Outline

Contents

**Introduction** 2

**Specifying syntax** 4
- Regular expressions ........................................ 4
- Finite automata ........................................... 5
- Grammars and parsing ..................................... 7

**Names and bindings** 11
- Scope ...................................................... 11
- Parameter-passing ........................................ 16
- Heap storage ............................................. 24

**Types** 25

**Functional programming and Haskell** 27
- Exercises on Haskell basics ............................ 27
- Functional datatypes ...................................... 30
- Parametric polymorphism ................................ 47
- Higher-order functions .................................. 50
- "Ad-hoc" polymorphism and type classes ............ 58
- Examples of larger Haskell libraries ................. 66

**Lambda calculus** 71
- Syntax ..................................................... 71
- Reduction .................................................. 73
- Parameter-passing disciplines .......................... 76
- Simply-typed $\lambda$ ..................................... 78
- Polymorphism in lambda calculi ....................... 79

**Object-oriented programming and Scala** 80
- Subclasses and what they unlock ...................... 80
- Function values .......................................... 82
- Generic patterns involving multiple classes ........ 82
- Type variables and members ............................ 83
- Implicits ................................................. 90

**Further topics** 90

**Hints and answers for selected exercises** 90
1 Introduction

Why you’ll hate this class

• It’s so tedious!
• I could do this in Java!
• This is so weird!

What this class gives you

• The vocabulary to discuss languages
• Experience now with what may come later
  – Java is a fine teaching language
  – And it’s comfortable for industry uses
  – But remember - it was once the cutting-edge technology
• What will be in the next five programming languages?
  – Career-focused, not first-job-focused

What we’ll do

• Name and compare the ideas behind different languages
• Experience programming languages different to those you’ll see elsewhere in the CS curriculum
  – Functional programming in Haskell
  – Object-oriented programming in Scala
  – Logic programming in Prolog
  – And we will see examples in other languages including Java, Common Lisp and Perl

Assessed work in CS421

• About 8 quizzes
• On-paper homework
  – Bring it to class; sometimes I’ll ask you to turn it in, to be counted as a small quiz
• Programming homework and projects
  – Probably two major projects with one in Haskell, one in Scala
• A final exam

Assessed work in CS521

• Programming projects
  – Probably two with one in Haskell, one in Scala
• Additional reading and assignments on material beyond the undergrad level
• A final exam
Obligatory administration

• There’s a syllabus - read it!
  – There’s a D2L quiz about the syllabus due Monday
  – Note in particular: absences for university travel (inc. sports) due this week if it is to be excused

• There’s a course website — cs.uwlax.edu/~jmaraist/421-fall-18
  – Check it frequently for news, announcements, assignments, schedule, notes etc.
  – D2L for some assignment submission, some quizzes - but not announcements
  – There’s an RSS feed attached to the web site

• There’s a study guide — linked from the web site
  – Contains lecture slides and exercises
  – First several sections available now, others will be announced via course website

• There’s email: jmaraist@uwlax.edu
  – Check it frequently for feedback on assignments, Q&A
  – Expect replies within a (business) day (but typically faster)
  – Administrative stuff always by email

• There are open-door hours
  – On the first slides, on the web page
  – Or by appointment, but email at least a (business) day ahead
    * Always include the questions you’ll want to discuss: So I can be prepared, because advising and paperwork often require no meeting, because every meeting needs an agenda
  – But I am not able to linger after class for extra questions, since I have a class immediately after, usually in another building

• Always silence your gadgets
  – Consider an app to do it for you so you don’t forget

• When you pick your seat, please:
  – Computers and handhelds to the back
  – Latecomers and early-leavers to the aisle

Class materials

The textbook is Programming Language Pragmatics, Michael L. Scott

• Get yourself a copy of the book

• Undergraduates: use the textbook rental service

• Graduates:
  – The bookstore will sometimes have used copies; ask at the back desk
  – You can often find cheap copies on Amazon or other online stores
  – In the past, grad students who tried to do without the book (and with an old edition) have complained about the difficulties in getting work done

See the course homepage for information about other resources

• Books on reserve in the library

• Online tutorial sites

• Other references
On class scheduling

• The required 400-level classes — 421, 441 and 442 — are all difficult and time-intensive classes
  – It can be a challenge to manage two of them at once
  – It is rarely a good idea to take all three at once

2 Specifying syntax

2.1 Regular expressions

What is there to a language?

Syntax

• The form of a program

• Essentially two aspects of syntax:
  – How you spell stuff — specified by a regular expression (regex)
    * Basic strings
    * Concatenation of two or more regexes
    * Choice from alternative regexes
    * Arbitrarily many repetitions of some regex
  – How you put correctly-spelled stuff together — specified by a context-free grammar (CFG), often in Backus-Naur form (BNF)
    * Give a starting symbol, other nonterminal symbols which are not part of the language
    * Rules say how a nonterminal may be rewritten to a string of other nonterminals and terminals

Semantics

• The meaning of a program

  – Most of this class focuses on language semantics

Writing down regular expressions

A language is a just a set of strings

• It can be finite (the first names of the people in this class) or infinite (phrases used to represent natural numbers)
• Any plain character in the language we’re generating is a regular expression by itself

Regular expressions are a notation for writing languages

• The empty string is a regular expression. Write it this way: $\varepsilon$
• Write two regular expressions next to each other to represent concatenation
• Separate alternatives with a vertical bar
• Use the Kleene star as a suffix for repetitions
• Use parentheses to make grouping clear

Followup reading: Scott, Ch. 1
Exercise 2.1. Write regular expressions for the following languages:

1. Strings which consist of an even number of "r"s
2. Strings which start with a lower-case letter, and are followed by any alphanumeric characters
3. Strings consisting of a number of even-valued digits with a single "E" before all of them
4. Strings consisting of one or more odd digits with a single "o" in front of them

Exercise 2.2. Write regular expressions over the alphabet \{0, 1\} for the following languages [Sipser]:

1. Strings which begin with a 1 and end with a 0
2. Strings which contain at least three 1's (not necessarily in order)
3. Strings which contain the substring 0101
4. Strings which are at least three characters long, and have 0 as their third character
5. Strings which start with 0 and have odd length, or start with 1 and have even length
6. Strings which do not contain the substring 110
7. Strings which are at least five characters long
8. Any string except 11 or 111
9. Strings where every odd position (starting counting from 1) is a 1
10. Strings which contain at least two 0’s and at most one 1
11. Either the empty string or 0
12. Strings which contain an even number of 0’s, or exactly two 1’s
13. All strings except the empty string

Exercise 2.3. Scott, Exercise 2.1.

Exercise 2.4. Write regular expressions for these languages:

1. All strings over \{0, 1, 2\} except for 2 and 10
2. All sequences of lower-case letters except for three strings: file, for and from [Scott, Exercise 2.3]

Exercise 2.5. Describe in English the language generated by the regular expression a*(ba*ba*)*. Your description should be high-level — the simple intuition about the strings, rather than a transliteration of the expression into English. [Scott, Ex. 2.9(a)]

2.2 Finite automata

Regular expressions generate, automata recognize

A finite automaton is a simple, idealized machine which corresponds to a language

- It has a number of states
  - One is initial
  - One is final
• When there is an item of input, the machine *transitions* from one state to another
  – Each transition is based on a single input item — no peeking ahead!
  – The number of states, transitions and transition labels *must be finite*

• If a string’s characters give transitions from the initial state to a final state, then the automaton *accepts* the string as part of its language
  – Otherwise, it *rejects* the string

**Depicting regular expressions**

We usually draw an automaton graphically

• States are circles
  – The initial state is marked with an arrow pointing to it
  – The final states are double-circled

• Transitions are arrows from one state to another
  – Labelled with its character
  – An arrow can start and end at the same state
  – To avoid the clutter of multiple arrows, can draw one arrow with multiple labels

![Automaton Diagram]

**Exercise 2.6.** Which of these strings does the automata below accept: a, b, c, ab, bb, ba, cb, cba, cab?

![Automaton Diagram]

**Exercise 2.7.** Write finite automata (using the circles-and-arrows notation) for each of the languages in Exercise 2.2.

**Deterministic or nondeterministic?**

A finite automaton is *deterministic* if for every state and input symbol, there is at most one possible transition

• Otherwise, the automaton is *nondeterministic*

• A nondeterministic automaton accepts a string if *any* series of transitions from initial to final state exists

• With nondeterministic automata, it is acceptable to label transitions with the empty string, or with multi-character strings
• It is *always* possible to write a deterministic finite automaton which corresponds to a nondeterministic automaton
  – But the nondeterminist automaton might be more concise
• It is *always* possible to write a finite automaton for the language of a regular expression
• But it is *not* possible to find a finite automaton for *every* language

**Followup reading:** Scott, Sec. 2.1-2.2

**Exercise 2.8.** Scott, Exercise 2.4

**Exercise 2.9.** Make sure each of the automata in the Exercise 2.7 are deterministic

**Exercise 2.10.** Scott, Exercise 2.2

### 2.3 Grammars and parsing

#### 2.3.1 Context-free grammars

**From regular expressions to grammars**

Regular expressions are one way define a language

• *Context-free grammars* written in the *Backus-Naur form* (BNF) are another

• Grammars generate a language based on rules for *rewriting* special symbols which are not in the language’s alphabet into other strings
  – The rules should eventually let us rewrite to a string which uses *only* characters in the language’s alphabet
  – The special symbols are called *nonterminals*, and the characters in the language’s alphabet are called *terminals*

**Writing down grammars**

• There’s a starting nonterminal symbol, with a rule for the form it can have:
  – S → hello goodbye

• There may be other nonterminals, with rules that refer to each other
  – S → T goodbye
  – T → hello

• Use a vertical bar to separate alternative choices, or give multiple rules for a nonterminal
  – S → T goodbye
    T → bonjour | gruessgott | hola
  – S → T goodbye
    T → bonjour
    T → gruessgott
    T → hola

• Extended BNF (EBNF) includes the Kleene star and plus notations
**Exercise 2.11.** Consider this grammar $G$, with start symbol $R$ [Sipser]:

\[
R \rightarrow XRX \mid S \\
S \rightarrow aTb \mid bTa \\
T \rightarrow XTX \mid X \mid \varepsilon \\
X \rightarrow a \mid b
\]

1. Give three examples of strings in $L(G)$
2. Give three examples of strings not in $L(G)$
3. True or false: can $T$ rewrite to $T$?
4. True or false: can $T$ rewrite to $aba$?
5. True or false: can $T$ rewrite to $abb$?
6. True or false: can $T$ rewrite to $ababa$?
7. True or false: can $R$ rewrite to $ababa$?
8. True or false: can $X$ rewrite to $XX$?
9. Describe $L(G)$ in English

**Exercise 2.12.** Give context-free grammars that generate the following languages over the alphabet $\{0, 1\}$. [Sipser]

1. Strings which begin with a 1 and end with a 0
2. Strings which contain at least three 1’s (not necessarily in order)
3. Strings which contain the substring 0101
4. Strings which start and end with the same symbol
5. Strings whose length is odd
6. Strings whose length is odd and whose middle symbol is 0
7. Strings which contain the same number of 1’s as 0’s
8. Strings which contain more 1’s than 0’s
9. Strings which are palindromes

**Exercise 2.13.** Write an unambiguous context-free grammar that generates exactly the same language as the regular expression $a^* (ba^*)^*$. [Scott, Ex. 2.9(b)]

**Exercise 2.14.** Describing a grammar’s language in plain English: Scott, Exercise 2.12(a), 2.15(a)

**Regex vs. grammars**

- Every language that can be written as a regex can be written as a CFG
- What about the reverse?
- CFGs give a sort of simple memory that a regex does not have
• The same-number-as and palindrome examples cannot be written as a regex

• Although grammars are expressive enough for programming language syntax, there are nonetheless languages which they cannot express...
  – Cliffhanger! To be resolved in CS453/553

Exercise 2.15. Rewrite your regular expressions from Exercise 2.2 as context-free grammars.

Parse trees
To demonstrate that a string really is generate by a grammar, we produce a parse tree
• Each internal node labelled with a nonterminal
  – Starting symbol at the root
• Each leaf labelled with a terminal
• If there is a rule $M \rightarrow u_1 u_2 \ldots u_n$, then a node labelled $M$ could have $n$ children labelled $u_1$ through $u_n$

Followup reading: Scott, Sec. 2.3 intro (to start of Sec. 2.3.1)

Exercise 2.16. Using the grammar of Exercise 2.11 give parse trees for these strings: babb, babb, aababb.

Exercise 2.17. Scott, Exercise 2.12(b)

Exercise 2.18. Scott, Exercise 2.13(a)

Exercise 2.19. Scott, Exercise 2.15(b)

2.3.2 Grammar properties
Some properties of operators
Properties
• Fixity: infix, prefix, postfix
• Arity
• Associativity
• Precedence

Examples
• In Java and C, ++ and – can be prefix or postfix
• Negation – is a prefix operator in most languages
• The arithmetic operators are usually infix
• Negation is unary, arithmetic operators are binary
  – The $(\_ \ ? \ _ : \ _)$ operator in C is tertiary
• In the standard interpretation of arithmetic expressions, addition, subtraction, etc. are left-associative
• In the standard interpretation of arithmetic expressions, multiplication binds more tightly than addition
Bad grammar
(Parentheses are literal, bars are metasyntactic)

\[
\text{Expr} \quad \rightarrow \quad \text{Expr} \; ^\star \; \text{Expr} \; \mid \; \text{Expr} \; \ast \; \text{Expr} \; \mid \; \text{Expr} \; / \; \text{Expr}
\quad \mid \; \text{Expr} \; + \; \text{Expr} \; \mid \; \text{Expr} \; - \; \text{Expr} \; \mid \; - \; \text{Expr}
\quad \mid \; ( \; \text{Expr} \; ) \; \mid \; 0 \; \mid \; 1 \; \mid \; \ldots
\]

• What’s so bad about this grammar?
• How do we parse 3+4*5?
  – Two ways: it is ambiguous
  – A grammar is ambiguous if it lets us build more than one parse tree for the same string

Exercise 2.20. Review the grammars you wrote in previous exercises. Which are ambiguous?

Better grammar

\[
\text{Expr} \quad \rightarrow \quad \text{Expr} \; + \; \text{Product} \; \mid \; \text{Expr} \; - \; \text{Product} \; \mid \; \text{Product}
\text{Product} \quad \rightarrow \quad \text{Product} \; \ast \; \text{Power} \; \mid \; \text{Product} \; / \; \text{Power} \; \mid \; \text{Power}
\text{Power} \quad \rightarrow \quad \text{Power} \; ^\star \; \text{Basic} \; \mid \; \text{Basic}
\text{Basic} \quad \rightarrow \quad ( \; \text{Expr} \; ) \; \mid \; - \; \text{Basic} \; \mid \; 0 \; \mid \; 1 \; \mid \; \ldots
\]

• Is it still ambiguous for 3+4*5?
  • The additional structure constrains the possible derivations so that they are unique

2.3.3 Top-down parsing

Parsing
Grammars generate, parsers recognize

• Top-down or bottom-up?
  • Top-down
    – Conceptually simple
    – More restrictions on the form of grammars which are allowed
    – Efficient
    – Can be implemented directly
  • Bottom-up
    – Start with the terminal symbols, reduce them into nonterminals
    – 3+4*5
    – Lookahead
    – Usually implemented indirectly, using a generator, with a pushdown automation details via tables

• Lots of work has been done (and continues) on parsing — to come in CS442/542
Writing a top-down parser

Top-down parsers can be easy to write

- Each rule becomes a separate subroutine
- Each rule’s routine expects a string matching that rule body
  - Match terminals by finding them in the input
  - Match nonterminals by calling the corresponding subroutine

The difficulties:

- **Choice!** When there is a vertical bar |, or multiple rules for the same nonterminal, how does our program know which to pursue?
- **Left-recursion!** When a nonterminal expands to another of itself in the left-hand position

\[
\begin{align*}
  \text{Expr} & \rightarrow \text{Expr} + \text{Product} \mid \text{Expr} - \text{Product} \mid \text{Product} \\
  \text{Product} & \rightarrow \text{Product} \ast \text{Power} \mid \text{Product} / \text{Power} \mid \text{Power} \\
  \text{Power} & \rightarrow \text{Power} ^ \text{Basic} \mid \text{Basic} \\
  \text{Basic} & \rightarrow ( \text{Expr} ) \mid - \text{Basic} \mid 0 \mid 1 \mid \ldots
\end{align*}
\]

Removing left-recursion

So a lack of ambiguity is

- **Necessary** for a sensible grammar for a programming language
- But not yet **sufficient**

Must restructure the grammar to get rid of the left-recursion

- The Kleene star/plus operators of EBNF are often key tools
- We look ahead into the input to resolve choice
  - For efficiency, a solution should look only a single unit of input ahead before making each decision!

Followup reading: Scott, Sec. 2.3.1-2.3.2

Exercise 2.21. Rewrite the arithmetic grammar to remove left-recursion, and write a simple parser to evaluate strings representing arithmetic expressions.

\[
\begin{align*}
  \text{Expr} & \rightarrow \text{Expr} + \text{Product} \mid \text{Expr} - \text{Product} \mid \text{Product} \\
  \text{Product} & \rightarrow \text{Product} \ast \text{Power} \mid \text{Product} / \text{Power} \mid \text{Power} \\
  \text{Power} & \rightarrow \text{Power} ^ \text{Basic} \mid \text{Basic} \\
  \text{Basic} & \rightarrow ( \text{Expr} ) \mid - \text{Basic} \mid 0 \mid 1 \mid \ldots
\end{align*}
\]

3 Names and bindings

3.1 Scope

3.1.1 Stack model of execution

The stack model of execution

- The standard, basic organization of memory includes a *stack* and a *heap*
– The stack grows from one end of memory
– The heap grows from the other end of memory
  *(For now we’re thinking only about the stack, and will discuss the heap later)*

• Each call to a subroutine pushes a frame onto a system stack.

