predicated, and nospy(predicate) to stop.
- When prolog reaches a predicate with a spy point, it enables the debugger.
- Type l/leap to skip to the next spypoint
- Type +/- to enable and disable the spypoint at the current call

Documentation

- Code file should have two levels of documentation: file level and predicate level
- Files should start with comments that outlines: purpose, author, date written, summary of code
- Predicates should have comments that identifies: meaning of each argument, what it does, modes where it can operate
- To document a predicate's mode: prefix arguments with + to indicate input that is grounded, − for output that may be ungrounded, and ? for either.
- : indicates that the argument is a meta-argument: code represented as terms
- To document a predicate: write a mode documentation for every mode that the predicate is designed to work in

Determinism

- The execution is deterministic when each goal is only unified with one rule. Deterministic execution will have no backtracking.
- Ideally we want to write rules that are determinstic: a goal can only unify with one rule head. This can be done by constraining each clause of a predicate to only unify under a unique scenario

Choicepoints

- When a clause unifies by there are later clauses that may also succeed, prolog will leave a choicepoint for backtracking later
- Choicepoint is back since it invokes backtracking and disables last-call-optimization. Deterministic rules should not create choicepoints.
- Prolog will backtrack to the latest choicepoint after the first assignment exits

If-then-else

- The syntax `(expr -> then ; else)` will try to unify expr to be true. If it is true, then it will query then. Otherwise, it will query else.
- Importantly, prolog doesn't backtrack to sat the expr after it found a substitution. This makes it deterministic
- Prefer indexing over if-then-else for determinism, since if-then-else leads to code that doesn't work (or produce not enough solutions) in multiple modes

6 Tail Recursion

Tail recursion

- A predicate is tail recursive if the only recursive call in any rules of the predicate is the last goal in its body
- More applicable and better in prolog since data structure memory is constructed at the heard before the recursion call, so no computation is needed after the last recursion call.
- Tail recursion optimization speeds up tail recursive predicates by making them execute like loops.

Stack

- Prolog implements calls and exits using a stack.
- A stack frame contains the local variables for a predicate call
- If the caller has nothing to do after its last call, Prolog will release its frame before calling. This is last call optimization

TRO

- A special case of last call optimization where the last call is recursive.
- Without TRO, requires a stack frame per iteration. With TRO, recursion executes like a loop with constant stack size
- If the caller leaves a choicepoint before last call, it will stay on the stack and prevent the caller's frame to be released.
- Therefore for TRO to work, the predicate must be deterministic and tail recursive

Non-tail recursive factorial

- Each recursive call for each level of the factorial creates a stack frame.
- Stack size is proportional to N, so stack explosion.

Accumulator

- In general to make a predicate tail-recursive, add an accumulator to its arguments and convert the predicate to do its action involving the accumulator
- Essentially the accumulator is the answer passed to each nested recursive call
- Base case will assign the result to the accumulator
- Recursive case will compute more onto the accumulator, passing it into the recursion
- Key is to define the accumulator and how it is updated

Factorial accumulator example. $fact(N, A, F)$ means if F is the factorial of N multiplied by A.

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

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

fact(N, F) :- fact(N, 1, F).
```

For-loop accumulator

- Accumulator is similar to the sum that is updated in the body of a for loop
- The recursion call updating the data structure is similar to decrementing the i in the for loop

Transformation

- A systematic way to transform a predicate to be tail recursive:
- First write the non-tail recursive predicate, $f(N, F)$
- Then write the definition of the accumulator version using the non-tail recursive predicate, $f(N, A, F)$
- Then replace the recursive call in the accumulator version $f(N, A, F)$ by the non-tail recursive body (unfolding)
- Simplify the body
- Recognize that the ending goals is similar to the body of the original accumulator version with a substitution, $f(N, A, F)$. Replace it with a recursive call to $f(N, A, F)$ under the substitution (folding)
- $f(N, A, F)$ is now tail-recursive

TRO performance

- It will always reduce the stack space by a constant factor
- If it can replace an operation with a computation of lower complexity, it can improve performance by a non-constant order.

Tail recursive reverse

```
rev([], A, A) :- A.

% reverse first argument and append it in front of A
rev([X|Xs], A, List) :-
 rev(Xs, [X|A], List).