• Each frame contains:
  – Storage for local variables
  – Storage for arguments
  – Pointer to top of previous frame

• The frame pointer is a CPU register used to point to the current frame

• This idealized version of the system stack organization gives us a form of operational semantics
  – Explain how we resolve variable references, parameter passing
  – Better than an English description, it’s a formal model

**Example**
For a program

```
sub f() {
    var z=2
    g(1)
}
sub g(x) {
    var y=3
    ...
}
```

When `f` calls `g`:

```
Frame for f

```

```
Frame for g
```

```
Top of stack
```

```
Frame for f

```

```
Frame for g
```

Exercise 3.1. Scott, Exercise 3.4, including Java examples

Exercise 3.2. Scott, Exercise 3.9

**How do nonlocal variables work in this model?**

```
sub wrapper(x, y) {
    local z = somefn(x,y);
    nested sub inner(w, acc) {
        if (w<1) {
```
return fn2(z, acc);
} else {
    return inner(w-1, fn3(acc));
}

return inner(x, y);

How do we resolve inner's reference to z?

What about when the else branch recurs on inner?

Need an additional entry in the frame for the static pointer

- Points to the frame of the environment which encloses this frame in the source code
Followup reading: Scott, Sec. 3.1-3.2

Exercise 3.3. Scott, Exercise 3.6, in particular 3.6(b)

Exercise 3.4. Scott, Exercise 3.11: assume the P calls Q, and Q calls R.

3.1.2 Static and dynamic scope

What does this program print?

global z = 100;

sub f() {
    print z;
}

sub g(y) {
    val z = y;
    f();
}

main:
    g(10);
    print z;

• If these subroutines act like Java static methods?
• Or if they follow the static pointer as we discussed last time?
    – Then: 100
• But this is just one way of doing things!
    – A particular language could define the scope of name-binding differently

Finding z under a static scope rule

Static scope says that we should use the most closely enclosing binding to a name when accessing that name

global z = 100;

sub f() {
    print z;
}

sub g(y) {
    val z = y;
    f();
}

main:
    g(10);
    print z;

• Know (at compile time) that the in-scope reference for z from f is one enclosing scope outward
• So the code generated for f should refer through the static enclosure once to find the frame with z’s storage
• Print 100 both times

Finding \( z \) under a dynamic scope rule

Dynamic scope says that we should use the most recent binding to a name when accessing that name

• Conceptually, this means we should follow the previous frame until we find a frame which stores a value for that name

```plaintext
global z = 100;
sub f() {
    print z;
}
sub g(y) {
    val z = y;
    f();
}
main:
g(10);
print z;
```

• Not using the static enclosing-environment pointers

• The most recent binding to \( z \) is by \( g \)

• But this binding will end when \( g \) exits
  – So print 10 then 100
Dynamic scope without search
Implementations of dynamic scope avoid searching the stack by using frames to store hidden, out-of-scope bindings

![Diagram of frame store]

Then $e$ can read the current (dynamic) value of $z$ from the global frame

**Followup reading:** Scott, Sec. 3.3

**Exercise 3.5.** Scott, Exercise 3.5

**Exercise 3.6.** Scott, Exercise 3.14

**Exercise 3.7.** Scott, Exercise 3.18

**Exercise 3.8.** Scott, Exercise 3.19

### 3.2 Parameter-passing

#### 3.2.1 Call-by-value

Some vocabulary about parameters

```python
def function1(x, y) = {
    return 2*x + 3*y;
}
```

// ... val z = 10;
print function1(3, z);

- $x$ and $y$ are *formal* parameters
  - When considering `function1` by itself, we can make no assumptions about the values of $x$ and $y$
- $3$ and $z$ are *actual* parameters
  - When we call `function1`, they certainly do have specific values
- What is the relationship between formal and actual parameters?
  - That is, how does a language define that the former should be bound to the latter?

Parameter-passing mechanisms
You may never have considered the matter up for debate

- Java and C seem to have essentially the same behavior for their parameter-passing
- But just like static vs. dynamic scope, the choice of parameter-passing mechanism is a choice made by a language’s designers
Call-by-value

C’s parameter-passing mechanism is named call-by-value

- First evaluate the actual parameter (if it is an expression), and then pass that value.
- This is what we assumed in our lecture example for scope.
- Probably the most common, and in many ways the simplest, of the parameter-passing mechanisms we will see.

3.2.2 Call-by-reference

Call-by-reference

The traditional alternative to call-by-value in imperative languages

- Rather than the value itself being stored in the new subroutine’s frame, a reference to the location of that value is communicated
- Crucially, assignment to the formal parameter also update the actual parameter, since there is only a single stored value

For example, running main in

```python
def f(x) = {
  x=10
}
def main = {
  val b=5
  f(b)
}
gives
```

• Today, most commonly seen in C++
• In most languages with call-by-reference, the actual parameter must be a storage location
  -- Not (for example) an arithmetic expression
• Orthogonal to many other choices, such as static vs. dynamic scope

Call-by-value and call-by-reference

• Given

```python
sub f(int x) {
  print x;
  x=3;
  return;
}
```
• What could happen when we evaluate

```java
int y=10;
f(y);
print y;
```

### 3.2.3 Call-by-sharing

**How does Java pass parameters?**

Scalar types are clearly passed by value, but what about object types?

• In a way, they are passed by value

```java
public void f(Object x) {
    x = new Object();
    // ...
}
```

The assignment does not change a caller’s variable

```java
final Object obj = "Hello";
f(obj);
println(obj); // Still shows Hello
```

• But in a way, they are passed by reference

```java
public void g(MyObj x) {
    x.setVal(x, 1.34);
    // ...
}
```

The assignment does change a caller’s field

```java
final MyObj obj = new MyObj(2.56);
println(obj.getVal()); // Shows 2.56
f(obj);
println(obj.getVal()); // Now shows 1.34
```

**Call-by-sharing**

We know enough about pointers to realize that what we are passing is a *pointer* to the actual object.

• And moreover that pointers are passed by value

• But the behavior is distinct enough from previous languages that we categorize Java’s mechanism as distinct from call-by-value
  
  – Named *call-by-sharing*
  
  – For non-simple types, pass a reference to some shared object
  
  – Side-effects altering the object are shared
  
  – But assignments to the formal parameter do *not* alter the actual parameter in calling routine
3.2.4 Call-by-copy-in/copy-out

Call-by-copy-in/copy-out

Like call-by-reference, concerns storage locations

- Before starting subroutine, evaluate the actual parameter
- Use the result value when starting subroutine
- *When finishing subroutine, copy the final value of the formal parameter back to the actual parameter.*

**Followup reading:**  (For Sec. 3.2.1-3.2.4) Scott, Sec. 9.3


**Exercise 3.10.** Scott, Exercise 9.17. This question seems to predate the introduction of variable-length argument lists to Java and its peer languages.

**Exercise 3.11.** Trace the evaluation of this `main` routine under both call-by-reference and call-by-copy-in/copy-out parameter-passing semantics.

```perl
int y=10;
sub g() {
    print y;
}
sub f(x) {
    x=3;
    g();
}
sub main() {
    f(y);
}
```

3.2.5 Call-by-name

Call-by-name

The parameter-passing mechanisms so far all start the same way

- *“First, evaluate the expression given as the actual parameter”*

But as usual, a language designer can choose differently.

Under *call-by-name*, formal parameters are substituted with the *unevaluated* actual parameter expression when a subroutine is called.

- So the expression may be evaluated multiple times
- But not until we reach each instance of the formal parameter
- And if the expression has side-effects, the effects may occur multiple times!

Call-by-name probably seems like the oddest of the mechanisms we’ve seen so far
• But it’s not a new idea — it was introduced into programming languages in the late 1950s with ALGOL60
• We’ll see an example from Scala shortly
• (And we’re not finished with parameter-passing mechanisms yet)

**Scala example: Complaints!**

```scala
object ComplaintCount {
  var num:Int = 0
  def another() = {
    num = num + 1
    num
  }
}

class Complaint {
  println("This is Complaint #" + ComplaintCount.another())
  def sendTo(who:String) =
    println("Hey " + who + ", I have a complaint!")
}

An unsurprising example of complaining

```scala
object ComplaintCount {
  var num:Int = 0
  def another() = {
    num = num + 1
    num
  }
}

class Complaint {
  println("This is Complaint #" + ComplaintCount.another())
  def sendTo(who:String) =
    println("Hey " + who + ", I have a complaint!")
}

object SenderBV extends App {
  tellAll(new Complaint())

  def tellAll(c:Complaint) {
    c.sendTo("Tom")
    c.sendTo("Dick")
    c.sendTo("Harry")
  }
}

• Not surprising when we run it: create a complaint, and send it around

> scala SenderBV
This is Complaint #1
Hey Tom, I have a complaint!
Hey Dick, I have a complaint!
Hey Harry, I have a complaint!

Call-by-name complaining

```scala
object ComplaintCount {
  var num:Int = 0
  def another() = {
    num = num + 1
    num
  }
}

class Complaint {
  println("This is Complaint "+ ComplaintCount.another())
  def sendTo(who:String) =
    println("Hey " + who + ", I have a complaint!")
}

object SenderBN extends App {
  tellAll(new Complaint())
  def tellAll(c: => Complaint) {
    c.sendTo("Tom")
    c.sendTo("Dick")
    c.sendTo("Harry")
  }
}

• Writing => as a prefix to a method parameter type means that the argument should be passed call-by-name
  – Not evaluated when the method is called
  – Evaluated fresh each time the method is used
• Now when we run it, we create a complaint each time we reference c

   > scala SenderBN
   This is Complaint #1
   Hey Tom, I have a complaint!
   This is Complaint #2
   Hey Dick, I have a complaint!
   This is Complaint #3
   Hey Harry, I have a complaint!

Call-by-name can boil down boilerplate
If you have used Java’s HashMap classes before, you have probably written code like this:

```
Scala’s equivalent to `HashMap` includes an extra method where the second parameter is call-by-name (indicated by the `=>`):

```scala
def getOrElse(key:K, default: => V): V
def getOrElseUpdate(key:K, defaultValue: => V): V
```

Call-by-name allows these common patterns to be more directly supported in the language.

**Call-by-name without side effect**

What would call-by-name mean in the context of Haskell?

- Remember that Haskell does not have side-effects
- Does this insight let us optimize call-by-name?

- We could:
  1. Wait until a formal parameter is used before we evaluate it
  2. Share the result of the first evaluation among the other duplications of the actual parameter

- This strategy is known as call-by-need, or lazy evaluation
  - In fact, Haskell is defined to be a lazy language
  - We will see how:
    - Haskell associates laziness with data type constructors as well as with function application
    - Laziness allows much greater expressiveness when programming

3.2.6 Lecture 35 — Macros

** Macros**

- Not all applications of functions to arguments must take place at runtime
- A "function" that generates new source text from arguments is called a *macro*
- Macro facilities are fairly common, but there is great variability in what they can do
  - On one end, the C preprocessor performs simple text substitution
  - At the other end, Common Lisp allows arbitrary Lisp code to be executed at compile time to calculate source code
  - Haskell and Scala also recently added macro systems, which we might try out.
    - Which is at odds with the book’s claim that macros are anachronistic.

**C macros**

Just simple text substitution

```c
#define LINE_LEN 80
#define PI 3.141592651358979323846264338327950L
#define DIVIDES(a,n) (!(n) % (a))
#define SWAP(a,b) { int tmp = (a); (a) = (b); (b) = tmp; }
#define MAX(x,y) (((x)<(y) ? (y) : (x))
```

- Was very useful for global or program constants
- Avoids overhead of function calls
- Note the extra parentheses
• What if a or b contain a reference to t from some surrounding scope?

• What if we call MAX(++m, ++n)?
  – Rewrites to ((++m)<(++n) ? (++n) : (++m))
  – Would it be a surprise when one variable is incremented twice?

Some things to know about Lisp

• Lisp uses prefix notation: all operators are written with the function first:

  (+ 3 x (* 5 y))
  (append (list 1 2 x) y (list z 8 9))

• The parentheses are for invocation, not grouping
  – Not optional
  – Extras not allowed
  – If you play with Lisp, make your editor highlight matching parentheses

• Lisp has a defconstant form, so we wouldn’t use its macros for LINE_LEN or PI.

Lisp macros

(deffunction divides (a n)
  '(zerop (mod ,n ,a)))

• The backtick ‘quotes a piece of syntax to be inserted by the compiler.

• The comma , injects syntax within the quoted expression.

Avoiding name capture

(deffunction swap (x y)
  (let ((tmp (gensym)))
    '(let ((,tmp ,x))
      (setf ,x ,y
           ,y ,tmp))))
(deffunction max (x y)
  (let ((xval (gensym))
        (yval (gensym)))
    '(let ((,xval ,x)
           (,yval ,y))
      (if (< ,xval ,yval) ,yval ,xval))))

• gensym creates and returns a new symbol table entry, guaranteed never to be the same as any other symbol

• Note that the calls to gensym are not part of the quoted and returned syntax
  – Evaluated, and their results used, at compile time

• Single evaluation of forms in max
  – C does not have a mechanism for statement-only features like storage allocation with an expression
  – Lisp does not distinguish between statements and expressions
3.3 Heap storage

The other end of memory

In the standard organization of memory, the stack grows from one end, the heap grows from the other

• The stack is organized FIFO
• The heap has no such time guarantees
• Allocations in the heap can vary in size, remain relevant for indeterminate periods

Simple heap management

Recall memory usage in the C/C++ family, or assembly language

• Declare specific data structures via struct, or a fixed multiple of size for an array
  – Very little in the way extending a data structure once declared
• One call malloc to allocate memory, another call free to release it
• Be wary of forgetting to free unused space!
• Be wary of keeping pointers into freed space!
• Fast and low overhead, but a high burden of error-prone space management on the individual application and programmer
• Problems of fragmentation — small, isolated free spaces separated by long-lived structures

Automatic garbage collection

In the 90s, automatic garbage collection became common

• Driven by higher-level (functional, object-oriented) academic languages showing feasibility
• Part of a trend of languages coming with larger and larger runtime systems and operating system links

Mark-and-scan garbage collection

• General idea: allocate heap space from the end of memory towards the stack
  – With each allocation, set aside extra bits for marks
• When the stack and heap collide (or when the heap hits a certain size), pause from executing program, and run garbage collector
• The garbage collector starts with pointers from registers and from the stack into the heap
• “Walks” the pointers, marking everything it finds as in use
• Then everything else must not still be in use, and can be re-used
Copying garbage collection

• General idea: divide the heap into two halves, allocate from only one half at a time
  – When that half fills, pause the program and run the garbage collector
• Again starting with live pointers from the heap and stack, copy live heap space from one half to the other half
  – Update pointers as they are walked
  – After copying resume the program, continuing to allocate from the half into which we just copied, until it fills and starts garbage collection again
• Can improve locality of reference, virtual memory performance

Generational garbage collection
Motivation: take advantage of the fact that space which has been used longer will probably also stay in use longer.

Divide the heap into generations, each of which is separately collected
• Older generations are collected less frequently
• Often combined with copy-collection — each generation in two parts, copying from one to the other

Reference counting
An appealing idea
• Every allocated chunk of memory has extra space set aside
• Like mark-scan, but space not used for marks
• Keep a count of the number of other places which point to it
• Circular structures can be a problem

Followup reading: RE-read Scott, Sec. 3.2.3-3.2.4

4 Types

Why types?
• Provide context for operations
  – For example, to distinguish integer and floating-point addition
• Detect and prohibit nonsensical operations
• Documentation which is automatically checked for correctness
• Opportunities for the compiler to optimize performance
  – Because we don’t have to check cases at runtime
  – Or for example register allocation in the presence of pointers
Scalar and composite

- **Scalar** types are indivisible
  - Most built-in types: integers, booleans, characters
  - In many languages, enumerated types
- **Composite** types are data structures with several distinct components
  - Some built-in types: String in Java, for example
  - Arrays
  - Most user- and library-defined types

When are two types the same?

- Matters when passing parameters, making assignments.
- Two general ways to decide:
  - Decide based on structure
  - Decide based on their name
- Record types

Structural equivalence

- These should be considered the same:

```c
type R1 = struct {
    int a, b;
}
type R2 = struct {
    int a;
    int b;
}
```

- What if the fields aren’t in the same order?

```c
type R3 = struct {
    int a;
    int b;
}
type R4 = struct {
    int b;
    int a;
}
```

Many (but not all) languages say that these are structurally equivalent
- Once again, it is a choice for the language designer

Name equivalence

- If the name is the same, the type is the same
  - Rules out the R1, R2 equivalence of the previous slide.
- What about type aliases?

```c
typedef old_type new_type;
```
- Of course they should be interchangeable!
  ```c
typedef unsigned int mode_t;
```
- Of course they should not be interchangeable!
  ```c
typedef double degrees_fahrenheit;
typedef double degrees_celsius;
```
- Sometimes and sometimes not?
5 Functional programming and Haskell

5.1 Exercises on Haskell basics

Exercise 5.1. [Hutton Ex. 2.7.2] Correctly parenthesize these numeric expressions:

- $2^3 \times 4$
- $2 \times 3 + 4 \times 5$
- $2 + 3 \times 4 ^5$

Exercise 5.2. Keller and Chakravarty, Sec. 1 (First Steps) Ex. 1-3.

Exercise 5.3. [Keller and Chakravarty] Which of the following identifiers can be function or variable names?

- square_1
- 1square
- Square
- square!
- =square’=

Exercise 5.4. [Keller and Chakravarty] Define a new function showResult that, for example given the number 123, produces a string as follows:

```
showResult 123 ==> "The result is 123"
```

Use the function show in the definition of the new function.

Exercise 5.5. [Includes items from Hutton] Which of these expressions are well-typed, and what types do those expressions have?

- ['a', 'b', 'c']
- ('a', 'b', 'c')
- ('a', 'b', 'c', 'a', 'b', 'c')
- ['a', 'b', 1]
- ('a', 'b', 1)
- [('a', 'b', 1), (',a', 'b', 'c')]
- (False, '0'), (True, '1')]
- ([False, '0'), (True, '1')]
- ([False, True], ['0', '1'])
- ([False, '0'], [True, '1'])
- [tail, init, reverse]
Exercise 5.6. Write Haskell definitions which have the following types.

- \((\text{Int}, \text{Int})\)
- \(\text{Int} \to \text{Int} \to \text{Bool} \to \text{Int}\)
- \(\text{Char} \to (\text{Char}, \text{Char})\)
- \(\text{Int} \to (\text{Int} \to \text{Int}) \to \text{Int}\)

Exercise 5.7. [Hutton Ex. 3.11.3] What types do these functions have? Try to work them out by hand before checking your answers in GHCI.

- \(\text{second } xs = \text{head } (\text{tail } xs)\)
- \(\text{swap } (x, y) = (y, x)\)
- \(\text{pair } x \ y = (x, y)\)
- \(\text{double } x = x \times 2\)
- \(\text{twice } f \ x = f (f \ x)\)

Exercise 5.8. Write a module `LesserInt` exporting a single function `lesserInt` which takes two integers, and returns the one which is lower in value.

To wrap your function in the module `LesserInt`, create a new file called `LesserInt.hs` whose first line is `module LesserInt where`, with your definition for `lesserInt` on its own line below.

Exercise 5.9. [Keller and Chakravarty] Write a function `showAreaOfCircle` which, given the radius of a circle, calculates the area of the circle.

```
showAreaOfCircle 12.3
```

Use the `show` function, as well as the predefined value `pi :: Floating a => a` to write `showAreaOfCircle`.

Exercise 5.10. [Keller and Chakravarty] Write a function `sort2`,

```
sort2 :: Ord a => a -> a -> (a, a)
```

which accepts two Int values as arguments and returns them as a sorted pair, so that `sort2 5 3` is equal to `(3, 5)`. How can you define the function using a conditional, how can you do it using guards?

Exercise 5.11. [Keller and Chakravarty] Define a module `IsLower` with a single function

```
isLower :: Char -> Bool
```

which returns `True` if a given character is a lower case letter. You can use the fact that characters are ordered, and for all lower case letters \(ch\) we have `'a' \leq ch \leq 'z'`. Alternatively, you can use the fact that \(['a'..'z']\) evaluates to a list containing all lower case letters. Write your own version of `isLower`; do not use the standard version in `Data.Char` (or even import `Data.Char`).

Exercise 5.12. [Thompson] Write a module `DoubleAll` exporting one function `doubleAll` of type `[Int] \to [Int]` which doubles each element of a list.

Exercise 5.13. [Thompson] Write a module `Capitalize` exporting one function `capitalize` which converts all lower-cases letters in its argument to upper-case letters, but leaves the other characters alone. The Haskell `Data.Char` library contains functions which will be useful here.
Exercise 5.14. [Thompson] Write a module `CapitalizeOnly` exporting one function `capitalizeOnly` which converts all lower-cases letter in its argument to upper-case letters, leaves upper-case letters alone, and removes other characters from the result. The Haskell `Data.Char` library contains functions which will be useful here.

Exercise 5.15. [Thompson] Write a module `Matches` exporting one function `matches` of type `Int->[Int]->[Int]` which returns all occurrences of the first argument in its second argument. So for example, `matches 10 [1,10,2,10,3,10,4]` returns `[10,10,10]`, and `matches 10 [11,14,17,21]` returns `[]`.

Exercise 5.16. [Keller and Chakravarty] Write a module `Mangle` exporting function `mangle`,

\[
mangle :: String \rightarrow String
\]

which removes the first letter of a word and attaches it at the end. If the string is empty, `mangle` should simply return an empty string:

\[
\begin{align*}
mangle \ "Hello" & \rightarrow \ "elloH" \\
mangle \ "I" & \rightarrow \ "i" \\
mangle \ "" & \rightarrow \ ""
\end{align*}
\]

Exercise 5.17. [Keller and Chakravarty] Write a module ` Divider` with a function `dividedBy` which implements division on `Int`,

\[
dividedBy :: Int \rightarrow Int \rightarrow Int
\]

by first writing a helper function that returns all the multiples of a given number up to a specific limit, and then using list functions on the resulting list.

\[
\begin{align*}
dividedBy 5 10 & \rightarrow 2 \\
dividedBy 5 8 & \rightarrow 1 \\
dividedBy 3 10 & \rightarrow 3
\end{align*}
\]

Exercise 5.18. [Keller and Chakravarty] Define a module `LengthTaker` with the function `length`,

\[
length :: [a] \rightarrow Int
\]

It is quite similar to `sum` and `product` in the way it traverses its input list. Since `length` is also defined in the Haskell standard Prelude, hide it by adding the line

\[
import Prelude hiding (length)
\]

to your module.

Exercise 5.19. [Hutton Ex. 4.8.1, with solution] Use Haskell library functions to define a function `halve`,

\[
halve :: [a] \rightarrow ([a],[a])
\]

Exercise 5.20. [Hutton Ex. 4.8.2, with solution] Define a module `Third` exporting a single function `third`,

\[
third :: [a] \rightarrow a
\]

which returns the third element in a list, a) Using `head` and `tail`. b) Using list indexing `!!`. c) Using pattern matching.

Exercise 5.21. Write a module `LastItem` exporting the function `lastItem`, which returns the last item in a list.
Exercise 5.22. Write a module `LastButOne` exporting the function `lastButOne`, which returns the next-to-last item in a list.

Exercise 5.23. [Keller and Chakravarty] Write a module `CountOdds` exporting a recursive function `countOdds` which calculates the number of odd elements in a list of `Int` values:

```
countOdds [1, 6, 9, 14, 16, 22] = 2
```

Hint: You can use the Prelude function `odd :: Int -> Bool`, which tests whether a number is odd.

Exercise 5.24. [Keller and Chakravarty] Write a module `RemoveOdd` exporting a recursive function `removeOdd` that, given a list of integers, removes all odd numbers from the list, e.g.,

```
removeOdd [1, 4, 5, 7, 10] = [4, 10]
```

Exercise 5.25. Write the function `isPalindrome`, which checks if a list is a palindrome, the same backwards as forwards.

Exercise 5.26. Write a module `NeighborDups` exporting a function `noNeighborDups` that removes all consecutive duplicates from a list of integers, e.g.,

```
removeOdd [1, 4, 5, 7, 10] = [4, 10]
```

Exercise 5.27. Write a module `EncodeDecode` which exports the function `lengthEncode`, for example,

```
lengthEncode "Aaabbbcddeeeabb"
==>> [(1, 'A'), (2, 'a'), (3, 'b'), (1, 'c'),
(2, 'd'), (3, 'e'), (1, 'a'), (2, 'b')]
```

Exercise 5.28. Extend your module `EncodeDecode` of Exercise 5.27 with the function `lengthDecode`, opposite of the above.

Exercise 5.29. Write a model `ListSplitter` exporting the function `(splitListAt n xs)`, which splits a list into two lists, the first one with `n` elements.

Exercise 5.30. Consider these declarations:

```
infixl 5 'test1'
infixl 7 'test2'
```

Complete the definition of `test1` and `test2` with two function declarations — it doesn’t matter what they do, just make them distinct enough for you to tell the difference between them as easily as you could tell the difference between other operators like addition and multiplication.

How do `test1` and `test2` behave differently with respect to each other? In a series of several applications of each?

Vary the declarations to use `infixr` and `infix` instead of `infixl`, and to use various different numbers.

How does this change how the operators behave?

5.2 Functional datatypes

5.2.1 Algebraic data types

Algebraic data types

Haskell declares new data types with the `data` declaration

```
data TYPENAME = CONSTRUCTOR1 ArgType1-1 ... ArgType1-n
| CONSTRUCTOR2 ArgType2-1 ... ArgType2-m
```
• The List type is the same idea, just with special syntax

Pattern matching
All data types can be pattern-matched in a function definition or case structure

data Season = Winter | Spring | Summer | Fall

isFall Fall = True
isFall _ = False

• Similarly for lists and for built-in enumerated types like Int

Exercise 5.31. [Keller and Chakravarty] Write a module Days which exports:

• The definition of Day from this page. Module Days should export both the name of the type, and the names of its constructors.

• A function which, given a day, returns the data constructor representing the following day:

  nextDay :: Day -> Day

Exercise 5.32. [Thompson] Write a module MonthsAndSeasons which exports a type Month as an algebraic type for the twelve months (use the full name of the month as constructors, and export both the type and constructor names), and a function monthSeason which maps a month to its member of the type Season,

  data Season = Winter | Spring | Summer | Fall

Exercise 5.33. [Thompson] Consider a module Shapes with this type of geometric shapes,

  data Shape = Circle Float
              | Rectangle Float Float

  encapsulating a value for the radius of a circle, or the dimensions of a rectangle.

  1. Add functions area and perimeter which take a Shape as an argument, and return the value of the respective property of that shape.

  2. Add a constructor Triangle to Shape for triangles. The new constructor should take three Float values, the length of the sides of the triangle.

  3. Add cases to area and perimeter for Triangle.

Exercise 5.34. [Keller and Chakravarty] How would you define a data type to represent the different cards of a deck of poker cards? How would you represent a hand of cards?

  Define a function value21 which, given a hand of cards calculates its values according to the 21- (Blackjack) rules: that is, all the cards from 2 to 10 are worth their face value. Jack, Queen, King count as 10. The Ace card is worth 11, but if this would mean the overall value of the hand exceeds 21, it is valued at 1.
Exercise 5.35. The standard functions `head` and `tail`,

\[
\text{head} :: [a] \rightarrow a \\
\text{tail} :: [a] \rightarrow [a]
\]

are partial. a) [Keller and Chakravarty] Implement total variants `safeHead` and `safeTail` by making use of `Maybe` in the function results. b) [Hutton Ex. 4.8.3 with solution] Implement `safeTail` to return an empty list where `tail` returns an error,

- Using a conditional expression
- Using guarded equation
- Using pattern matching.

Exercise 5.36. [Keller and Chakravarty] Write a function `myLength`

\[
\text{myLength} :: [a] \rightarrow \text{Int}
\]

that, given a list \( l \), returns the same result as `length l`. However, implement `myLength` without any explicit pattern matching on lists; instead, use the function `safeTail` from the previous exercise to determine whether you reached the end of the list and to get the list tail in case where the end has not been reached yet.

List comprehension notation

Express one list in terms of other lists

\[
\text{Prelude}> \ [ 2*x \mid x \leftarrow [1,2,3] ] \\
[2,4,6] \\
\text{Prelude}> \ [ (x,y) \mid x \leftarrow [1,2,3], y \leftarrow ['a','b','c'] ] \\
[(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b'),
  (2,'c'),(3,'a'),(3,'b'),(3,'c')] \\
\text{Prelude}> \ [ x \mid x \leftarrow [1..10], x \mod 3 == 1 ] \\
[1,4,7,10]
\]

Exercise 5.37. Use list comprehension notation to complete this function definition to take a list of integers, and return a list containing only the elements of the argument which are divisible by three:

\[
\text{dividesByThree} :: \text{[Int]} \rightarrow \text{[Int]}
\]

\[
\text{dividesByThree} \ \text{xs} = \ [ x \mid x \leftarrow \text{xs} 
\]

Exercise 5.38. Use list comprehension notation to write the function `capVowelsFirst` that takes a list of strings, and return a list containing only the elements of the argument which start with a capital vowel.

5.2.2 Leftist heaps

Trees and heaps

Lovely memories from the simpler days of CS340

- A tree is a structure which can either be
  - Empty, or
  - A node, with a value plus subtrees

In a binary tree, every node has two subtrees

- A heap, generally speaking, is a structure used for finding and deleting minimum elements
Often implemented through an array, or as a binary tree
- The value at any node is no larger than the values at either child

**Tree rank**
The *rank* of a tree node is the length of its right spine

---

**Exercise 5.39.** Write a Haskell data type `BlackWhiteTree` with two constructors
- `Black`, which takes two `BlackWhiteTree` arguments
- `White`, which takes no arguments

Encode each of the above examples (plus the one below) as `BlackWhiteTree` values with names `bw1`, `bw2`, etc.

**Exercise 5.40.** For your `BlackWhiteTree` type of Exercise 5.39, write a function `bwNodeRank` which returns the rank of the top node of a `BlackWhiteTree`.

**The leftist property**
A tree has the *leftist* property when the rank of any left child is at least as big as the rank of its right sibling

---

**Exercise 5.41.** For your `BlackWhiteTree` type of Exercise 5.39, write a function `bwHasLeftist` which returns `True` when given the root node of a tree with the leftist property.

---

33
A leftist heap of floating point values

Let’s design a leftist heap LDH for holding floating-point values

The heap should support these operations:

```haskell
emptyHeap :: LDH
isEmpty :: LDH -> Bool
insert :: Double -> LDH -> LDH
merge :: LDH -> LDH -> LDH
findMin :: LDH -> Double
deleteMin :: LDH -> LDH
```

The last four functions should all preserve both heap ordering and the leftist property

**Standard trick: store the rank**

- To speed comparisons, we store the rank in each node
- To better guarantee properties, we restrict access to the constructors

```haskell
module LeftistDoubleHeap (LDH, emptyHeap, isEmpty, insert, merge, findMin, deleteMin) where

data LDH = EmptyLDH | NodeLDH Int Double LDH LDH

Some helper functions

We will need the rank of nodes

```haskell
leftistRank :: LDH -> Int
leftistRank EmptyLDH = 0
leftistRank (NodeLDH n _ _ _) = n
```

Assemble a `NodeLDH` so that we satisfy the leftist property

```haskell
makeLDH :: Double -> LDH -> LDH -> LDH
makeLDH e h1 h2 = let r1 = leftistRank h1
                   r2 = leftistRank h2
                   in if r1 >= r2
                       then NodeLDH (1+r2) e h1 h2
                       else NodeLDH (1+r1) e h2 h1
```

The easy ones

Returning and testing for an empty tree is straightforward

```haskell
emptyHeap :: LDH
emptyHeap = EmptyLDH

isEmpty :: LDH -> Bool
isEmpty EmptyLDH = True
isEmpty _ = False
```
Merging two heaps

The main decision in merging two trees is picking the smaller of the two top elements to be the new top element

- We merge the right spines in the same way that we can merge sorted lists
- Since the right spine is never longer than the left spine, we are assured of $O(\log n)$ merging
- The `makeLDH` helper assures that the leftist property is upheld

```haskell
merge :: LDH -> LDH -> LDH
merge EmptyLDH h = h
merge h EmptyLDH = h
merge h1@(NodeLDH _ e1 l1 r1) h2@(NodeLDH _ e2 l2 r2) =
  if e1 < e2
    then makeLDH e1 l1 (merge r1 h2)
    else makeLDH e2 l2 (merge h1 r2)
```

Merging example

```haskell
merge (NodeLDH 3 1.1 aa (NodeLDH 2 3.0 bb (NodeLDH 1 5.4 cc EmptyLDH))))
(NodeLDH 2 2.0 dd (NodeLDH 1 4.2 ee EmptyLDH))

==> if 1.1 < 2.0
then makeLDH 1.1 aa (merge (NodeLDH 2 3.0 bb (NodeLDH 1 5.4 cc EmptyLDH)))
(NodeLDH 2 2.0 dd (NodeLDH 1 4.2 ee EmptyLDH)))
else makeLDH 2.0 dd
  (merge (NodeLDH 3 1.1 aa (NodeLDH 2 3.0 bb (NodeLDH 1 5.4 cc EmptyLDH)))
   (NodeLDH 1 4.2 ee EmptyLDH)))

==> makeLDH 1.1 aa (merge (NodeLDH 2 3.0 bb (NodeLDH 1 5.4 cc EmptyLDH)))
(NodeLDH 2 2.0 dd (NodeLDH 1 4.2 ee EmptyLDH)))

==> makeLDH 1.1 aa
  (if 3.0 < 2.0
   then makeLDH 3.0 bb (merge (NodeLDH 1 5.4 cc EmptyLDH))
     (NodeLDH 2 2.0 dd (NodeLDH 1 4.2 ee EmptyLDH)))
   else makeLDH 2.0 dd (merge (NodeLDH 2 3.0 bb (NodeLDH 1 5.4 cc EmptyLDH)))
     (NodeLDH 1 4.2 ee EmptyLDH)))

==> makeLDH 1.1 aa
  (makeLDH 2.0 dd (merge (NodeLDH 2 3.0 bb (NodeLDH 1 5.4 cc EmptyLDH)))
   (NodeLDH 1 4.2 ee EmptyLDH)))

==> makeLDH 1.1 aa (makeLDH 2.0 dd
  (if 3.0 < 4.2
   then (makeLDH 3.0 bb (merge (NodeLDH 1 5.4 cc EmptyLDH))
     (NodeLDH 1 4.2 ee EmptyLDH)))
   else (makeLDH 4.2 bb (merge (NodeLDH 2 3.0 bb (NodeLDH 1 5.4 cc EmptyLDH)))
     (NodeLDH 1 4.2 ee EmptyLDH))))

==> makeLDH 1.1 aa (makeLDH 2.0 dd
  (makeLDH 3.0 bb (merge (NodeLDH 1 5.4 cc EmptyLDH)) (NodeLDH 1 4.2 ee EmptyLDH)))

==> makeLDH 1.1 aa (makeLDH 2.0 dd
  (if 5.4 < 4.2
   then (makeLDH 5.4 cc (merge EmptyLDH (NodeLDH 1 4.2 ee EmptyLDH)))
   else (makeLDH 4.2 ee (merge (NodeLDH 1 5.4 cc EmptyLDH) EmptyLDH))))

==> makeLDH 1.1 aa (makeLDH 2.0 dd (makeLDH 3.0 bb
  (if 5.4 < 4.2
   then (makeLDH 5.4 cc (merge EmptyLDH (NodeLDH 1 4.2 ee EmptyLDH)))
   else (makeLDH 4.2 ee (merge (NodeLDH 1 5.4 cc EmptyLDH) EmptyLDH))))

==> makeLDH 1.1 aa (makeLDH 2.0 dd (makeLDH 3.0 bb (makeLDH 4.2 ee
  (merge (NodeLDH 1 5.4 cc EmptyLDH) EmptyLDH)))

==> makeLDH 1.1 aa (makeLDH 2.0 dd (makeLDH 3.0 bb (makeLDH 4.2 ee
  (NodeLDH 1 5.4 cc EmptyLDH)))
```
**Insertion and deletion can just use heap merging**