rev(A, List) :-
 same_length(A, List),
 rev(A, [], List).
```

Note that at each stage of the accumulator, it takes X and appends it in front of A. This takes $O(n)$. Note that TRO will not occur under a mode where the first argument is unground (since it creates a choicepoint between the rules). Hence we ground it by calling `same_length` on A.

Difference pairs

- The trick of an accumulator is common: predicate outputing a list will have an extra argument (accumulator) specifying what comes at the end of the list
- The difference pair is the pair of (accumulator, output list). The predicate aims to compute the difference between the difference pair (the XS, X in rev)

7 Higher Order Predicates

Homoiconicity

- Prolog is homoiconic, meaning that prolog code can run and manipulate prolog programs as data (metaprogramming).
- The predicate `clause(+Head, -Body)` is true if Head can be unified with a knowledge database clause head, and the Body with the corresponding clause body under the same variable assignment. Note that the parameters are compound/functor terms that are treated as predicates
- A fact is a clause where the body is true.

Autoloading

- Most prolog builtins are autoloaded: prolog detects when you use it for the first time (predicate is undefined) and autoload it.
- Higher order call for a predicate not yet autoloaded will fail.
- To manually load a predicate, either: call it directly, call `use_module(library(...), [...])`.

A prolog interpreter

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

Higher order call

- The `call/1` predicate executes the term as a goal, returning true if it is unifiable and false otherwise.
- Allows treating any prolog term as a goal and executes it.
- To support currying, other arities of `call` will pass further arguments into the functor argument before calling it. This allows the same goal template to be used multiple times with different arguments for `call`.

Maplist

- The `maplist(:Goal, ?List1, ?List2)` applies the predicate (Goal) to each pair in List1 and List2.

- It is builtin, which reduces boilerplate in coding the recursion everything and makes the intent clearer
- Other arity versions of `maplist` will allow more or less arguments into Goal.

All solutions

- The predicate `setof(+Template, :Goal, -List)` binds List to a sorted (undup) list of all instances of Template that sats Goal
- Template can be any term, usually containing variables in Goal. Setof does not further bind any variables in Template.
- If Goal has variables not in Template, setof will backtrack over all distinct instances of these free variables in Goal and do setof assuming the bindings (free variable is A, List is blah, free variable is B, List is blah blah).
- To use existential quantification in Goal, write the variables before caret in Goal. This prevents backtracking. Multiple variables before caret are in [], and multiple goals after caret are in ()
- `bagof/3` is similar to `setof/3`, except that it doesn't sort or dedup solutions. This is not purely logical since the order of the solutions should not matter nor should duplicates exists (but bagof enforces an order).

Term comparison

- The ordering predicates can compare all terms: @ <, @ =<, @ >, @ >=.
- Only use these for grounded terms since the term might change class after unification (variable to atom after unification).
- The ordering between terms from low to high: variables, numbers, atoms, compound. Ordering within each class as expected. Compound terms are first ordered by arity, then by functor name, then by arguments left to right.

Sorting

- Three builtins to sort grounded lists according to @ < ordering
- `sort(+List, -Out)` sorts and removes duplicates
- `msort(+List, -Out)` sorts without removing duplicates
- `keysort(+KeyValue, -Out)` stable sorts X part of $X - Y$ terms

Determining term types

- integer, float, number, compound, atom are true if the terms are their respective types (at the call site). They DO NOT instantiate variables in arguments and are false for variables.
- var for unbounded variable, unvar for bounded variable. ground for any grounded term recursively
- Used to write code that works in different modes by doing an if-else on argument modes to dynamically dispatch to implementations that works on specific modes

Input output

- Not pure or logical, io operations executed when they are reached in resolution, not undone on backtracking
- `read(X)` reads a term into X. `write(X)` prints a term X.
- Handle code that does IO carefully, since their ordering matters. Recommend to isolate IO code into their own small part and maintain most code IO free

Can define custom operators by `op(+Precedence, +Infixness, +Name)`. This is more complex and allows clearer terms

The predict `assertz(:Fact)` will add the fact or rule into the knowledge database.

Cut

- The cut operator ! interferes with backtracking. When it is reached during resolution, it discards all previous choicepoints and commits to the unification.
- Cuts can be used to define negation and if-then-else. It improves efficiency (good for performance) and expression no backtracking.
- Messes with logical aspect, avoid. Can fail in different modes

8 Constraint Logic Programming

Types of programming

- Imperative program specifies the exact sequence of actions to take
- Functional program specifices how the result is computed at an abstract level. The code only suggests an ordering of action that the compiler can deviate from
- Logic program specifies a set of equality cnstraints that the terms must sat, and searches for a solution
- Constraint program allows for more general constraints than logical program. The searching algo for solutions will follow a fancy algorithm.

Constraint programming problem specification

- A set of variables, each having a known domain
- A set of constraints, each involving one or more variables
- Optional objective function

Constraint programming solver

- Find a solution (set of assignments of values to variables) that sats all constraints
- The objective function maps each solution to a number. Optimizer will find the lowest cost solution

Constraint problem kinds/systems

- Herbrand constraint system. Variables are terms. Constraints are unification and equality. Prolog constraint system
- Finite domain. Each variable's domain has a finite number of elements

- Boolean sat. Variables are booleans. Constraints assert the truth of a proportional formula.
- Linear inequality constraint. Variables are real numbers. Constraints are linear inequalities.

Herbrand constraints

- Herband constraints are just equality constraints over terms. Exactly like a prolog program
- Prolog will search for solutions that sats the equality constraints

Search strategy

- Prolog uses "generate and test" to search for variable bindings that sat constraints. Non-deterministic goals generates potential solutions, and later goals tests those solution.
- Constraint logic programming uses a more efficient "constrain and generate". Allows for more sphisticated constraints. Done in prolog using attributed variables that allows custom unification process.

FD constraint solving using propagation and generate: propagation and labelling

- Propagation reduces the domain of each variable as much as possible. Done by iterating over all variables and constraints, removing value from its domain if it invalids a constraint, schedule that variable to be checked again.
- Propagation ends if: each variable has a domain size of one (success), some variable has an empty domain (failure), there are no more constraints to be done (move to labelling).
- Labelling will: pick a not-yet-fixed variable, partition its domain into k parts, recursively invokes the solver k times for each domain restriction
- The solver will alternate between propagation and labelling
- Labelling will generate a search tree. The size of the tree depends on the propagation effectiveness — more domain values eliminated will lead to smaller search tree and less searching time.

CLPFD library

- Models the finite domain on integers constraint system
- Contains for arithmetic constraints that works in all modes. These arithmetic constraints are prefixed with a hash
- `var in Low..High` to constrain a variable between two inclusive numbers. And `List ins Low..High` for a list of variables
- Propagation involves the library revising the domains of relevant variables given an arithmetic constraint. This may be enough to reduce the domains to a single element. Otherwise there may be multiple solutions. To get explicit solutions, use `label([A,B,C])`, which will enumerate all assignments that sats the constraints.

Sudoku is a classic finite domain constraint sat problem.

- Variables are the 81 numbers
- Constraints are uniqueness constraints of the variables
- Solution