```haskell
insert :: Double -> LDH -> LDH
insert e h = merge (NodeLDH 1 e EmptyLDH EmptyLDH) h

findMin :: LDH -> Double
findMin EmptyLDH = error "Reading from empty heap"
findMin (NodeLDH _ e _ _) = e

deleteMin :: LDH -> LDH
deleteMin EmptyLDH = error "Deleting from empty heap"
deleteMin (NodeLDH _ _ h1 h2) = merge h1 h2
```

**Exercise 5.42.** Assemble the module `LeftistDoubleHeap`, and define several example heaps, extracting information from each.

**References**


**5.2.3 Red-black trees**

**Red-black trees**

A *balanced* tree has the same number of elements and the same depth on the left- and right-sides of every node

- Balanced trees guarantee $O(\log n)$ operations in many cases
- True balance can be expensive to maintain, so a number of algorithms allow us to approximate balance more cheaply
- Red-black trees are an approximation to balanced trees
  - Not perfectly balanced, but close enough

Starts with an ordered binary tree

- No duplicate elements, modeling a set

Adds a *color*, red or black, to each node of the tree

- Leaves are considered black

Plus two *invariants* about the structure of the tree:

1. All paths from the root of the tree to an empty leaf must have the same number of black nodes
2. No red node has a red child

**Red-black tree data type**

We’ll design an implementation for red-black trees holding floating-point values

- Could also model the color with a `Bool` field, for example `True` for black and `False` for red

```haskell
module RedBlackDoubleTree (RBDT, emptyTree, isEmpty, member, insert) where

data Color = Red | Black

data RBDT = EmptyRBDT
  | BranchRBDT Color RBDT Double RBDT
```
Exercise 5.43. Write a function \texttt{verifyRBInvariants} which checks that an RBDT value satisfies the invariants that

1. Its numbers come in order
2. No red node has a red child
3. Every path from the root to an empty leaf has the same number of black nodes

Make sure that your function traverses the tree only \textit{once}, and does not re-descend to re-count the black nodes at every branch (there is a hint for this last requirement on page 90).

Basic operations

- Empty trees are straightforward

\begin{verbatim}
emptyTree :: RBDT
emptyTree = EmptyRBDT

isEmpty :: RBDT -> Bool
isEmpty EmptyRBDT = True
isEmpty _ = False
\end{verbatim}

- Checking for membership is just as in any ordered binary tree

\begin{verbatim}
member :: Double -> RBDT -> Bool
member _ EmptyRBDT = False
member e (BranchRBDT _ lt e0 rt) =
  case compare e e0 of
    LT -> member e lt
    EQ -> True
    GT -> member e rt
\end{verbatim}

Top-level insertion

We will adopt the helpful convention that our trees will always have a black root node, even if top-level manipulations end with a red root

\begin{verbatim}
insert e t =
  case helper e t of
    (BranchRBDT _ lt e0 rt) -> BranchRBDT Black lt e0 rt
    _ -> error "Internal error"
    -- Because helper never returns an empty node
\end{verbatim}

Insertion and balancing

- Superficially the \texttt{helper} looks like recursive colorless sorted-tree insert, but does extra work to preserve the invariants

\begin{verbatim}
helper :: Double -> RBDT -> RBDT
\end{verbatim}

- The \texttt{helper} returns a singleton tree when it reaches an empty leaf

\begin{verbatim}
helper e EmptyRBDT = BranchRBDT Red EmptyRBDT e EmptyRBDT
\end{verbatim}
To preserve the number of black nodes on each path, the new node is red

- But this may give us a red node with a red child

\[
\text{helper } e \text{ tt}@\text{BranchRBDT cl lt e0 rt} =
\begin{align*}
\text{if } e &< e0 \\
\text{then } &\text{balance cl (helper e lt) e0 rt} \\
\text{else if } e &> e0 \\
\text{then } &\text{balance cl lt e0 (helper e rt)} \\
\text{else } &\text{tt}
\end{align*}
\]

So we apply a separate `balance` function instead of the `BranchRBDT` constructor to check for violations of the red-red invariant.

When the helper breaks the red-red invariant

`helper` adds a new bottommost node

\[
\begin{align*}
\text{balance Black a x} \\
\text{(BranchRBDT Red (BranchRBDT Red b y c) z d) =} \\
\text{BranchRBDT Red (BranchRBDT Black a x b) y (BranchRBDT Black c z d)}
\end{align*}
\]

**balance** must find the bad pattern

- When using a subtree with a red root which has a red child
- Which can only happen if parent node is black
- Rearrange the tree to restore the invariants
- Push the forbidden red-red pair upwards

**Four cases which balance must find**

\[
\begin{align*}
\text{balance :: Color -> RBDT -> Double -> RBDT -> RBDT} \\
\text{balance Black (BranchRBDT Red (BranchRBDT Red a x b) y c) z d =} \\
\text{BranchRBDT Red (BranchRBDT Black a x b) y (BranchRBDT Black c z d)} \\
\text{balance Black (BranchRBDT Red a x (BranchRBDT Red b y c)) z d =} \\
\text{BranchRBDT Red (BranchRBDT Black a x b) y (BranchRBDT Black c z d)} \\
\text{balance Black a x (BranchRBDT Red (BranchRBDT Red b y c) z d) =} \\
\text{BranchRBDT Red (BranchRBDT Black a x b) y (BranchRBDT Black c z d)} \\
\text{balance Black a x (BranchRBDT Red b y (BranchRBDT Red c z d)) =} \\
\text{BranchRBDT Red (BranchRBDT Black a x b) y (BranchRBDT Black c z d)} \\
\text{balance color left root right = BranchRBDT color left root right}
\end{align*}
\]

- Can you see now the two reasons why we always set the color of the final root node to black?
Exercise 5.44. Assemble the module RedBlackDoubleTree, and define several example trees, extracting information from each. Use your function verifyRBinvariants from Exercise 5.43 to make sure it behaves correctly.

Exercise 5.45. We can optimize this code slightly based on the way helper knows whether its recursive call is in the left or right subtree. Replace balance with two functions balanceLeft and balanceRight, which check for a red-red violation only in the left or right subtree, respectively. Then update helper to call the appropriate replacement for balance. Use your function verifyRBinvariants from Exercise 5.43 to make sure it behaves correctly.

References


5.2.4 Huet’s Zipper

Locations within a tree

Sometimes we need to discuss not just a tree (or other structure) but a particular subtree of the overall structure

• For example, to visually navigate a structure
  – Moving left and right, up and down
  – Possibly editing along the way

• We need to separate one subtree from its context

• The zipper is a technique for implementing this shifting view

• Intuitively, the technique peels up part of a structure, as if turning a glove inside-out when removing it from your hand

We will work on a general tree

data Tree = Branch [Tree]
          | Leaf Double

It is unusual to have no values at branches, but it will simplify the presentation

What is a context?

If we grab on to the link between a branch and one of its child trees, what is the context that we find on the other end from that branch?

• Siblings to its left
• Siblings to its right
  – The siblings are just trees
• More context above
  – Up to the root node
• We can encode the context as a data type

data Context = Root
             | Siblings [Tree] Context [Tree]

• Then a tree with a particular subtree highlighted is just a pair of this context and the subtree

data Location = Loc Context Tree
Moving around within a node

What does it mean to "navigate" from a node to one of its siblings?

For example, if we "move right"

- The first sibling to the right becomes the subtree of interest
- The previous subtree of interest becomes a new sibling to the left

\[
goRight \ (\text{Loc} \ (\text{Siblings} \ ls \ p \ (r:rs)) \ t) = \text{Loc} \ (\text{Siblings} \ (t:ls) \ p \ rs) \ r \\
goRight _ = \text{error} \ "\text{Cannot go right}"
\]

And similarly for moving left

\[
goLeft \ (\text{Loc} \ (\text{Siblings} \ (l:ls) \ p \ rs) \ t) = \text{Loc} \ (\text{Siblings} \ ls \ p \ (t:rs)) \ l \\
goLeft _ = \text{error} \ "\text{Cannot go left}"
\]

- Note that the left siblings are stored with the nearest first
  - So reversed from a left-to-right ordering

Moving up

What about moving up in the tree?

- The previous subtree of interest, plus the siblings of the current context, become part of a new branch node of interest
- The context above the old context becomes the new context

\[
goUp \ (\text{Loc} \ (\text{Siblings} \ ls \ p \ rs) \ t) = \text{Loc} \ p \ (\text{Branch} \ (\text{pushOnto} \ ls \ (t:rs))) \\
\text{where pushOnto} \ [] \ xs = xs \\
\quad \text{pushOnto} \ (y:ys) \ xs = \text{pushOnto} \ ys \ (y:xs) \\
goUp _ = \text{error} \ "\text{Cannot go up}"
\]

Descending into a child node

- Its first child becomes the current subtree
- Its other siblings, plus the old context above, become the new context

\[
\text{goDown} \ (\text{Loc} \ p \ (\text{Branch} \ (t:ts))) = \text{Loc} \ (\text{Siblings} \ [] \ p \ ts) \ t \\
\]

No descending into a leaf, or an empty branch

- We identify with the link between parent and child, and there are no links below a leaf

Adding a subtree

We can make changes to the tree as we navigate it

- Since we reassemble tree and context structure as we go, we do not need to change nonlocal structures

Moving to the left or right is straightforward

\[
\text{insertLeft} \ (\text{Loc} \ (\text{Siblings} \ ls \ p \ rs) \ d) \ t = \text{Loc} \ (\text{Siblings} \ (t:ls) \ p \ rs) \ d \\
\text{insertRight} \ (\text{Loc} \ (\text{Siblings} \ ls \ p \ rs) \ d) \ t = \text{Loc} \ (\text{Siblings} \ ls \ p \ (t:rs)) \ d \\
\]

When we insert into the branch below, the new subtree becomes the current focus

\[
\text{insertBelow} \ (\text{Loc} \ p \ (\text{Branch} \ sibs)) \ t = \text{Loc} \ (\text{Siblings} \ [] \ p \ sibs) \ t
\]
Removing the current subtree

Removing is a little complicated, because if we remove the current subtree then we also need to move

- We need to pick default directions
- Move right if possible, else try left, else try up

\[
\begin{align*}
\text{prune } (\text{Loc } (\text{Siblings } l s p (r:rs)) \_ ) &= \text{Loc } (\text{Siblings } l s p rs) r \\
\text{prune } (\text{Loc } (\text{Siblings } (l:l) s p []) \_ ) &= \text{Loc } (\text{Siblings } l s p []) l \\
\text{prune } (\text{Loc } (\text{Siblings } []) s p []) &= \text{Loc } p (\text{Branch } []) \\
\text{prune } (\text{Loc } \text{Root } \_) &= \text{error } "\text{Cannot prune root node}" \\
\end{align*}
\]

Derivatives

How do we think about zippers for arbitrary data types?

- The intuition comes from the derivatives of calculus
- \(d(u+v)=du+dv\)
- \(d(uv)=udv+vdu\)
- Multiplication is analogous to gathering data together in a tuple or with a constructor
- Addition is analogous to the alternative constructors allowed for a data type
- So the context associated with a pair would be either
  - A regular left element, plus a context to the right; or
  - A context to the left, and a regular right element
- And if a type can have one of two forms, then contexts over that type will also have one of two forms

Exercise 5.46. Extend the general trees and contexts of this section to have a value associated not just with the leaves, but with branches as well.

Exercise 5.47. Develop a notion of contexts for binary trees.

References

5.2.5 Laziness in data structures

Haskell evaluation does as little as possible

What does this function do with its first and third arguments?

secondOfThree _ x _ = x

It is literally true that it does nothing with these arguments — Haskell does not even evaluate them

- You can prove this to yourself with this expression:
  secondOfThree (error "Boom") 1 (error "Boom")
- Returns 1 — and does not throw an error
- This is laziness — not evaluating an argument until it is actually needed

And it’s finely-grained

Whenever a function doesn’t require some parts of an argument, those parts won’t necessarily be evaluated

- secondOfList (_:_x:_ _) = x
  If we apply this function to a list with errors, we won’t trigger these errors if the second element of the list doesn’t have errors
  secondOfList (error "el" : 3 : error "rest of the list")
  This expression returns 3
- Laziness applies not just to function application, but also to data constructors
- Until themselves pattern-matched, data structure components will not be evaluated

Unbounded lists

- We can describe lists which are arbitrarily long
- If we do something that requires the whole list (like displaying it, or foldl), then of course our program will not terminate productively
- But if we use a function like take or takeWhile then we can draw only as much as we need

Specifying an unending list

- Simple cases

  onesForever = 1 : onesForever
twoAndUp = [2..]
fromThreeByFives = [3,8..]
byTens = byTens’ 10 where byTens’ x = x : byTens’ (10+x)

Exercise 5.48. Complete the recursive function genFibs in the style of byTens so that we could use genFibs to define a list corresponding to the Fibonacci numbers.

```haskell
genFibs :: Int -> Int -> [Int]
genFibs n1 n2 = n1 : -- FILL IN HERE
```

The first argument n1 corresponds to the present "first" element of the list of numbers, and the two arguments to the recursive call should lead to subsequent elements.

Prelude> take 6 (genFibs 10 100)
[10,100,110,210,320,530]
Exercise 5.49. Use genFibs from the previous exercise to define the list fibs of Fibonacci numbers.

Prelude> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]

Exercises ??-??
Write recursive function genFibs in the style of byTens so that we could use genFibs to define a list corresponding to the Fibonacci numbers, and use it to define the standard list of all Fibonacci numbers starting 0, 1, 1, and so on.

Laziness in data structures

The Sieve of Eratosthenes

• Write out the numbers we’re interested in testing for primality

1  2  3  4  5  6  7  8  9  10  
11 12 13 14 15 16 17 18 19 20  
21 22 23 24 25 26 27 28 29 30  
31 32 33 34 35 36 37 38 39 40  
41 42 43 44 45 46 47 48 49 50  
51 52 53 54 55 56 57 58 59 60  
61 62 63 64 65 66 67 68 69 70  
71 72 73 74 75 76 77 78 79 80  
81 82 83 84 85 86 87 88 89 90  
91 92 93 94 95 96 97 98 99 100

The Sieve of Eratosthenes

• Write out the numbers we’re interested in testing for primality
• 1 is not a prime, so scratch it out

4  2  3  4  5  6  7  8  9  10  
11 12 13 14 15 16 17 18 19 20  
21 22 23 24 25 26 27 28 29 30  
31 32 33 34 35 36 37 38 39 40  
41 42 43 44 45 46 47 48 49 50  
51 52 53 54 55 56 57 58 59 60  
61 62 63 64 65 66 67 68 69 70  
71 72 73 74 75 76 77 78 79 80  
81 82 83 84 85 86 87 88 89 90  
91 92 93 94 95 96 97 98 99 100

The Sieve of Eratosthenes

• Write out the numbers we’re interested in testing for primality
• 1 is not a prime, so scratch it out
• Look at the lowest unmarked number — mark it as prime
The Sieve of Eratosthenes

- Write out the numbers we’re interested in testing for primality
- 1 is not a prime, so scratch it out
- Look at the lowest unmarked number — mark it as prime
- But strike out its multiples — they’re definitely not prime

```
4  2  3  4  5  6  7  8  9  10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100
```

The Sieve of Eratosthenes

- Write out the numbers we’re interested in testing for primality
- 1 is not a prime, so scratch it out
- Look at the lowest unmarked number — mark it as prime
- But strike out its multiples — they’re definitely not prime
- And so on with the new lowest unmarked number

```
4  2  3  4  5  6  7  8  9  10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100
```
The Sieve of Eratosthenes

• Write out the numbers we’re interested in testing for primality
• 1 is not a prime, so scratch it out
• Look at the lowest unmarked number — mark it as prime
• But strike out its multiples — they’re definitely not prime
• And so on with the new lowest unmarked number, and so on

|   | 2  | 3  | 5  | 7  | 11 | 13 | 17 | 19 | 23 | 29 | 31 | 37 | 41 | 43 | 47 | 53 | 59 | 61 | 67 | 71 | 73 | 79 | 83 | 89 | 97 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 4  | 6  | 8  | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 26 | 28 | 30 | 32 | 34 | 36 | 38 | 40 | 42 | 44 | 46 | 48 | 50 |
|   | 44 | 46 | 48 | 50 | 52 | 54 | 56 | 58 | 60 | 62 | 64 | 66 | 68 | 70 | 72 | 74 | 76 | 78 | 80 | 82 | 84 | 86 | 88 | 90 |
|   | 92 | 94 | 96 | 98 | 100| 102| 104| 106| 108| 110| 112| 114| 116| 118| 120| 122| 124| 126| 128| 130| 132| 134| 136|

The Sieve of Eratosthenes

• Write out the numbers we’re interested in testing for primality
• 1 is not a prime, so scratch it out
• Look at the lowest unmarked number — mark it as prime
• But strike out its multiples — they’re definitely not prime
• And so on with the new lowest unmarked number, and so on, and so on

|   | 2  | 3  | 5  | 7  | 11 | 13 | 17 | 19 | 23 | 29 | 31 | 37 | 41 | 43 | 47 | 53 | 59 | 61 | 67 | 71 | 73 | 79 | 83 | 89 | 97 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 4  | 6  | 8  | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 26 | 28 | 30 | 32 | 34 | 36 | 38 | 40 | 42 | 44 | 46 | 48 | 50 |
|   | 44 | 46 | 48 | 50 | 52 | 54 | 56 | 58 | 60 | 62 | 64 | 66 | 68 | 70 | 72 | 74 | 76 | 78 | 80 | 82 | 84 | 86 | 88 | 90 |
|   | 92 | 94 | 96 | 98 | 100| 102| 104| 106| 108| 110| 112| 114| 116| 118| 120| 122| 124| 126| 128| 130| 132| 134| 136|

The Sieve of Eratosthenes

• Write out the numbers we’re interested in testing for primality
• 1 is not a prime, so scratch it out
• Look at the lowest unmarked number — mark it as prime
• But strike out its multiples — they’re definitely not prime
• And so on with the new lowest unmarked number, and so on, and so on, and so on
• So let’s code that up

**Fibonacci numbers vs. primes**

In all of the by-tems, Fibonacci numbers and prime numbers examples, we generate the list one element at a time

• Of course — as for any linked list!

• More to the point: we write code responsible for generating only one "next" element, and leave the rest of the elements to subsequent evaluations

• In *byTens*, the next list element was just a function argument
  – Set up future list elements by changing the argument to the recursive call

• In *genFibs*, same idea, just need two arguments

• For prime numbers, the helper function argument is the list of candidates for prime numbers under Eratosthenes’s algorithm
  – On each pass, we recognize the first of the candidates as prime
  – Filter its multiples from the list of candidates passed to the recursive call

**Coding up the sieve**

• One pass of the sieving algorithm:
  – Accept the first element as prime
  – Remove all multiples of the first element from the rest of the list
  – Sieve what’s left

```haskell
sieve (x:xs) = x : sieve (filter (\z -> z `mod` x > 0) xs)
```

• Then the list of all prime numbers is just

```haskell
primes = sieve [2..]
```

• So long as we don’t try to find the last element of the list!

```haskell
*Prelude> take 30 primes

*Prelude> head (filter (> 10000) primes)
10007
```
So how do the various list functions work with nonterminating lists?

- Recall foldl and foldr,

\[
\begin{align*}
\text{foldl } f \ z \ [] & = z \\
\text{foldl } f \ z \ (x:xs) & = \text{foldl } f \ (f \ z \ x) \ xs \\
\text{foldr } f \ z \ [] & = z \\
\text{foldr } f \ z \ (x:xs) & = f \ x \ (\text{foldr } f \ z \ xs)
\end{align*}
\]

- What if we want to apply a folding function to a list like primes?
  - (Say, with an \( f \) that constructs some other data structure)
- foldl will try to deconstruct the list all the way to the end before returning anything else
- If \( f \) returns some data constructor, then foldr can avoid trying to traverse the whole list

5.3 Parametric polymorphism

5.3.1 Polymorphic functions

What type do these functions have?

- What type does \( f \) have?

\[
f :: \text{Int} \to \text{Int} \\
f x = x
\]

  - \text{Int} -> \text{Int}, of course

- What type does \( g \) have?

\[
g x = x
\]

  - It could have type \( \text{Int} \to \text{Int} \)
  * But it could have type \( \text{String} \to \text{String} \)
  * Or \( \text{[Int]} \to \text{[Int]} \)
  * Or \( \text{Bool} \to \text{Bool} \)
  * Or \( \text{Double} \to \text{Double} \)
  * Or \( \text{(Double, Int, String)} \to \text{(Double, Int, String)} \)
  - Is there any type Haskell type \( a \) for which \( g \) could not have type \( a \to a \)?
    * No! Absolutely any \( a \) works

Polymorphic functions

We can write (or Haskell can deduce) \textit{polymorphic} function types with unspecified parts to them

- Just as Java can have generic methods and classes
  - In functional languages, you’ll see the phenomenon referred to as \textit{polymorphic} more often than \textit{generic}

- Polymorphic functions

\[
a \to a \\
\text{(([Char], a) -> b) -> [Char] -> (a -> b)}
\]
• What are these a, b, c?
  – *Type variables*
  – Quantified outermost, so t \rightarrow t means (\forall t.(t \rightarrow t))
  – Write type variables with an initial lower-case letter

• What about a function of type \textbf{Int} \rightarrow a — could such a function exist?
  – Yes, but it is not very interesting
    
    ```haskell
    boring :: Int -> a
    boring z = error "How dull, always an error"
    ```
  – In a certain sense, we cannot expect to get more information ("Any type! Any at all!") from a function than we put in ("Just an Int, nothing else")

**Benefits of polymorphic types**

• Detect and prohibit 	extit{further} nonsensical operations
  
• 	extit{Finer-grained} documentation which is automatically checked for correctness

• Reduce code duplication

• More easily distinguish bugs in using a library from bugs within a library

**5.3.2 Polymorphic data types**

Data types can be polymorphic too

• You may already have noticed that functions on lists can be polymorphic

  ```haskell
  reverse :: [a] -> [a]
  reverse xs = rev' [] xs
  where rev' acc [] = acc
       rev' acc (x:xs) = rev' (x:acc) xs
  ```

  – Lists are a \textit{polymorphic type}

  – So what is the type of the empty list (outside of a context which restricts it)?

• Your types can be polymorphic too

  ```haskell
  data BinaryTree a = Branch (BinaryTree a) (BinaryTree a)
                    | Leaf a
  ```

  ```haskell
  binaryTreeMap f (Leaf x) = Leaf (f x)
  binaryTreeMap f (Branch t1 t2)
      = Branch (binaryTreeMap f t1) (binaryTreeMap f t2)
  ```

  – Here we distinguish the \textit{type constructor} BinaryTree from types like BinaryTree Int, BinaryTree Float, or BinaryTree a.

  – By itself, BinaryTree is not a type
One second thought

How does search in this binary tree work?

data BinaryTree a = Branch (BinaryTree a) (BinaryTree a)
  | Leaf a

binaryTreeMap f (Leaf x) = Leaf (f x)
binaryTreeMap f (Branch t1 t2)
  = Branch (binaryTreeMap f t1) (binaryTreeMap f t2)

• With abstracted types like a, we cannot assume things about them, like whether they are an Int or String
• We also cannot assume that they support comparison!
• To define binary trees as we would really expect, we will need other tool from Haskell’s toolkit — another day

Collections classes

In many languages, collections classes are the best-known use case of polymorphic types

• Set<A>, Map<A,B>, List<A>

• Avoid casts from versions of the collections library which just use Object as the type of all contents

5.3.3 Further type definitions

There are two more ways of defining a new type in Haskell

• One way is a type synonym, keeping equivalence
• Another way does not preserve interchangeability

Equivalent synonyms via type
type defines an abbreviation for our convenience

• The prelude defines String this way:
  
  type String = [Char]

  We can use String in our type or instance declarations, but ghci can’t always echo the name back to us
• We can give type variables for polymorphic types as well:
  
  type ListOfTuplesWithInt a = [(a,Int)]

  tupWith1 :: ListOfTuplesWithInt Bool
  tupWith1 = [(True, 3), (False, 4), (False, 5)]

Distinct synonyms via newtype

newtype defines a type synonym which is not interchangeable with the original

• newtype DifferentTuple = DiffTup (Int, String)
• We use it as if it had been declared with data
  
  intFromDiffTup (DiffTup (n,__)) = n

• But there are important differences with data
– There can be only one constructor form
– That constructor can have only one value associated with it
  * Which is why we have a tuple here, and not two separate values
– The overhead of distinguishing the different data constructors can be compiled away

• The usual style with newtype is to give the type and constructor the same name

```haskell
newtype DifferentTuple = DifferentTuple (Int, String)
intFromDiffTup (DifferentTuple (n,_)) = n
```

• Optionally, we can also declare an accessor function at the same time

```haskell
newtype DifferentTuple = DifferentTuple { getDiffTup :: (Int, String) }
```

• And type variables are allowed

```haskell
newtype ZZ a = ZZ { getZzA :: a }
```

**Using newtype for alternative instances**

One use of newtype is to associate different instance declarations with a type.

```haskell
newtype WordInt = WI Int
```

```haskell
instance Show WordInt where
    show (WI 0) = "zero"
    show (WI 1) = "one"
    show (WI 2) = "two"
    show (WI 3) = "three"
    show (WI 4) = "four"
    show (WI 5) = "five"
    show (WI 6) = "six"
    show (WI 7) = "seven"
    show (WI 8) = "eight"
    show (WI 9) = "nine"
```

**Exercise 5.50.** Consider the declaration:

```haskell
newtype NewTypeExampleWithInt = { theInt :: Int }
```

What type does Haskell tell you that the function theInt has? Create some NewTypeExampleWithInt values; how do you use them with theInt? What results do you get?

### 5.4 Higher-order functions

#### 5.4.1 Functions as values

**First-class citizens**

In Haskell, we say that functions are *first-class citizens* of the language

What does this mean?

• We can write them as standalone constants *without necessarily binding them to a name* — just as for any other value
• We can pass them to another function, so that a formal parameter has function type; or bind them to a local name — just as with any other value
• We can return one function from another function — just as any with other value
• We can use anonymous function constants, or names locally bound to a function, in just the same way as names globally bound to a function — just as any with other value

A function is a mapping from arguments to results

We can describe that mapping as a lambda expression

\arg \rightarrow \text{result}

The backslash abbreviates the Greek letter λ.

\x \rightarrow x+1
\ss \rightarrow "Pre" + \ss

There can be multiple parameters

\x \ y \rightarrow 2\times x+y

Sometimes called a lambda abstraction

• Abstracting the names over the body of the result

Scope

Functions can refer to names outside the scope of their arguments

\a \rightarrow \sin (2\times a + \pi/2)

This is valid even for local names

let x = 5\pi in \z \rightarrow \sin (x + z/2)

Note that Java has a limited facility for \lambda abstractions

(int x, String y) \rightarrow x + y.length()

• Since Java 8
• Understood by Java as an anonymous class implementing a single-method interface
• Has stricter rules for shadowing, using out-of-scope names

Functions as arguments

We can pass functions as arguments to other functions

callWithThree :: (Int->Int) \rightarrow \text{Int}
callWithThree f = f 3
double x = x+x
triple x = 3\times x

• callWithThree double returns 6
• callWithThree triple returns 9

The functions can be polymorphic
callWithThree :: (Int->a) -> a
callWithThree f = f 3
howManyZs 0 = ""
howManyZs n = "Z" : howManyZs (n-1)

- callWithThree double still returns 6
- callWithThree triple still returns 9
- callWithThree howManyZs returns "ZZZ"

Functions as results
Functions can also return another function as a result

whichIncrementer :: Bool -> (Int -> Int)
whichIncrementer x = if x then (\x -> x+1) else (\y -> y+2)

- So (whichIncrementer True) 10 returns 11
- (whichIncrementer False) 20 returns 22

How we write types

- Recall how we write the types for multi-argument functions

    myFormula :: Int -> Int -> Int
    myFormula m n = 20*m + n

- In particular, we do not write the type like this:

    (Int, Int) -> Int

- The notation suggests that there are several functions involved
  - There are!
  - Functions in this form are said to be curried

Currying

- Let’s say we need a function of type Int->Int
- We can give myFormula one of its arguments now, and (presumably) others later

    myFormula :: Int -> Int -> Int
    myFormula m n = 20*m + n
    let needsOneInt :: Int -> Int
        needsOneInt = myFormula 100
        in needsOneInt 5
    - Returns 2005

- Since myFormula is curried, we can partially apply it
It’s not a cooking reference

Haskell Brooks Curry

- Born 1900 in Massachusetts, majored in mathematics at Harvard, then returned for a master’s in physics
- During his master’s work, learned of the then-ongoing work of Whitehead and Russell to ground mathematics in formal logic
- Returned to mathematics for his Ph.D., focusing on the new combinatory logic of Schoenfinkel
- Spent most of his career at Pennsylvania State College
- Retired 1970, died 1982

Combinatory logic and its impact

- Similar in scope to Church’s lambda calculus
- Wrote and taught extensively about combinatory logic and the logical foundations of mathematics
- Memorialized with the Curry-Howard correspondence, Curry’s paradox, and three programming languages named after him

Operator sectioning

Partial application also allies to binary operators

- In this context, also known as sectioning
- Requires parentheses

```
gg = (2 +)
hh = (* 5)
kk = (1.0 /)
```

So

- `gg 10` reduces to `2+10`
- `hh 10` reduces to `10*5`
- `kk` takes the reciprocal if its argument

But note that `(- 1)` is not a function value, it is a number one less than zero

- Use `(-) 1` to section subtraction
5.4.2 Patterns of recursion over lists: filter, map, fold

Finding patterns

sumOneTo :: Int -> Int
sumOneTo x | x>0 = x + sumOneTo (x-1)
sumOneTo _ = 0

prodOneTo :: Int -> Int
prodOneTo x | x>0 = x * prodOneTo (x-1)
prodOneTo _ = 1

Three patterns of behavior on lists

Filtering Derive one list from another by selecting some of its arguments

Mapping Transform one list to another by transforming its individual elements

Folding Combine the elements of list with each other to produce a result

Filtering  
Given a list of integers, return a list of the even integers in the argument list

• justEvens :: [Int] -> [Int]
  justEvens [] = []
  justEvens (x:xs) | x `mod` 2 == 0 = x : justEvens xs
  justEvens (_:xs) = justEvens xs

Given a string, return a string of only the lower-case letters in the original string (assuming Data.Char imported for isLower)

• justLower :: String -> String
  justLower [] = []
  justLower (x:xs) | isLower x = x : justLower xs
  justLower (_:xs) = justLower xs

Given a list of lists, return the list containing only the lists of length 2 or more from the argument

• justLengthy :: [[a]] -> [[a]]
  justLengthy [] = []
  justLengthy (x:xs) | length x > 1 = x : justLengthy xs
  justLengthy (_:xs) = justLengthy xs

These functions operationally differ only in the tested condition

The **filter** function

  We can pass a **predicate** as an extra parameter

filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs) | p x = x : filter p xs
filter p (_:xs) = filter p xs
Exercise 5.51. Define justEvens, justLower and justLengthy using filter instead of explicit recursion.

Mapping one function to another
Given a list of integers, return a list with the argument values multiplied by 11

- `elevenfold :: [Int] -> [Int]`
  `elevenfold [] = []`
  `elevenfold (x:xs) = (11*x) : elevenfold xs`

Given a string, return that string cast to lower-case (assuming `Data.Char` imported for `toLowerCase`)

- `allToLower :: String -> String`
  `allToLower [] = []`
  `allToLower (x:xs) = toLower x : allToLower xs`

Given a list of lists, return the list containing the reverses of the original argument’s lists

- `reverseAll :: [[a]] -> [[a]]`
  `reverseAll [] = []`
  `reverseAll (x:xs) = reverse x : reverseAll xs`

These functions operationally differ only in the operation applied to each element

The filter function
We can pass the transforming function as an extra parameter

\[
\text{map} :: (a -> b) -> [a] -> [b]
\]
\[
\text{map} \ _ \ [] = []
\]
\[
\text{map} \ f \ (x:xs) = f \ x \ : \ \text{map} \ f \ xs
\]

Exercise 5.52. Define elevenfold, allToLower and reverseAll using map instead of explicit recursion.

Combining the elements of a list together
Given a list of integers, return the result of adding the elements together

- `sumTogether :: [Int] -> [Int]`
  `sumTogether [] = 0`
  `sumTogether (x:xs) = x + sumTogether xs`

Given a list of lists, return the concatenation of all of these lists together (using ++ and not worrying too much about efficiency)

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

These functions operationally differ in two places

- The value to which we map the empty list
- The way we combine one element with the result of combining together the rest of the elements

55
A fold function
We can pass the base value and combining function as two extra parameters

\[
\text{fold} :: (a \to b \to b) \to b \to [a] \to b
\]

\[
\text{fold} \_ \_ z [] = z
\]

\[
\text{fold} f z (x:\text{xs}) = f x (\text{fold} f z \text{xs})
\]

- fold is often referred to as reduce

**Exercise 5.53.** Define sumTogether and concatTogether using fold instead of explicit recursion.

**Definitely a trap**
Here is one of those simple questions which seems like something out of primary school but which is probably a trap:

What is 10-3-2-1?

Then what is \( \text{fold} (-) 0 [10, 3, 2, 1] \)?

- Remember that \((-)\) is the sectioned version of subtraction
- It’s 8! It was a trap!

**Two folds**
The trap is that we implicitly defined a certain associativity in our first try at fold — and it happened to be right-associative

- Often right-associativity is what we need
- But in the big picture, we do need the choice of associativity
- Haskell renames our fold as foldr

\[
\text{foldr} :: (a \to b \to b) \to b \to [a] \to b
\]

\[
\text{foldr} \_ \_ z [] = z
\]

\[
\text{foldr} f z (x:\text{xs}) = f x (\text{foldr} f z \text{xs})
\]

- There is also foldl

\[
\text{foldl} :: (b \to a \to b) \to b \to [a] \to b
\]

\[
\text{foldl} \_ \_ z [] = z
\]

\[
\text{foldl} f z (x:\text{xs}) = \text{foldl} f (f z x) \text{xs}
\]

- Caveat: these are not the same types that you would see if you asked ghci

:t foldr

so we will revisit these signatures later!
Exercise 5.54. [Keller and Chakravarty] Rewrite the definition of \texttt{mapInts}

\begin{verbatim}
mapInts :: (Int -> Int) -> [Int] -> [Int]
mapInts f [] = []
mapInts f (x : xs) = f x : map f xs
\end{verbatim}

to use case notation. That is, complete the following definition

\begin{verbatim}
mapInts :: (Int -> Int) -> [Int] -> [Int]
mapInts f xs = case xs of
...  
\end{verbatim}

Exercise 5.55. [Keller and Chakravarty] The \texttt{map} function is just a special case of \texttt{foldr}. Can you rewrite the \texttt{map} definition in terms of \texttt{foldr}? Complete the following definition:

\begin{verbatim}
map :: (a -> b) -> [a] -> [b]
map f = foldr ...  
\end{verbatim}

Exercise 5.56. Use a fold function to implement the exclusive-or function \texttt{xor} of type \([\text{Bool}] \rightarrow \text{Bool}\), which returns \texttt{True} when there is exactly an odd number of \texttt{True} values in the list.

Exercise 5.57. Use a fold function to concatenate a list of lists together into a single list,

\texttt{concatAll :: [[t]] \rightarrow [t]}

Exercise 5.58. Use \texttt{filter} to write a function that removes the vowels from a string.

Exercise 5.59. Redefine \texttt{length} and \texttt{reverse} using the fold functions.

Exercise 5.60. For all of the functions with fold, which are more efficient with \texttt{foldr}, and which are more efficient with \texttt{foldl}?

Exercise 5.61. [Keller and Chakravarty] Rewrite the definition of \texttt{map}

\begin{verbatim}
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x : xs) = f x : map f xs
\end{verbatim}

to use case notation. That is, complete the following definition

\begin{verbatim}
map f xs = case xs of
...  
\end{verbatim}

Exercise 5.62. Write \texttt{intTreeFoldl} and \texttt{intTreeFoldr}, folding functions on integer binary trees,

\begin{verbatim}
data IntTree = Branch IntTree IntTree  
| Leaf Int
\end{verbatim}

The functions should have signatures

\begin{verbatim}
intTreeFoldl :: (t -> Int -> t) -> t -> IntTree -> t
intTreeFoldr :: (Int -> t -> t) -> t -> IntTree -> t
\end{verbatim}

and should apply the operations and default with the given associativity of the values in the leaves. For example,

\begin{verbatim}
*Main> let t1 = (Branch (Branch (Leaf 10) (Leaf 3))
(Branch (Leaf 2) (Leaf 1)))
*Main> intTreeFoldl (\x y -> x-y) 0 t1
-16
*Main> intTreeFoldr (\x y -> x-y) 0 t1
8
\end{verbatim}
Exercise 5.63. Write `binaryTreeFoldl` and `binaryTreeFoldr`, folding functions on general binary trees,