```
:- use_module(library(clpfd))

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|R1], [D,E,F|R2], [G,H,I|R3]) :-
 all_distinct([A,B,C,D,E,F,G,H,I]),
 blocks(R1, R2, R3).
```

Linear inequality constraints

- Need to set up a system of linear constraints, and an objective function formula to maximize.
- The CLPR library allows for this. The variables are real numbers. Constraints are linear inequality constraints
- Write all linear inequality constraints in {}, and `maximize(X)` to find the variable assignment that optimize the variable X.
- An example is

```
{250*B + 200*C =< 10000},
{Revenue = 4*B + 6.5*C},
maximize(Revenue).
```

9 Basic Haskell

FP

- Basis of FP is equational reasoning: if two expressions have equal value, then they are replacable
- FP uses equational reasoning to rewrite a complex expression to be simpler, until it is a single literal

Comments

- Start with `--` and continue til the end of line

Offside rule

- Given n and m be the ident for two lines
- If $m > n$ (more nested), then second line is a continuation of first line

- If $m = n$ (equal nested), then the second line is a new construct at the same level as first line
- If $m < n$ (less nested), then the second line is a continuation of the parent construct that the first line is a part of (or a new construct on the parent construct level)

Lists

- List type definition `[] x:xs` for lists
- Syntax sugar `[a,b,c]` is just `a:b:c:[]`.

Function

- A function definition consists of several equations, each is a equality between the LHS and RHS
- Every function definition typically pattern matches on the LHS arguments
- The set of patterns should be exhaustive on the LHS. It is also good practice to make the set exclusive.
- In all cases, there are exactly one pattern that will be applied/matched for every possible call (unless no pattern matches).
- Typically for a function call expression, haskell will repeatedly search for a pattern matched equation and replace the expression with the pattern matched equation's RHS.

Function Syntax

- A function call contains no parenthesis or commas
- Parenthesis are only needed around individual arguments to change precedence
- The name of functions are sequence of letters, numbers, and underscores. Must begin with a lowercase letter.

Recursion

- Often a function on a data structure argument is defined to pattern match on each possible structure of the data structure, (empty list and non-empty list for lists)
- The base case is an equation that has no recursive calls
- The recursive case is an equation with a recursive call

Expression evaluation

- To evaluate an expression, haskell repeatedly iterates some steps:
- look for a function call in the expression
- search the list of equations defining the function top-to-bottom, trying to match
- set values of variables in the matching pattern with expression arguments
- replace the LHS with the RHS on the equation in the expression
- Loop stops when there are no function calls to replace

Order of evaluation

- If there are more than one function call that we can equate, the implementation doesn't define which one to use
- Church-Rosser theorem: for lambda calculus, regardless of the order that the origin term is rewritten, the final result is the same.
- Since all functions are pure, it doesn't matter which call to equate, the final result is the same (Church-Rosser theorem, haskell is a variant of lambda calculus)
- Church-Rosser theorem is not applicable to imperative languages. The order of evaluation may matter to the final result.
- Evaluation order doesn't change result, but it does change the efficiency due to short-circuiting/lazyness

Imperative and declarative

- Due to side effects, imperative program's behavior depends on its history (and order of evalaution)
- Understanding effectful programs requires thinking about history, making it harder to understand
- Managing side-effects is diffcult. Haskell guarantees no side-effects
- Main difference between pure imperative and pure functional is side-effects
- Haskell does allow one type of side-effects: exceptions. We can ignore this since it always terminates the program

Referential transparency

- No side-effects leads to referential transparency: an expression is replacable by its value
- Because a function always returns the same value on the same input
- Impure FP languages are impure because they permit some side effects like assignments or IO. So they are not refernetial transparent

Single assignment

- Imperative languages contains variables storing a value. Assignment changes that value
- FP variables are single assignment: there are no re-assignment operator, you can only define a variable's value but not redefine it after.

Assignment operator

- Explicit way. A let clause `let pi = 3.14 in ...`, pi is only defined in the local scope after `in`
- Implicit way. A variable in a pattern on the LHS of an equation `len (x:xs) = ...`, which are assigned during pattern matching. The variables are only defined in the scope of the function.

10 Haskell Types

Haskell type system

- Strong, safe, static
- Strong means no loopholes, cannot randomly type cast
- Safe means a running programming will never crash due to type errors
- Static means that types are checked at compile time, not at run time. Haskell will not start with a type errors

Basic haskell types

- Bool, True or False
- Int, 32 or 64 bits depending on platform. Integer for unbounded ints. There are other fixed bit integer types like Int64
- Double and Float
- Char for characters

List type

- List is a type constructor. It takes a type `t` and creates new type called a list of `t`
- Can be nested like `[[Int]]`
- Type constructors are similar to generics in other languages

Strings

- Strings are just a list of characters. Literally synonyms in the type system

Names of haskell types all start with capital letters

GCHI

- REPL Interpreter for haskell
- Prelude is haskell's stdlib that is automatically loaded
- `:l file.hs` to load a file
- `:t expr` to check the type of the expression
- Use `:set +t` to always print the expression type

Function types

- Each function should have its type declared for good programming style. Otherwise, haskell can infer the function type automatically.
- Function type declaration is `name :: type`
- All local variable types are also inferred is not declared.
- Type declaration helps to improve haskell's error messages, and makes functions easier to understand. It will raise an error if the function is used in a way that is incompatible with its type (in call or in definition)

Number typeclass

- All integer types belong in the typeclass `Num a`, where `a` is a type variable

- The notation: `Num a => a` means that if `a` is a numeric type, then the expression is type `a` (basically `a` is in the typeclass `Num`)
- All arithmetic operators work on all number types. All integers by default are expressed as a num typeclass

If-then-else

- If-then-else is an expression `if cond then v1 else v2`
- The else arm is not optional, and both arms are expressions
- Be aware of the offside rule when using if-else-then

Guards

- Guards specific cases to use when defining a function. The equal sign is after each guard
- If no guard conditions matches, an exception is raised

Parametric polymorphism

- A function with free type variables in the type definition is parametric polymorphic. It means that the function can handle types with any type that is parametric in the parameter
- Haskell automatically derives parametric type definitions for functions

11 User defined types

Enumeration

- Do `data Name = Enum1 | Enum2`
- The type `Name` is an arity type constructor. The enums are data constructors. Both can take arguments and must be in uppercase
- Type names and data constructor names can be the same

Standard library types

- Prefer user defined types to prevent accidential semantic switch (using Gender vs Boolean). Improve type safety
- Haskell can catch semantic errors using user-defined types
- Use separate types for separate semantic distinctions

Structure types

- Do `data Card = Card Suit Rank`

Type instance creation

- Data constructors are functions that create an instance of a user defined type
- Can use data constructors as functions

Show typeclass

- A typeclass containing a show function with signature `show :: (Show a) -> a -> String`

- Can manually implement the typeclass for type `T` using `instance (Show T) where ...`
- Can let Haskell autoderive the typeclass by `deriving Show`

Eq and Ord typeclass

- Eq typeclass exports the `==` and `/=` operator. Use `deriving Eq` to automatically derive it for a type
- Ord typeclass exports all comparison operators like `>=`, `<`. Use `deriving Ord` to derive it for a type (first sorted by order of data constructor in type def, then sorted by data constructor arguments)
- Ord typeclass depends on the Eq typeclass

Disjunction and conjunction

- Disjunction refers to enumerated types
- Conjunction refers to structured types
- Haskell allows type definition to have both at the same time — discriminated union types. Discriminated since haskell knows what variant the value is; Union since it is a structured type
- Due to disjunction and conjunction being boolean operators, this type system is an Algebraic type system with Algebraic data types

C only have undiscriminated unions, since the type system doesn't know which variant is stored in a value of said type.

Maybe

- Monad type either: `Nothing` or `Just x`
- Type constructor is a polymorphic type: `Maybe a`

12 Haskell Types Comparison

Haskell algebraic data type

- Offers a much more direct definition of the type relationship/structure that the programmer desires
- Also shorter definition free of implementation notes

Error comparison

- C implementation is more error prone due to: accessing meaningless field, forgetting to initialize some fields, forgetting to switch/process on enums.
- Java will catch the 1st and 3rd.
- Haskell can catch all three.

Memory comparison

- C representation may take more memory or risk using unions that increase complexity
- Java and Haskell has similar memory usage

Maintain comparison

- To extend the datatype enum: Java requires adding a new class and implementing method; C requires adding an enum and adding code to handle the new enum; Haskell adds a new alternative to the type, and add code to functions handling the new enum
- To extend on the interface method: Java requires adding a new abstract method and implementing for all types; C and haskell requires adding another function

Haskell pattern matching

- `case value of Pat1 -> ... Pat2 -> ...`
- Case-of is an expression construct that pattern matches on `value`. Can be used instead of functions

Warn incomplete pattern

- A flag that warns missing cases in pattern matches
- Useful in maintenance to protect against adding a new variant. Compiler can warn functions that need to be updated
- Otherwise, haskell will throw an exception when missing alternative.
- C will sliently ignore the new type and compute an incorrect result
- Java abstract method will also throw an error, grants the same safety as haskell

Data structure process comparison

- C requires pointer dereferencing to process variants, which is dangerous
- Haskell provides safe variant switching with pattern matching

13 Declarative Programming Paradigm

C to Haskell conversion

- Straight-line (step-by-step) code converted to expressions or `let ... in ...` or where.
- Loops converted to recursive functions, likely defined as an auxiliary function outside the function.

Type signature preference

- The explicit function type signature should always be of the most general form
- Prefer type classes over explicit types if operators are needed

Recursion vs iteration

- Functional languages have no iteration constructs. They use recursion instead
- Recursion only is not a limitation at all, almost all loops can be implemented using recursion, but some recursion are hard to implement in iteration

Key considerations when choosing fp

1. Code writing process. Haskell will warn about more (logical) errors such as incomplete pattern matches, easier process.