```
data BinaryTree a = Branch (BinaryTree a) (BinaryTree a)  
    | Leaf a
```

Exercise 5.64. Write `binaryTreeFilter`, of type `(a -> Bool) -> (BinaryTree a) -> Maybe (BinaryTree a)`. It must be `Maybe`, since we have no constructor for an empty tree.

References
The material of this section is standard in functional language textbooks and tutorials. A classic paper which carries the idea (much) further is


And then further still,


5.5 "Ad-hoc" polymorphism and type classes

5.5.1 Type classes

Almost everything
Some functions aspire to be polymorphic, but cannot quite make it

- Many values can be ordered, but how do we compare pairs? Or lists?
- Many values can be tested for equality, but what about functions?
- Many values can be converted to strings for display, but again, what about functions? What about our complicated data types which we simply do not need to represent as a string?

And the types do matter!
In these examples, unlike in the parametric polymorphic functions we saw earlier, the actual quantified type really does matter

- The way we compare two integers really is different than the way we compare two floating-point values
- So the actual type really does matter, all the way down to the machine level

Classes of types
Haskell lets us express these quantifications by using *type classes*

- Operations (the functions for comparison, formatting, etc) are associated with a type class
- Every type which is a member of a class *must* implement all of the operations associated with that class
- Types are *individually* declared to be members of a class
Declaring classes

For example:

```haskell
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

- Must give type signatures, since we’re specifying operations without (necessarily) giving an implementation
- When we ask Haskell what the types of these functions are, it tells us *explicitly* about the type constraint
  - `(==) :: Eq a => a -> a -> Bool`
  - The quantification is not universal, but limited to types of the particular class
  - We can write these constraints too, when giving an explicit signature of a function we write

Default implementations of functions

We can give *default* implementations of some (or all) of the functions associated with a class

```haskell
class Eq a where
    (==) :: a -> a -> Bool
    (=/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

- Notice here that we define the two operations in terms of each other!
- So the instance declaration must define at least one of the two operations — otherwise these definitions make no sense

**Exercise 5.65.** Adapt your module *LeftistDoubleHeap* from Exercise 5.44 as simply *LeftistHeap*, with type `LH` polymorphic in the type of element which it contains. Since `merge` makes a comparison on elements of the contained type, most of the exported functions will need an `Ord a` constraint,

```haskell
emptyHeap :: Ord a => LH a
isEmpty :: LH a -> Bool
insert :: Ord a => a -> LH a -> LH a
merge :: Ord a => LH a -> LH a -> LH a
findMin :: Ord a => LH a -> a
deleteMin :: Ord a => LH a -> LH a
```

**Exercise 5.66.** Adapt your module *RedBlackDoubleTree* from Exercise 5.42 as simply *RedBlackTree*, with type `RBT` polymorphic in the type of element which it contains. Since `helper` and `member` make a comparison on elements of the contained type, most of the exported functions will need an `Ord a` constraint,

```haskell
emptyTree :: Ord a => RBT a
isEmpty :: RBT a -> Bool
member :: Ord a => a -> RBT a -> Bool
insert :: Ord a => a -> RBT a -> RBT a
helper :: Ord a => a -> RBT a -> RBT a
balance :: Color -> RBT a -> a -> RBT a -> RBT a
```
Exercise 5.67. [Keller and Chakravarty] Implement a function `deleteSorted`,

\[
deleteSorted :: \text{Ord} \ a \Rightarrow \ a \rightarrow \ [a] \rightarrow \ [a]
\]

which removes a value passed as first argument from a sorted list given as the second argument. If the value does not occur in the list, the list is returned unchanged. Exploit the fact that the list is sorted: if an element is not present in the list, stop the search as early as possible.

Exercise 5.68. Change the declaration of instance `Show MyComplex` so that the real coefficient is not shown when it is zero and the complex coefficient is non-zero.

Exercise 5.69. Complete the declaration of instance `Show WordInt` to print any integer as words.

Another example - complex numbers

There’s a built-in class of complex numbers, but let’s make our own

data MyComplex = MyComplex Double Double

instance Show MyComplex where
    show (MyComplex x iy) = case compare iy 0 of
        GT -> show x ++ "+" ++ show iy ++ "i"
        EQ -> show x
        LT -> show x ++ "-" ++ show (-iy) ++ "i"

instance Num MyComplex where
    (MyComplex x iy) + (MyComplex x' iy') = MyComplex (x + x') (iy + iy')
    (MyComplex x iy) - (MyComplex x' iy') = MyComplex (x - x') (iy - iy')
    (MyComplex x iy) * (MyComplex x' iy') = MyComplex (x*x' - iy*iy')
                             (x*iy' + x*iy')
    abs (MyComplex x iy) = MyComplex (sqrt (x*x + iy*iy)) 0
    signum (num@(MyComplex x iy)) = let (MyComplex a _) = abs num
                                          in MyComplex (x/a) (iy/a)
    fromInteger n = MyComplex (fromInteger n) 0.0

More complex operations

instance Fractional MyComplex where
    (MyComplex x iy) / (MyComplex x' iy')
        = let denom = x'*x' + iy'*iy'
            in MyComplex ((x*x' + iy*iy')/denom) ((x'*iy - x*iy')/denom)
    recip (MyComplex x' iy')
        = let denom = x'*x' + iy'*iy'
            in MyComplex (x'/denom) (-iy'/denom)
    fromRational n = MyComplex (fromRational n) 0.0

• Could define trigonometric etc. operations for `Floating`.

Instances of a polymorphic datatype

Two ways:

1. Declare the full type to be an instance, possibly with constraints on the type variables

2. Declare the type constructor to be an instance of class on higher kinds
Exercise 5.70. Adapt your module `LeftistHeap` from Exercise 5.65 to separate the functions into a class definition of heaps and heap operations, with type `LH` being one instance of the class. Use the approach of higher-kind ed classes so that heaps are polymorphic.

Exercise 5.71. Adapt your module `RedBlackTree` from Exercise 5.66 to separate the functions into a class definition of trees and tree operations, with type `RBT` being one instance of the class. Use the approach of higher-kind ed classes so that trees are polymorphic.

Constraints on an instance declaration

```haskell
instance Eq a => Eq (BinaryTree a) where
  (Leaf x) == (Leaf y) = (x==y)
  (Branch xs1 xs2) == (Branch ys1 ys2) =
  xs1 == ys1 && xs2 == ys2
_ == _ = False

instance Show a => Show (BinaryTree a) where
  show (Leaf x) = "(Leaf " ++ show x ++ ")"
  show (Branch xs1 xs2)
    = "(Branch " ++ show xs1 ++ ", " ++ show xs2 ++ ")"
```

- Like constraints on a type signature

5.5.2 Type classes and software architecture

Remember the `Shape` datatype from Exercise 5.33

```haskell
data Shape = Circle Float
            | Rectangle Float Float
```

You added a `Triangle` constructor

- What else did you have to change when you added that constructor?
  - Immediately, the `perimeter` function gave warnings
  - There was now a possible form of `Shape` which the function could not address!

What if we were in deeper than that?

`Shape` was a small example for a learning exercise, but sometimes we must change software which is larger

- New features added to old systems
- A new case introduced under late-revised software requirements
- Impacts of bug reports, shifted priorities, etc.
- A type like `Shape` might be pattern-matched in many functions across many files
- If many function suddenly fail to compile, or even just each generate verbose warnings, further progress can be slow

Type classes offer an alternative structure

- Instead of a `data` type, define a class
- Instead of constructors, define an instance type
Instead of a data type, define a class

From

data Shape = ...
-- and functions perimeter, area follow
to

class Shape a where
    perimeter :: a -> Float
    area :: a -> Float

Instead of constructors, define an instance type

From

data Shape = Circle Float | Rectangle Float Float

perimeter (Circle radius) = 2 * pi * radius
area (Circle radius) = pi * radius * radius
to

data Circle = Circle Float
instance Shape Circle where
    perimeter (Circle radius) = 2 * pi * radius
    area (Circle radius) = pi * radius * radius

data Rectangle = Rectangle Float Float
instance Shape Rectangle where
    perimeter (Rectangle l w) = 2 * (l + w)
    area (Rectangle l w) = l * w

Exercise 5.72. Write a type Triangle (specified by the three sides of the triangle, as in Exercise 5.33) which is also in class Shape.

Pros and cons

If we use a type with several constructors

• Adding a new function is straightforward
• But adding a new constructor can be rough

If we use a class with several instance types

• Adding a new function can be rough
• But adding a new constructor is straightforward

This is an instance of the Expression Problem

• An old challenge for language designers, identified in print in 1975
• How can a language support both adding datatype cases and adding functions on the datatype
  – In a type-safe way, and
  – Without recompiling existing code?
• We will look at two other solutions to the Expression Problem when we look at object-oriented languages
References


5.5.3 A class of sorting trees

This tree and that tree

Recall our red-black trees of Section 5.2.3:

```haskell
data Color = Red | Black
data RBDT = EmptyRBDT
          | BranchRBDT Color RBDT Double RBDT
```

with functions `emptyTree`, `isEmpty`, `member`, `insert`

Plain-old binary trees

It is easy to imagine a library of red-black trees coexisting with other tree implementations, and even just plain-old binary trees:

```haskell
data BinaryTree = EmptyBT | LeafBT BinaryTree Double BinaryTree
```

• Or even a *non-sorted* tree version!

```haskell
data NaiveBinaryTree =
          EmptyNBT | LeafNBT NaiveBinaryTree Double NaiveBinaryTree
```

Again both with functions `emptyTree`, `isEmpty`, `member`, `insert`

A tree class

We can describe a class of trees to which all of these implementations adhere:

```haskell
class SearchTree t where
  emptyTree :: t
  isEmpty :: t -> Bool
  member :: Double -> t -> Bool
  insert :: Double -> t -> t
```

Simple trees into the class

```haskell
instance SearchTree BinaryTree where
  emptyTree = EmptyBT

  isEmpty EmptyBT = True
  isEmpty (LeafBT _ _ _) = False
```

and so on

• The same functions, just as part of an `instance` declaration instead of top-level
Red-black trees into the class
And likewise for the red-black trees:

```haskell
instance SearchTree RedBlackTree where
    emptyTree = EmptyRBT
    isEmpty EmptyRBT = True
    isEmpty _ = False
    member EmptyRBT _ = False
    member e (NodeRBT _ lt e0 rt) = case compare e e0 of
        LT -> member e lt
        EQ -> True
        GT -> member e rt
```

What about polymorphism?
What we’d really like is a polymorphic datatype for our trees

• Not just for Double values, but for any types (consistently within one tree!)

• This is a little harder to combine with classes
  – We need a type constraint to apply to the content type of the tree, but the operations to apply to the tree itself
  – The right solution is to allow classes over type constructors, and not just types
  – We will come back to this problem when we look at kinds

5.5.4 Automatically deriving instances of the built-in classes

Automatic instances
As part of a data definition, we can automate instance declarations for several of the built-in classes

• Often saves us from boilerplate code

• Sacrifices flexibility — we get one kind of instance implementation automatically
  – Sadly, Haskell still cannot read our minds!

Equality

data Shape = Circle Float
    | Rectangle Float Float
    deriving (Eq)

Two Shape values will be equal if

• They have the same constructor

• The respective fields are also equal

• If we want to disregard some fields, we must write our own instance declaration!

Ordering

data Shape = Circle Float
    | Rectangle Float Float
    deriving (Ord)

Imposes a total order on Shape values

• All circles are less than all rectangles!
Convertability to text

data Shape = Circle Float
    | Rectangle Float Float
deriving (Read, Show)

Imposes a total order on Shape values

• All circles are less than all rectangles!

5.5.5 Higher-kinded class type variables

Class Functor — things we can map over

• Generalizes the map function on lists

• From standard prelude:

    class Functor f where
        fmap :: (a -> b) -> f a -> f b

• f is a type constructor, like BinaryTree

   *Main> :kind Int
   Int :: *
   *Main> :kind BinaryTree
   BinaryTree :: * -> *
   *Main> :kind Either
   Either :: * -> * -> *
   *Main> :kind (->)
   (->) :: * -> * -> *

• So Functor is a class of higher-kinded objects, of type constructors, instead of types

BinaryTree as a Functor

instance Functor BinaryTree where
    fmap f (Leaf x) = Leaf $ f x
    fmap f (Branch xs1 xs2) = Branch (fmap f xs1) (fmap f xs2)

• No constraints on BinaryTree's type arguments
  – There are no explicit type arguments to constrain!

Another type constructor class — Foldable

• We’ve seen this in the type signature for the fold functions

   *Main> :t foldr
   foldr :: (a -> b -> b) -> b -> t a -> b
   *Main> :t foldl
   foldl :: (b -> a -> b) -> b -> t a -> b

• The list type constructor [] is a member of Foldable

• With your functions binaryTreeFoldl and binaryTreeFoldr we could define:

    instance Foldable BinaryTree where
        foldl = binaryTreeFoldl
        foldr = binaryTreeFoldr
5.6 Examples of larger Haskell libraries

5.6.1 Monadic parser combinators

Laziness as a design technique

- There’s more to laziness than the ability to describe and partially evaluate an infinite list
  - (The list of prime numbers is in fact infinite, even if we only ever calculate some prefix of it.)
- We can structure a system so that we describe large lists, knowing that we will only calculate as much as we need

A larger example: a parsing library

- What, in general, is a parser?
- The parser of most compilers takes a list of lexemes, and returns some abstract representation of the program
  - *Lexeme*: a simple grouping of characters into basic program units, like identifiers, keywords, constants, and specific punctuation
  - Typically lexemes are specified by a regular expression and the program itself is specified by a grammar
- So at first glance, a parser might have type `[input] -> output`

Composing parsers

- What we’d really like is a handy way to combine parsers
  - Will let us write parsers which look like grammars
  - Much more maintainable than explicit recursive descent, or certainly a bottom-up parser
- One parser might do some of the work, leaving the rest for another parser
- So the result of a parser *must also return the unused input*,
  
  `[input] -> (result,[input])`

But what about failure?

- If we combine parsers in *alternation* (accept either an X or a Y), then we may fully expect one of the two to *always to fail*
  - An operation might be +, or it might be − — but it can’t be both, so one of those options would fail
  - A basic expression could be a constant, or it could be a variable — but it can’t be both, so one of those options would fail
- How do we handle this?
  - We could do something horrible by throwing exceptions with `error` and catching them
  - But `error` is made for genuine errors; this is something we expect routinely
  - Replace `failure` with a *list of successes*.
- A parser returns *the different possible ways of parsing its input*

```haskell
newtype Parser input output = Parser ([input] -> [(output,[input])])
parse (Parser p) = p
```
Running a parser

• So we’ve reasoned that the right type for a parser should be

```haskell
newtype Parser input output = Parser ([input] -> [(output,[input])])
parsers (Parser p) = p
```

• Whatever failures and multiple parses we find at intermediate points within the grammar, we usually expect the top-level final parser to produce a unique result from all of the input

```haskell
getParse :: Parser i o -> [i] -> o
getParse parser input =
  case parse parser input of
      [(result,[])] -> result
      [] -> error "No parse"
      [(result,_,)] -> error "Input not consumed"
      _ -> error "Parse is not unique"
```

• Or maybe we won’t care about the uniqueness, and we relax that restriction

Building blocks

• How do we build a parser?

• What are our starting points?

• The most basic possible parsers will either return some result without consuming input, or return no result

```haskell
accept :: result -> Parser input result
accept res = Parser \inp -> [(res, inp)]
```

```haskell
reject :: Parser input result
reject = Parser \inp -> []
```

• For example:

```haskell
> parse (accept 'x') "abc123"
[('x','abc123')]
> parse reject "abc123"
[]
```

Considering the input

• The most basic parsers are a little surprising, since they do not actually look at their input!

• Here’s a simple parser which expects an exact piece of input

```haskell
literal :: Eq a => a -> Parser a a
literal s = Parser literal'
  where literal' (x:xs) | x == s => s = [(x, xs)]
                  _ = []
```

• For example:
> :t letterA
letterA :: Parser Char Char
> parse letterA "Asdf"
[("A","sdf")]
> parse letterA "asdf"
[]
> parse letterA "1234"
[]

**General criteria for input**

- More generally, we can define `literal` as a special case of a parser that takes a predicate for an acceptable piece of input

```
satisfy :: (a -> Bool) -> Parser a a
satisfy f = Parser (\inp -> case inp of
  (x:xs) | f x -> [(x,xs)]
  _ -> [])
```

```
literal :: Eq a => a -> Parser a a
literal s = satisfy (== s)
```

- For example:

```
> import Data.Char
> :t isUpper
isUpper :: Char -> Bool
> let upperLetter = satisfy isUpper
> :t upperLetter
upperLetter :: Parser Char Char
> parse upperLetter "ASDF"
[("A","SDF")]
> parse upperLetter "asdf"
[]
```

**Need more useful results**

With `satisfy` and `literal`, the result of a parser is just one piece of its input

- But usually we think of a parser as *transforming* its input in some way
- We can transform the result by applying a function

![Diagram of parser transformation](image)

- We want to take this combination itself as one of our composable parsers

**The transformer**

```
infixl 6 ‘using’
using :: Parser inp res -> (res -> res’) -> Parser inp res’
(Parser p) ‘using’ f
  = Parser (\inp -> [(f res, inp’) | (res,inp’) <- p inp])
```
• For example:

```haskell
> let upDown = upperLetter 'using' toLower
> :t upDown
upDown :: Parser Char Char
> parse upDown "ASDF"
[(\'a\',"SDF")]
> parse upDown "asdf"
[]
```

Combining parsers — choice

• The `alt` combinator combines two parsers as alternatives

```haskell
infixl 4 'alt'
alt :: Parser inp res -> Parser inp res -> Parser inp res
p1 'alt' p2 = Parser (\input -> parse p1 input ++ parse p2 input)
```

• For example:

```haskell
> let lowerA = literal 'a'
> let lowerZ = literal 'z'
> let lowerAorZ = lowerA 'alt' lowerZ
> :t lowerAorZ
lowerAorZ :: Parser Char Char
> parse lowerAorZ "asdf"
[(\'a\',"sdf")]
> parse lowerAorZ "zxcv"
[(\'z\',"xcv")]
> parse lowerAorZ "qwer"
[]
```

What should it mean to combine two parsers in sequence?

Certainly a key idea is that the second parser should operate on the input remaining from the first parser.

But what do we do with two results?

Let the first result produce the next parser

The solution: use the first result to produce the second parser

• Instead of combining a `Parser in out1` with a `Parser in out2`,
• Combine a `Parser in out1` with a `function of type out1 -> Parser in out2`
Combining parsers — sequence

- The `thn` combinator combines two parsers sequentially

  ```haskell
  infixr 5 'thn'
  thn :: Parser inp res1 -> (res1 -> Parser inp res2) -> Parser inp res2
  p1 'thn' fp2 = Parser (
      
    )
  ```

- For example:

  ```haskell
  > let parseSecondAorZ c1
  >     = lowerAorZ 'thn' \c2 -> accept (c1,c2)
  > let parseTwoAorZ = lowerAorZ 'thn' parseSecondAorZ
  > :t parseTwoAorZ
  > parseTwoAorZ :: Parser Char (Char, Char)
  > parse parseTwoAorZ "azsxdcfv"
  `[((a,'z'),"sxdcfv")]
  > parse parseTwoAorZ "asxdcfv"
  `[]`
  ```

The Kleene star

- What does it mean to apply a parser zero or more times?
  - As a result, we would expect to get a list of that parser’s result type
  - We could apply it once, and then apply it some more — that’s a `thn`
  - Or we could do something else — that’s `alt`
  - The something else is `nothing` — which `accept` gives us

- Translating to Haskell:

  ```haskell
  many :: Parser inp res -> Parser inp [res]
  many p = (p 'thn' \first ->
    many p 'thn' \rest ->
    accept (first : rest))
  'alt' accept []
  ```

- For example:

  ```haskell
  > let lowerAorZs = many lowerAorZ
  > :t lowerAorZs
  > lowerAorZs :: Parser Char [Char]
  > parse lowerAorZs "azazsxdcfv"
  `[(("azaz","sxdcfv"),("aza","azsxdcfv"),("az","azsxdcfv"),
    ("a","azasxdcfv"),("","azasxdcfv"))]
  ```

The Kleene plus

- We remove the option to do nothing
some :: Parser inp res -> Parser inp [res]
some p = p 'thn' \first ->
    many p 'thn' \rest ->
    accept (first : rest)

• For example:

> let lowerAorZs1 = some lowerAorZ
> parse lowerAorZs1 "azazsxdcfv"
[(["azaz","sxdcfv"),("aza","zsxdcfv"),
  ("az","azsxdcfv"),("a","zaszsxdcfv")]

Multiple results

• The multiple results let us account for the fact that we do not know what subsequent parsers may demand

> parse lowerAorZs "azazsxdcfv"
[(["azaz","sxdcfv"),("aza","zsxdcfv"), ("az","azsxdcfv"),
  ("a","zaszsxdcfv"), ("","azaszsxdcfv")]

  – We may demand that a single \text{z} must follow the lowerAorZs

> let p2 = lowerAorZs 'thn' \xs ->
  lowerZ 'thn' \x ->
  accept (xs,x)
> :t p2
p2 :: Parser Char ([Char], Char)
> parse p2 "azazsxdcfv"
[(["aza","z"'),"sxdcfv"),(["a","z"'),"azsxdcfv")]

  – Is it inefficient to produce all of these parses?

    * No — Laziness will only generate the possibilities which are necessary!

  – We may demand a single \text{z}, and then a single \text{s}, as followers

> let p3 = lowerAorZs 'thn' \xs ->
  lowerZ 'thn' \_ ->
  literal 's' 'thn' \_ ->
  accept xs
> parse p3 "azazsxdcfv"
[(["aza","xdcfv")]

6 Lambda calculus

6.1 Syntax

Untyped lambda calculus

• A calculus is a way of writing expressions, plus a way of relating one expression to another

• The lambda calculus lets us discuss many programming language features in a very minimal language

• Terms $M, N, M'$ etc.:

  – Variables $x, x', y, z_0, z_1, \ldots$

  – Abstractions $\lambda x. M$

  – Applications $MN$
Use parentheses to disambiguate

- Abstractions include as much as possible to the right: \( \lambda x. MN \) means \( \lambda x. (MN) \), not \( (\lambda x. M)N \)
- Function application is left-associative: \( LMN \) means \( (LM)N \), not \( L(MN) \)

**Exercise 6.1.** Remove all possible parentheses from these expressions so as not to change the interpretation of each.

1. \(((\lambda x. (\lambda y. ((x)y)))(\lambda z. (z)))\)
2. \((xy)(xz)\)
3. \((\lambda x. ((\lambda y. (\lambda z. z)))y)\)
4. \((\lambda z. (z(y)))((\lambda u. u))\)

Answers to the first two items are on p. 90.

**Properties of terms**

- **Free and bound variables**
  - A lambda abstraction \( \lambda x. M \) binds occurrences of \( x \) in \( M \)
  - Static scope for parameters
  - A variable with a corresponding abstraction is said to be free

- Formulas for the variables which occur free and bound in a term:

  \[
  \begin{align*}
  \text{fv}(x) &= \{x\} \\
  \text{fv}(MN) &= \text{fv}(M) \cup \text{fv}(N) \\
  \text{fv}(\lambda x. M) &= \text{fv}(M) \setminus \{x\}
  \\
  \text{bv}(x) &= \{\} \\
  \text{bv}(MN) &= \text{bv}(M) \cup \text{bv}(N) \\
  \text{bv}(\lambda x. M) &= \text{bv}(M) \cup \{x\}
  \end{align*}
  \]

- If \( \text{fv}(M) = \{\} \), then \( M \) is **closed**. Otherwise, \( M \) is **open**.

- **Values** represent the forms of expression which (informally) we take to be end products of a computation.
  - Abstractions are values
  - Applications are non-values
  - Variables are negotiable!
    * Sometime we take them to be values
    * Sometimes not
    * It depends on the technical details of the particular system we will consider

- **Syntactic identity** \( \equiv \)

**Exercise 6.2.** Write out the sets of free variables and of bound variables for each of the following expressions.

- \((\lambda x. (\lambda y. zxy))(\lambda z.xzw)\)
- \(\lambda x. xy(xz)\)
- \((\lambda x. (\lambda y. (\lambda z. z)) y)\)
Substitution

A basic operation is substituting a term \( N \) for a variable \( x \) in some other term \( M \)

- Written \( M \left[ \frac{N}{x} \right] \)
- Specifically:
  \[
  x \left[ \frac{N}{x} \right] \equiv N \\
  x \left[ \frac{N}{y} \right] \equiv x \quad x \neq y
  
  (LM) \left[ \frac{N}{x} \right] = (L \left[ \frac{N}{x} \right]) (M \left[ \frac{N}{x} \right])
  
  (\lambda x. M) \left[ \frac{N}{x} \right] = \lambda x. (M \left[ \frac{N}{x} \right]) \\
  (\lambda x. M) \left[ \frac{N}{y} \right] = \lambda x. (M \left[ \frac{N}{y} \right]) \\
  (\lambda x. M) \left[ \frac{z}{x} \right] \left[ \frac{N}{y} \right] = \lambda z. (M \left[ \frac{z}{x} \right] \left[ \frac{N}{y} \right]) \\
  x \in \text{fv}(N), z \notin M, N
  
  Exercise 6.3. Simplify each of the following expressions, writing them as plain lambda terms without substitutions.
  
  - \( ((\lambda x. (\lambda y.zxy))(\lambda z.xzw)) \left[ \frac{\lambda k.kk}{x} \right] \)
  - \( (\lambda x. (\lambda y.zxy)) \left[ \frac{\lambda k.kk}{x} \right] \)
  - \( (\lambda x. (\lambda y.zxy)(\lambda z.xzw)) \left[ \frac{\lambda k.kk}{x} \right] \)

  Exercise 6.4. [Barendregt, Sec. 2.2] In combinatory logic one considers a small number of closed terms rather than general lambda expressions. One common system uses three terms, \( I \equiv \lambda x.x, S \equiv \lambda x.\lambda y.\lambda z.xz(yz), K \equiv \lambda x.\lambda y.x. \) Show that
  
  - \( I = S\ K\ K \)
  - \( \lambda x. M = K M \) if \( x \notin \text{fv}(M) \)
  - \( \lambda x. M N = S(\lambda x. M)(\lambda y. N) \)

  What are the normal forms of these terms:
  
  - \( (\lambda y.yyy)((\lambda a.\lambda b.a)I(S\ S)) \)
  - \( S\ S\ S\ S\ S\ S\ S\ S\ S \)

6.2 Reduction

Relating terms by reduction

Three \textit{reduction} rules

\[
  (\alpha) \quad \lambda x. M \rightarrow \lambda y. M[y/x] \quad \text{if } x \neq y, y \notin \text{fv}(M)
  
  (\beta) \quad (\lambda x. M) N \rightarrow M[N/x] \quad \text{if } x \notin \text{fv}(M)
  
  (\eta) \quad \lambda x. M x \rightarrow M \quad \text{if } x \notin \text{fv}(M)
  
  73
• In modern presentations of the lambda calculus for computer science, we do not consider $\alpha$ reduction to be "interesting" computational work
  – So when we write a term, we mean to denote the equivalence class of terms modulo $\alpha$ reduction
• Barendregt’s hygene condition
• A term with no redexes is said to be in normal form
  – Can also discuss $\beta$-normal or $\eta$-normal form
• From reduction to equality

**Exercise 6.5.** Identify all of the redexes in the following terms.

- $(\lambda x.\lambda y. yx)(\lambda y. (\lambda z. wz)y)$
- $z(\lambda z. (\lambda x. xx)(\lambda y. zxy))$

**Exercise 6.6.** Apply the hygene condition to each of the expressions in Exercises 6.2, 6.3 and 6.5

**Exercise 6.7.** Reduce each of the expressions in Exercises 6.3 and 6.5:
  - To $\beta$ normal form
  - To $\eta$ normal form
  - To $\beta,\eta$ normal form
  - To each of the possible ways of contracting one single redex in each term

**Properties of reduction**

- Uniqueness of normal forms
  - If $M \rightarrow M_1$, $M \rightarrow M_2$, and both $M_1$ and $M_2$ are normal forms
    – Then $M_1 \equiv M_2$ (modulo $\alpha$
  - However, it is not guaranteed that every term will have a normal form!

**Confluence** (aka the Church-Rosser property, aka the diamond property)

- If $M \rightarrow M_1$ and $M \rightarrow M_2$
  – Then there is some $N$ such that both $M_1 \rightarrow N$ and $M_2 \rightarrow N$

**Church encodings**

- Booleans
  – true $\equiv \lambda m. \lambda n. m$
  – false $\equiv \lambda m. \lambda n. n$
  – if $\equiv \lambda p. \lambda m. \lambda n. pmn$

- Pairs
  – mnpair $\equiv \lambda x. \lambda y. \lambda f. fxy$
- \texttt{fst} \equiv \lambda x. \lambda y. x
- \texttt{snd} \equiv \lambda x. \lambda y. y

- \textbf{Numbers}
  - \texttt{0} \equiv \lambda f. \lambda x. x
  - \texttt{1} \equiv \lambda f. \lambda x. fx
  - \texttt{2} \equiv \lambda f. \lambda x. f(fx)
  - \texttt{3} \equiv \lambda f. \lambda x. f(f(fx)) \text{ and so on}
  - \texttt{isZero} \equiv \lambda n. n (\lambda x. \texttt{false}) \texttt{true}
  - \texttt{succ} \equiv \lambda n. \lambda f. \lambda x. nfx
  - \texttt{plus} \equiv \lambda n1. \lambda n2. \lambda f. \lambda x. n1f(n2fx)
  - \texttt{times} \equiv \lambda n1. \lambda n2. \lambda f. \lambda x. n1(n2fx)
  - Subtraction, division more complicated but possible
  - Then negative numbers, rationals, reals, etc.

\textbf{Exercise 6.8.} Define a closed lambda term \texttt{and} so that

\begin{itemize}
  \item \texttt{and true true} \rightarrow_{\beta} \texttt{true}
  \item \texttt{and true false} \rightarrow_{\beta} \texttt{false}
\end{itemize}

and so on. Write out the details of each of the four relationships.

Do the same for \texttt{or}, \texttt{not} and \texttt{xor}.

\textbf{Exercise 6.9.} Define a partial \texttt{signum} operator, which returns \texttt{0} for \texttt{0}, and \texttt{1} for any positive number.

\textbf{Exercise 6.10.} Write out the step-by-step details of the following reductions:

\begin{itemize}
  \item \texttt{fst (mkpair 2 3)} \rightarrow_{\beta} \texttt{2}
  \item \texttt{snd (fst (mkpair (mkpair 3 4) \texttt{false}))} \rightarrow_{\beta} \texttt{4}
  \item \texttt{succ 4} \rightarrow_{\beta} \texttt{5}
  \item \texttt{succ (succ 2)} \rightarrow_{\beta} \texttt{4}
  \item \texttt{plus (plus 2 3) 1} \rightarrow_{\beta} \texttt{6}
  \item \texttt{iszero 0} \rightarrow_{\beta} \texttt{true}
  \item \texttt{iszero (succ 1)} \rightarrow_{\beta} \texttt{false}
  \item \texttt{times 2 (plus 1 2)} \rightarrow_{\beta} \texttt{6}
\end{itemize}

\textbf{Exercise 6.11.} Define the following functions. Write out reduction sequences to show that each is defined correctly on \texttt{1}, \texttt{2} and \texttt{4}.

\begin{itemize}
  \item \texttt{factorial}
  \item \texttt{sumOneUp x} = \sum_{i=1}^{x} i
\end{itemize}
Delta reduction

• The Church encodings show how to represent various concepts as plain lambda expressions
  – It’s interesting that it’s possible
  – In fact, every computable function can be represented in \( \lambda \)
• But we can also extend the language of terms
  – Extend lambda terms to include various other symbols
  – Specify a function \( \delta \) mapping strings of these symbols to one symbol
  – Then \( \delta \) reduction rewrites applications across a string:
    * If \( \delta(s_1 \cdots s_n) = s_0 \)
    * Then \( \cdots ( (s_1 s_2) s_3 ) \cdots ) s_n \rightarrow_\delta s_0 \)

6.3 Parameter-passing disciplines

Two parameter-passing mechanisms

Earlier this semester we discussed call-by-name and call-by-value

• Call-by-value has a simple explanation in the stack-frame model
• Call-by-name... not so much

So we will use the \( \lambda \) calculus to build a simple (but rigorous!) comparison of the two

• Distinguish them by the answer to the question
  – How do we pick the next \( \beta \)-redex?
• Here we specifically mean \( \beta \)-redexes — we are interested in runtime computation

Picking the next redex

We have already seen some aspect of a strategy for picking a redex

Consider \( (\lambda x. x)(\lambda y. (\lambda z. z) y)(\lambda m. \lambda n. n m m) \)

• At the top level, this is an application
  – \( (\lambda x. x)(\lambda y. (\lambda z. z) y) \) on the left side
  – \( (\lambda m. \lambda n. n m m) \) on the right side
• This whole term is not itself a \( \beta \)-redex
  – We would need an abstraction on the left
  – There is work to do on the left side before it becomes an abstraction
  – So in this situation we have usually looked for our first redex on the left side

A first rule for finding the next redex

• When looking for a redex with an abstraction
  – If the term on the left is not an abstraction
  – Then look for the next redex on the left

So then what?
If there is an abstraction on the left, do we then reduce the application?

Maybe.

• If call-by-value, no — in call-by-value languages, we expect a value for an actual parameter
• If call-by-name, yes — we are happy to use unevaluated expressions as actual parameters
Call-by-name and call-by-value in words
Can we do better than an informal English description??

Call-by-name

• If a term is an abstraction, we have a value
• If a term is a variable, we are stuck
• If a term is an application of an abstraction, reduce this term
• If a term is an application of a non-abstraction, look for the next redex on the left side of the application

Call-by-value

• If a term is an abstraction, we have a value
• If a term is a variable, we are stuck
• If a term is an application of an abstraction to a value, reduce this term
• If a term is an application of an abstraction to a non-value, look for the next redex on the right side of the application
• If a term is an application of a non-abstraction, look for the next redex on the left side of the application

Contexts

A context is just a term with a hole

\[ C ::= [] \mid M C \mid C M \mid \lambda x. C \]

We can break up any term into a context, plus the expression in its hole

For example, \((\lambda x.x) (\lambda y. (\lambda z.z) y) (\lambda m. \lambda n.n m m)\) is

• The context \([\_]\) \((\lambda m. \lambda n.n m m)\)
• With \((\lambda x.x) (\lambda y. (\lambda z.z) y)\)

It is also

• The context \((\lambda x.x) [\_]\) \((\lambda m. \lambda n.n m m)\)
• With \lambda y. (\lambda z.z) y

But we are mostly interesting in seeing a \(\beta\)-redex in the hole!

We can describe \(\beta\) reduction in any position as:

\[ C[M] \rightarrow C[N] \text{ if } M \rightarrow N \]

Using contexts to describe finding the next redex

We describe the position of the next redex by giving a subset of contexts, called evaluation contexts

• One grammar for call-by-name, one for call-by-value

Call-by-name informally

• If a term is an abstraction, we have a value
• If a term is a variable, we are stuck
• If a term is an application of an abstraction, reduce this term
• If a term is an application of a non-abstraction, look for the next redex on the left side of the application

Call-by-name evaluation contexts

\[ E ::= \emptyset \mid E \ M \]

Call-by-value informally

• If a term is an abstraction, we have a value
• If a term is a variable, we are stuck
• If a term is an application of an abstraction to a value, reduce this term
• If a term is an application of an abstraction to a non-value, look for the next redex on the right side of the application
• If a term is an application of a non-abstraction, look for the next redex on the left side of the application

Call-by-value evaluation contexts

\[ E ::= \emptyset \mid E \ M \mid (\lambda x. M) \ E \]

Also for call-by-value

Restrict \( \beta \) reduction to value arguments

\[
(\beta_V) \quad (\lambda x. M)V \to M \left[ \frac{V}{x} \right]
\]

6.4 Simply-typed \( \lambda \)

Simply-typed lambda calculus

• Environments and judgments
  – An environment is a set of assumptions about how free variables are typed
    * For example: \( x: \text{Int}, f:\text{Int} \to \text{Int} \)
    * Use upper-case Greek letters for environments: \( \Gamma \)
    * Also called contexts
  – A judgment is a statement that we have proven an expression to be of a particular type
    * Environment \( \vdash \) term : type

• Simply-typed lambda calculus (with integers)

\[
\begin{align*}
\text{Var} & \quad \Gamma, x : A \vdash x : A \\
\text{Const} & \quad \Gamma \vdash \text{Int} \quad \text{etc.} \\
\text{Abstr} & \quad \Gamma, x : A \vdash M : B \\
\Gamma & \vdash \lambda x. M : A \to B \\
\text{Apply} & \quad \Gamma \vdash M : A \to B \quad \Gamma \vdash N : A \\
\Gamma & \vdash MN : B
\end{align*}
\]

• Important properties
  – Progress: If \( \Gamma \vdash M : A \), then either \( M \to N \), or \( M \) is a value
  – Preservation: If \( \Gamma \vdash M : A \), and \( M \to N \), then \( \Gamma \vdash N : A \)
  – Normalizing: If \( \Gamma \vdash M : A \), then \( M \) reduces to a normal form in a finite number of steps
Exercise 6.12. For each of the closed terms in Exercises 6.1 through 6.5 determine which are simply-typable (and their types), and which are not. For each of the open terms, which are simply-typable under a suitable environment? In both, write out the typing trees (and environments) for each which is typable; explain why not for the ones which are not.

Pair and sum types

• Pairs

\[
\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \times B}
\]

\[
\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \text{fst} M : A}
\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \text{snd} M : B}
\]

\[\text{fst}(M, N) \rightarrow M\]
\[\text{snd}(M, N) \rightarrow N\]

• Sums

\[
\frac{\Gamma \vdash M : A}{\Gamma \vdash \text{left } M : A + B}
\frac{\Gamma \vdash M : B}{\Gamma \vdash \text{right } M : A + B}
\]

\[
\frac{\Gamma \vdash L : A + B \quad \Gamma, x : A \vdash M : C \quad \Gamma, y : B \vdash N : C}{\Gamma \vdash \text{case } L \text{ of left } x.M \mid \text{right } y.N : C}
\]

\[\text{case } (\text{left } L) \text{ of } x.M \mid \text{right } y.N \rightarrow M \left[\frac{L}{x}\right]\]

\[\text{case } (\text{right } L) \text{ of } x.M \mid \text{right } y.N \rightarrow N \left[\frac{L}{y}\right]\]

Exercise 6.13. What are appropriate reduction rules for expressions with $\times$ and $+$ types?

6.5 Polymorphism in lambda calculi

A simple version of polymorphism

• Extend types: could also be a type variable $\alpha$, or a quantification $\forall \alpha. T$

• Rules:

\[
\frac{\Gamma \vdash M : A}{\Gamma \vdash \forall \alpha. M : \forall \alpha. A}
\quad \text{if } \alpha \not\in \text{fv}(\Gamma)
\]

\[
\frac{\Gamma \vdash M : \forall \alpha. A}{\Gamma \vdash M : A \left[\frac{T}{\alpha}\right]} \quad \text{for any type } T
\]

• More polymorphism than practical
  – Most languages restrict where type variables can be abstracted
  – In Java/Scala, at class/method declarations

Exercise 6.14. Work out closed polymorphic types for each of the Church-encoded terms above. There may be several possible types for each, with quantifiers placed in different positions. Give types with:

• The quantifiers placed as deeply within the type as possible
• All quantifiers to the far outer left of the type
7 Object-oriented programming and Scala

7.1 Subclasses and what they unlock

Extending classes
You have probably been writing subclasses since your first course in Java.