2. Reliability of code. In haskell, names of auxiliary functions provides documentation. Functions ease the correctness argument (compared to loop invariants). More reliable code produced
3. Productivity. Large library of pre-written functions allows easy reuse. Extracting loops to reusable recursive functions also increases reuse. Increased documentation may take extra time but helps everyone to understand it better later.
4. Efficiency of code. Potentially larger stack size compared to iterative version. Less efficient in general due to more stack frame allocations. Compilers can help to optimize recursive code and convert them to iterations.

Efficiency

- Typically 10% to 100% slower than C. But much faster than interpreted languages
- Tradeoff between higher-level and performance in a language. Thus the tradeoffs between productivity (high-level) and efficiency (performance)

Immutatable data structures

- Declarative languages have immutable data structures
- To update a ds, create another version with the change
- Can retain the old ds if the software requires it for: recursion, undo, gathering statistics

Garbage collector

- Declarative languages often have no memory management features
- Automatic memory management using the Haskell GC that automatically frees unused memory cells

14 Polymorphism

Polymorphic types

- Written as `data Type a b = Data a b`, to indicate that the type constructor takes two type variables a and b
- Polymorphic functions can use polymorphic types without changing the implementation much.
- Can restrict the polymorphic types in functions with type classes on its type arguments

General comparison

- Some types cannot be compared for equality. Two functions ideally are equal if all sets of input argument leads to the same outputs. It is proven that function equality is undecidable (there cannot be an algorithm that solves this generally).
- Some types can be compared for equality but not for ordering. Examples are a set of integers.

Haskell comparison typeclasses

- Equality comparison is implemented in Eq typeclass
- Ordering comparison is implemented in Ord typeclass
- Ord typeclass requires Eq typeclass

Typeclass constraint

- Do `(Typeclass a) => a` to restrict the type `a` to the typeclass
- Multiple typeclass constraints are separated by commas

Data Map

- Do `import Data.Map as M`
- Map represents a polymorphic self-balancing binary tree.
- Key functions are: insert, lookup, index, size

Deriving

- Eq, Ord, Show, typeclasses (and more) can be automatically derived on custom algebraic datatypes
- The derived comparison method is sensitive to the order of data constructors in the type definition and the ordering of argument per data constructor

Recursive types

- Recursive types using themselves in the data constructor. It requires a base case
- Non-recursive types does not

Mutually recursive types

- Mutually recursive types use each-other for recursion, but maybe not directly use itself
- For mutually recursive types, it is enough for one type to have a non-recursive data constructor

Structure induction

- Code on nonrecursive type is simple
- Code on recursive type requires: an equation for the nonrecursive data constructor, a recursive equation for the recursive data constructor (often with the constraint that the recursive call is strictly smaller than the argument)
- The recursive function definition outlines a correctness argument by induction. The number to induct on is the number of nested data constructors in the data. Base case corresponds to the base case induction. Recursive case corresponds to the induction step.
- Mutually recursive types can also have structure induction proofs.
- Picking the right type structure is important since the code tightly follows the type structure

Formality

- Formal proofs can prove the correctness of functions and entire programs
- Requires a formal sepcification of expected relationships between input/outputs of functions
- Uncommon to do formal proofs in practice due to cost.
- Functional languages typically have informal correctness arguments by specifying in natural language about the success criteria for each function

Let and where

- Both are value alias
- Let can be used anywhere, where can only be used at the top level

15 Higher Order Functions

Orders

- First order values are pure data (as in data structures)
- Second order values are functions whose arguments 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 $n-1$ (or lower) order values

First order functions are second order values. Etc

Haskell supports higher order functions/values out-of-the-box.

Backticks

- To convert any function to an infix operator, surround it by backticks
- Operators written with backticks have high precedence and is parsed left-to-right associated
- Can also define non-alphanumeric operators with custom precedence and fixity.

Lambdas

- Used to pass a namely function argument to a higher order function
- Lambda expressions are `\x y -> blah`. Syntax based on lambda calculus
- Lambda calculus uses a lambda with the argument list followed by a dot. Like

$$\lambda x.\lambda y.x$$

Partial applying (currying)

- Partial application of a function returns a closure function with the remaining inputs unfilled and the supplied input filled
- Value/point-free-definition of functions are functions defined partially without taking the last argument to operate on.

- Enclosing infix operators in parenthesis will convert it to a function. Can partially apply it too by enclosing its operands in the parenthesis.
- The type signature of functions represents the fact that all functions can be curried. Note that the arrow type operator is right associative.

Function composition

- the `.` operator composes two functions
- its type is `(.) :: (b->c) -> (a->b) -> a -> c`
- Defining a function as a sequence of functions using functional composition is called the point/value-free style

Higher order programming

- Advantages of: code reuse, higher level abstraction, set of prebuilt solutions to frequently used operations
- Code that doesn't use higher order functions tend to follow the copy-paste anti-pattern, since similar code snippets are copy-pasted for reuse
- Benefit from higher order programming scales linear with the complexity of the data structure traversed, since one implementation is required per branch per reducer (compared to one implementation per reducer)

Some common builtin higher-order-functions

- Filter
- Map, replacing loop
- Fold, replacing recursion. Derived: sum, product, concat, maximum, minimum, reverse (using flip)

Folds

- A reduction operation, reducing a list to a single value
- Three variants: left, right, balanced fold
- Parameters are: binary operator, identity element, foldable list
- Left fold has the accumulator on the left and first in the binary operator. Right fold has the accumulator on the right and second in the binary operator.
- Balanced fold is not in prelude. It uses divide and conquer (merge sort) to fold. Its assumes the same type between the list and the accumulator. Used for parallel computing
- foldl1 and foldr1 requires no identity element. They will raise errors on empty lists
- `foldl' and foldr'` are strict versions of fold. Prefer fold-left strict, then fold-right, then fold-left

Foldable

- Any types having the typeclass `Foldable t` can also be folded over.
- Foldable requires defining just foldr on a custom type.

- The foldable derived functions, like: length, sum, maximum, minimum, all works after deriving foldable

List comprehensions

- The syntax `[f x | x <- xs, cond x, let z = x, y <- ys]` creates a list using list comprehension
- The LHS of `|` is a template indicate the resulting list values
- The RHS containing `x <- xs` denotes a generator from list xs
- The RHS containing `cond x` denotes a test on elements from ealier (left) generators
- The RHS containing `let z = x` denotes a local variable definition
- The rightmost (latest) generator loops the fastest in the resulting list.

Visitor pattern

- Haskell can use a higher order function to reduce/visit a data structure. Similar to the visitor pattern
- Types of the function makes it clear that it is collecting data or create an updated data structure. Other imperative languages may have the visit method mututate existing data
- Haskell traversal functions are typically next to each other. Visitor visit methods are dispersed among the classes.

Librarys and frameworks

- Libraries export functions that your program calls. Sometimes they are higher order functions. The control is in the program. (Data structure library exporting api to work with data structures)
- Framework calls functions that the program provides when it is time. The control is in the framework. (Webserver framework calling user endpoints only when a web response demands it)
- Haskell frameworks can be a library function call at root that is higher order. Can also be mimiced in other languages. Better inplace of code generation (like OpenGL c frameworks) which removes abstraction

16 Functors and Applicative Functors

Units

- Bugs due to different units of a quantity can be avoided by wrapping the identical underlying type behind a quantity data type
- Example is a Length data type with Meters or Feet data constructors each maintaining a double. This prevents mixing meters and feet doubles.
- Another example is encoding different currencies all represented by Decimal
- There may also be bugs when the units are identical but have different semantics. Solution is to define a new type per semantic.

- Example is that both the duration seconds and unix epoch seconds are in seconds, but are defined differently and can't be mixed.

Functor

- Models a box that we can apply a function to the contents within
- Exports `fmap :: (Functor f) => (a -> b) -> f a -> f b`, which converts a function to a function over a functor box.
- `fmap = <$>` which is an infix operator
- Instances: list, maybe, functions

Applicative functor

- All applicative functors are functors. This means that we can define `fmap` using `pure, <*>`
- Models a functor where we can apply a multi-arity function to the contents within. Equivalently, a functor which can contain functions that is applied onto functors itself.
- Exports `pure :: (Applicative f) => a -> f a` which wraps an item into the applicative, and `<*> :: (Applicative) => f (a -> b) -> f a -> f b` which applies a function already in an applicative to the applicative functor contents
- Instances: list, maybe, functions

17 Monads

Any and all

- Any is true when the predicate is true on any list element. Or existential quantifier
- All is true when the predicate is true on all list elements. Or universal quantifier

Flip

- Flip reverses the order of a two arity function
- Alternatively, use the partial operation application syntax

Monad

- A monad is a type constructor that represents a computation which returns a value. Alternatively, it can be represented as a box carrying contents with some other metadata.
- These computations can be combined using bind, and can be constructed using return.
- The two operations are: `return :: a -> m a`, and `(>>=) :: m a -> (a -> m b) -> m b`
- All monads are applicatives and functors

Monad usage

- Wrap the content using return
- Apply sequencing operation using bind on the monad, which returns another monad

Monad Maybe (Either)

- Return wraps the object into just

- Bind will apply the function if it is a success variant. Otherwise it directly returns the error variant
- In a sequence of binds, the result is successful only when all invocations succeed. When a single failure occurs, no further operation is done

Unit type

- `()` is a unit type, the type of 0-tuple.
- It has only one variant, namely the type itself `()`

IO

- IO is a monad. An expression with type `IO t` means an input-output computation that also returns `t`. These are called IO actions
- The input actions are: `getChar, getLine`
- The output actions are: `putChar, putStr, putStrLn, print`
- Return will just return the value without doing IO
- Bind `f >>= g` will first perform f, then calls g with f's return value, finally returning the return value of g.

Hello World haskell

```
main =
 putStr "Hello, "
  >>= \_ ->
   putStrLn "World!"

-- or using do-notation
main = do
 putStr "Hello, "
 putStrLn "World!"
```

Do-block

- Simplifies monad bind (`>>=`) and sequence (`>>`) operations using syntactic sugar
- Do-block consists of a series of steps (each on its own line). The steps are:
- `expr` which are chained using monadic sequence
- `val <- expr` which are chained using monadic bind
- `let var = val` which are regular let bindings. The do-block cannot end with this.
- To return a value wrapped in a monad at the end of the do-block (since it must end with a monad value), use the `return` function

Apply operator

- The `$` operator applies a LHS function with an RHS argument
- It has the lowest operator precedence, so can be used in place of parenthesis

18 Monad Advanced

IO actions as descriptions

- The correct interpretation of a method that returns `IO t` is that they return both: a value of type t, and a description of the IO action to generate t
- The bind operator is interpretated as taking the description of two IO actions and combine them into one IO action (which executes both IO actions in order)
- The return method creates a no-op IO action description with a fixed value.

Description Execution (Conceptual)

- Every haskell binary program must have a function `main :: IO ()`
- Conceptually: the OS starts the program and invokes the haskell runtime system, the haskell runtime calls main and returns a description of an IO action (consisting of a sequence of nested IO actions), the runtime then executes the description IO action (which will execute each IO operation in the sequence)

Description Execution (Practice)

- The compiler and runtime in practice will ensure to execute the IO operation as soon as its description is computed, assuming that: the description is guaranteed to end up in the list of operations returned in main, and all previous IO operations are executed
- The assumptions are needed since: we don't execute IO actions that the program doesn't need, and we don't want to execute IO actions in the wrong order

Non-immediate IO action execution

- Creating an IO description does not execute it directly (nor ensure that it will be executed)
- IO actions are first class monads, so we can: put them into lists and bsts
- We can execute those IO actions in data structures later by sequencing them in main

Input process output

- Batch program reads input, processes, and prints output
- Interactive program does the above for each interaction.
- Majority of code is in the processing stage without IO
- Imperative languages does not distinguish this stage. Haskell does by the lack of IO action

Haskell IO benefits

- Only a subset of methods are IO actions
- Unit tests for non-io functions are simple input/output pairs. Testing is easy
- Code without IO can be optimizes by replacement
- Calls to functions without IO can be done in parallel

Debugging

- Cannot insert random printf in haskell code

- Debugging printfs are only for debugging, so we violate the ordering requirement of IO. `unsafePerformIO :: IO a -> a` performs IO anywhere without caring about the order. It takes an IO description (with value), runs the IO action, returns the value.
- Do not use unsafePerformIo outside of debugging.

State monad

- Used for computations that need to pass a state throughout the computation. Operations are: accessed, updated, passed around
- Simulates an imperative style with a global variable
- `newtype State s v = State (s -> (v, s))`, meaning an operation that updates the state s and returns a new value v.
- The partial type `State s` is a monad.
- The monad operation will take a state operation, perform the update, and pass the value returned into f.
- The method `get :: State s s` simply gets the current state
- The put `put :: s -> State s ()` updates the state, and return a unit monad
- To run the states, `runState :: s -> State s v -> v`

19 Laziness

Eager vs Lazy eval

- Most programming languages use eager eval, where expressions are evaluated as soon as they are bound to variables (explicit or implicit in function call).
- Haskell uses lazy evaluation, so the expr is not evaluated until it is needed
- Conditions when expr are evaluated: wants to ouput the value, want to match the value against a pattern, wants the value as input to arithmetic operation

Infinite data structures

- Laziness allows programs to work with infinite data structures, since it only evaluate the part that is required
- `take 5 [1..]` will work
- Can be used to quickly generate fib, or primes (using sieve)

Unevaluated expressions

- Expressions are not evaluated until their values are needed. Instead, Haskell stores the unevaluated expression data as a thunk
- The data for the partial expression that lazily computes the value is called a thunk (or suspension, promise)

Parametric polymorphism

- The name of type polymorphisms

- Requires that the memory values of all types to be identically sized. Mechanism, boxing and unboxing
- This common size is often the machine pointer size (word size, 64 bits). Larger elements are stored as a pointer pointing to the heap
- The arguments of a function in suspension can then be stored in an array of words.

Laziness benefits

- Allows the programmer to focus on what the method/function means, rather than exactly how it computes
- Does not waste resources, evaluate expressions at most once in the same method. This means that when the suspense is evaluated, the value is stored for further uses of the same function

Call by need

- Operations like printing, arithmetic, and pattern matching requires their subexpressions to be partially evaluated until the top level data constructor is determined. The arguments of those data constructor can be in suspense
- This is called "call by need" since expressions are only evaluated when its value is needed

Control structures as functions

- Since haskell is lazy, the trivial implementation of `ite` works and only evaluates the correct branch
- Laziness prevents unnecessary slowdowns when using functions over explicit control structures

Sorting and minimum

- Haskell sort is $O(n \log k)$ where n is the length of the list and k is the number of elements taken. Works well with laziness

Multiple passes

- Multiple chained computations are reduced to a single pass with no extra memory overhead storing previous data structures. This is unlike imperative languages which may store additional memory on them
- Hence haskell will require a maximum memory at one time equal to the size of the suspense tree, which is often much smaller than imperative languages

Lazy input

- `readFile` returns the file content lazily, in that it only reads the next character when the program needs it
- Similarly with file output and stdout

20 Performance

Laziness overhead

- A lot of suspensions are created and evaluated, requiring unpacking suspensions
- Every access to a value requires checking if the value has been computed

Dominant laziness effect

- Sometimes it speedups the program due to avoiding executing redundent computations. Sometimes it adds overhead
- The dominant effect depends on the program and its input, but often it is a lottory that evens out in the end.

Strictness

- Bottom $\perp$ is the value of an expression that loops infinitely (or throws exception)
- A function is strict if it always needs the values of all its arguments. Formally, if any of its arguments is $\perp$, then the function result is $\perp$
- $+$ is a strict function, and `ite` is non-strict
- Strictness analysis (GHC included) is a compiler pass who analyze the code of the program and mark each function as strict and non-strict
- When a function is determined strict, then the compiler will generate code that an imperative language compiler would generate instead of using suspensions (call by value)
- Otherwise if a function is deemed non-strict, the compiler will generate code that creates a suspension

Unpredicability

- Laziness makes it harder for the programmer to undersetand where the program is spending most of its time and where most memory is allocated
- Small changes in where the program needs a value can cause great changes in what parts of the suspension tree are evaluates, causing great changes in time and space complexity of program
- It is hard for programmers to be aware (at the same time) of all relevant details of the program in order to optimize it, so modern haskell compilers have good profilers to help programmer to understand the cpu and memory behaviors of their program

Memory efficiency

- In a naive implementation of BST (persistent BST), when inserting a new node, every node on the path to the new node is copied and modified.
- In a balanced bst with height $O(\log n)$, insertion has better memory efficiency under imperative language if the old version is not needed. Otherwise, haskell will do better

Deforestation

- If the code follows a pattern of converting data structures to intermediate data structures, we can optimize this by directly computing the last data structure from the firt.
- This is called deforestation (forest are the intermediate data structures) since the interemdiate ones are often trees

Simple deforesting

- Converting two maps/filters into one by composing their lambda argument
- To combine map and filter, use a custom filtermap using recursion. This can be applied in general for multiple list functions
- To compute the standard deviation, instead of three passes, we can do one pass computing all relevant information.

Cords (accumulators)

- Append takes $O(n^2)$ time in haskell. In order to get constant time concat (doublely linked list), we can switch to a cord data structure
- Cord is defined as `data Cord a = Nil | Leaf a | Branch (Cord a) (Cord a)`. Then append is always $O(1)$ due to the branch data constructor.
- The naive conversion from Cord to list is still $O(n^2)$. To be linear, the idea is to only use the $O(1)$ list cons as well as keeping an accumulator to get $O(1)$ list creation

```
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
```

- Accumulator (difference-lists) are used to grant $O(1)$ list concat when processing lsits

Sortedness check

- The obvious sorting definition needs three cases: empty, one, two
- To reduce it down to two cases, use a subfunction with the type `a -> [a] -> Bool` that checks if a is smaller than that of $[a]$.
- Without optimization, this is much faster. With optimization turned on, it is even

Optimization

- Optimization/compilation speedups the code a lot
- Always benchmark on the compiled code instead of the interpreted code