```scala
class A {
  def f(x:Int):Int = x+10
  // ...other definitions...
}
class B extends A {
  override def f(x:Int):Int = 2
  def g(s:String) = f(s.length())
}
```

• The subclass is a foundational aspect of object-oriented programming
  – Class B "inherits everything" from class A except for having its own recipe for method `f`
  – Class B has a method `g` not present in class A

• Adding this mechanism to a language opens up many interesting semantic questions
  – As usual, not every language makes the same decisions as Java

First questions
```scala
class A { // 1
  def f(x:Int):Int = x+10 // 2
  // ...other definitions... // 3
}
class B extends A { // 5
  override def f(x:Int):Int = 2 // 6
  def g(s:String) = f(s.length()) // 7
}
```

What do these lines print?
```scala
val b:B = new B()
println(b.f(5))
```

• No particular surprise: 2

What about these lines?
```scala
val a:A = new B()
println(a.f(5))
```

• For a few minutes, we’ll ignore the question of why it’s OK create a B and assign it to a storage location declared to hold an A

• If we’re working with Java or Scala, then the output is the same: 2

• But again we have found a feature which is a decision made by the language designer, and which differs in other languages
Dispatch

• The association of a method call with code to be run is called dispatch

• What determines how we dispatch the call to a.f(5) ?

class A {
  def f(x:Int):Int = x+10
  // ...other definitions...
}
class B extends A {
  override def f(x:Int):Int = 2
  def g(s:String) = f(s.length())
}

object AB {
  def main(args:Array[String]):Unit = {
    val a:A = new B()
    println(a.f(5))
  }
}

• Java and Scala use dynamic dispatch
  – The instantiated type B determines the dispatched method
  – The type of the storage location for a subsequent assignment or method call does not matter
  – The type at instantiation determines the dispatch of all method calls

• Some other languages, notably C++ by default, use static dispatch
  – The declared type A determines the dispatched method
  – The actual type of the object itself is not consulted
  – This dispatch is determined at compile-time
    * Whereas the instantiation type of objects is a runtime property

Subtypes

Subtyping is another key idea in object-oriented languages

If A is a subtype of B, then we should be able to use an instance of A in any context which calls for a B

• Might seem to be contrary to the idea of static typing, since one type can be provided where a different type is called for
  – But in fact, static typing is absolutely possible with subtyping — Java has used it for years!

• So with classes A and B, it is the principle of subtyping which justifies allowing the assignment

val a:A = new B()

  – Since any operation which might be demanded of an A can be provided by a B
  – It does not matter that the implementation of the methods in B may be different
  – It does not matter that there may be additional methods in B, since if a context expects an A it would never invoke them

• This relationship is not symmetric:
Subtypes and subclasses

Subtypes and subclasses are not the same idea!

• It is true that if A is a subclass of B (which we write A <: B), then A is a subtype of B
• But there are also other ways to judge that one type should be taken as a subtype of another
  – In particular, there are interesting interactions between generics and subtyping, and between function types and subtyping
  – Will look in more detail at this relationship in the next few lectures

7.2 Function values

Like Haskell, Scala has anonymous functions
  Functions are first-class values in Scala as well as in Haskell

In the Scala interpreter:

```scala
scala> val f = { (x:Int) => x+10 }
f: Int => Int = <function1>
scala> f(3)
res0: Int = 13
```

• Syntax for an anonymous function: { (x:Int) => x+10 }
  – The curly braces are optional, but make it much clearer
• We can bind functions to names, and call them
  – The parentheses are not optional: it’s f(3), not f 3
• There are function types
  – f has type Int => Int
  – Unlike Java, there actually is a separate function type!
  – Not an abbreviation for a single-member interface
• Functions are not printable, so instead it prints <function1>

7.3 Generic patterns involving multiple classes

Multiclass genericity

```
Defining related generic classes

```java
public interface LeftSide< A extends LeftSide<A,B>,
        B extends RightSide<A,B> > {
    public B toRight();
}

public interface RightSide< A extends LeftSide<A,B>,
        B extends RightSide<A,B> > {
    public A toLeft();
}

public class MyLeft implements LeftSide<MyLeft, MyRight> {
    public MyLeft(int leftVal) { this.leftVal = leftVal; }
    private int leftVal;
    public MyRight toRight() {
        return new MyRight(Integer.toString(leftVal));
    }
}

public class MyRight implements RightSide<MyLeft, MyRight> {
    public MyRight(String rightVal) { this.rightVal = rightVal; }
    private String rightVal;
    public MyLeft toLeft() { return new MyLeft(rightVal.length()); }
}
```

7.4 Type variables and members

7.4.1 Scala generics and variance

Generics

Scala’s system for *generic types* (that is, parametric polymorphism) resembles Java in simpler cases

```scala
class Buffer[X](private var contents:X) {
    def get():X = contents
    def set(x:X):Unit = {
        contents = x
    }
}
```

Generics and methods

Individual methods can also be generic

```scala
class GenMethod {
    def stringLen[A](a:A):Int = a.toString().length()
}
```

• Note order of parameter lists
• Straightforward for resolving scoping of variables

```scala
class ElementGrabber {
    def firstOf[A](z:List[A]):A = z match {
        case (x :: xs) => x
        case _ => throw new RuntimeException("Empty")
    }
}
```
Generics and subtyping

class G[X] {
}

• If \( B \) is a subtype of \( A \), then is:
  – \( G[B] \) a subtype of \( G[A] \)?
  – \( G[A] \) a subtype of \( G[B] \)?
  – Or neither?

• By default there’s no relation

scala> val gA : G[A] = new G[B]()
<console>:10: error: type mismatch;
  found : G[B]
  required: G[A]

scala> val gB : G[B] = new G[A]()
<console>:10: error: type mismatch;
  found : G[A]
  required: G[B]

Covariant subtyping

We can declare a **covariant** relationship between \( G \) and its type argument

class G[+X] {
}

• Then \( G[B] \) is a subtype of \( G[A] \)

scala> val gA : G[A] = new G[B]()
gA: G[A] = G@327514f
scala> val gB : G[B] = new G[A]()
<console>:10: error: type mismatch;
  found : G[A]
  required: G[B]

Contravariant subtyping

The opposite relationship is **contravariant** subtyping

class G[-X] {
}

• Then \( G[A] \) is a subtype of \( G[B] \)

scala> val gB : G[B] = new G[A]()
gB: G[B] = G@3e6ef8ad
scala> val gA : G[A] = new G[B]()
<console>:10: error: type mismatch;
  found : G[B]
  required: G[A]
Another use of covariance: method result types

- Overriding \( g \) is allowed because of covariance for method result types

```scala
class Cov1 {
  def g():A = new A()
}

class Cov2 extends Cov1 {
  override def g():B = new B()
}
```

- Why is this allowed?
  - Consider any use of the result of a call to \( g \) on a \( \text{Cov1} \)
    ```scala
    val c:Cov1 = getMeACov1(...) 
    val a:A = c.g()
    ```
  - Any \( \text{B} \) instance can be assigned to \( a \), since \( \text{B} \) is a subtype of \( \text{A} \)
  - So if \( g \) were actually to return a \( \text{B} \), the assignment is still valid
  - So an override such as \( \text{Cov2} \)'s is always OK
  - (This works in Java, too)

Covariance and contravariance apply to function types too

- Another background class

```scala
class E() {
}
```

- Think about function types

```scala
scala> val fEA: E => A = { x:E => new A() } 
fEA: E => A = <function1>
scala> val fEB : E => B = { x:E => new B() } 
fEB: E => B = <function1>
```

- What is the relationship between \( \text{E} => \text{A} \) and \( \text{E} => \text{B} \) ?
  - \( \text{E} => \text{B} \) is a subtype of \( \text{E} => \text{A} \)

```scala
scala> val fE : E => A = fEB 
fE: E => A = <function1>
```

- So the function type is covariant in the result type

- But is there a relationship between \( \text{A} => \text{E} \) and \( \text{B} => \text{E} \) ?

How can contravariance be a thing?

- Again two functions

```scala
scala> val fAE : A => E = { x:A => new E() } 
fAE: A => E = <function1>
scala> val fBE : B => E = { x:B => new E() } 
fBE: B => E = <function1>
```
• What is their relationship?
• Not covariant — \texttt{A} \Rightarrow \texttt{E} is \textit{not} a subtype of \texttt{B} \Rightarrow \texttt{E}

```
scala> val gE : A => E = fBE
<console>:11: error: type mismatch;
  found : B => E
  required: A => E
```

• But it is contravariant in the argument position —

```
scala> val gE : B => E = fAE
gE: B => E = <function1>
```

• Why is this right?
  – If a value \( x \) can be supplied for a type \( A \), it must be able to do everything we expect of an \( A \)
  – If we know that it \textit{can’t} do certain of those things, than it’s not acceptable at that type
  – If we can say this is systematically of all elements of another type \( B \), then \( B \) must not be a subtype of \( A \).

**Capabilities example**

class A() {
}
class B() extends A() {
}
class C() extends A() {
}
class E() {
}

• A function of type \( A \Rightarrow E \) should be able to take an argument of type \( A, B \) or \( C \)
  – If we assert this typing, then we are committed to this capability
  – If we provide a function of type \( B \Rightarrow E \), then we fall short of this commitment

• A function of type \( B \Rightarrow E \) should be able to take an argument of type \( B \)
  – But not necessarily an argument of type \( A \) or \( C \)
  – A function of type \( A \Rightarrow E \) delivers on the commitment made by an assertion of the type \( B \Rightarrow E \)
  – So \( A \Rightarrow E \) is a subtype of \( B \Rightarrow E \), even though \textit{(because!)} \( B \) is a subtype of \( A \)

**A sample class**

• \( A \) and \( B \) as last time

class A { }
class B extends A { }

• \( G \) with a method that uses its type variable
class G[T](maker: Int=>T) {
    def mthd(x: Int): T = maker(x)
}

• What variance can we imagine for T?

**Does covariance make sense?**
For a value g of type G[A], could we provide something of type G[B]?

- The context might call g.mthd(i), and expect to get a value of type A
- That call on an object of type G[B] would return a B
- But B is a subtype of A, so it’s OK
  – That’s also written B <: A
- So it’s reasonable to consider G[B] a subtype of G[A]
- And we could declare

  class G[+T](maker: Int=>T) {
      def mthd(x: Int): T = maker(x)
  }

**A sample class**
- A and B as yesterday

  class A { }
  class B extends A { }

- G with a method that uses its type variable

  class G[T](maker: Int=>T) {
      def mthd(x: Int): T = maker(x)
  }

- What variance can we imagine for T?

**Does contravariance make sense?**
For a value g of type G[B], could we provide something of type G[A]?

- The context might call g.mthd(i), and expect to get a value of type B
- That call on an object of type G[A] would return a A
- A is not a subtype of B, so that’s a type mismatch
- So it’s not reasonable to consider G[A] a subtype of G[B]

87
Another sample class

class A { }
class B extends A { }

class H[T]() {
    def mthd(x: T) = 99
}

• What variance can we imagine for \( T \)?

Does covariance make sense?
For a value \( h \) of type \( H[A] \), could we provide something of type \( H[B] \)?

• The context might call \( h.mthd(new \ A()) \)
• \( mthd \) on an object of type \( H[B] \) would require an argument of type \( B \)
• \( B \) new \( A() \) cannot be used where a \( B \) is required, because \( A \) is \textit{not} a subtype of \( B \) — it’s the other way around
  – So that’s a type mismatch
• So it’s not reasonable to consider \( H[B] \) a subtype of \( H[A] \)

Another sample class

class A { }
class B extends A { }

class H[T]() {
    def mthd(x: T) = 99
}

• What variance can we imagine for \( T \)?

Does contravariance make sense?
For a value \( g \) of type \( H[B] \), could we provide something of type \( H[A] \)?

• The context might call \( h.mthd(new \ B()) \)
• \( mthd \) on an object of type \( H[A] \) would require an argument of type \( A \)
• \( B \) new \( B() \) \textit{can} be used where type \( A \) is required, because \( B<:A \)
• So it’s reasonable to consider \( H[A] \) a subtype of \( H[B] \), and we could declare

\[
\text{class } H[-T]() \{ \\
    \text{def } mthd(x: T) = 99 \}
\]

Limits on using variance declarations

• The use of \([+T]\) and \([-T]\) variance declarations is limited by how \( T \) is used in the class
• Scala tracks which type variables are used in \textit{covariant} positions, and which are used in \textit{contravariant} positions
• We are never \textit{required} to use a variance declaration
  – But we are forbidden from making a declaration opposite to actual usage
  – Which means that in some cases, we can make neither a covariant nor a contravariant declaration
7.4.2 Higher-kind type variables

7.4.3 Dependent types and the lambda cube

Where can we use types in Scala?

- Types of parameters, `val` or `var` declarations
- Arguments of generic classes/traits
- Fields of classes/traits
  - Possibly abstract
  - And then subsequently used in the types of other fields and methods
    ```scala
trait T {
  type A
  var stored: A
  def store(a: A) { stored = a; return }
  def process[B](f: A=>B) = f(stored)
}
```
  - Concrete subclasses must provide an actual type, just as they must provide method definitions
    ```scala
class C extends T {
  type A = String
  var stored ="
}
```

How does this not break variance?

```scala
trait T {
  type A
  var stored: A
  def store(a: A) { stored = a; return }
  def process[B](f: A=>B) = f(stored)
}
```

- Taking subclasses of `T` with different values for `A` would seem to conflict with variance
  - How can a subclass be a subtype if `A` is in a contravariant position?
- This isn’t an issue because `A` is not a simple parametrized type provided from outside `T`
  - For any instance `t:T`, the type `A` depends on the term `t`
  - `A` is another field of `t`, but it can be used as a type
    ```scala
class G {
  def f(t:T):Int = 4
  def g(t:T):t.A = t.stored
  def h(t:T):(t.A=>Unit) = { (x:t.A) => t.store(x) }
}
```
  - However we cannot say
    ```scala
def k(t:T, x:t.A) = t.store(x)
```
Constraints on abstract types

• We can give constraints on abstract type fields

```scala
trait T {
    type A <: G
    var stored: A
    def store(a: A) { stored = a; return }
    def process[B](f: A => B) = f(stored)
}
```

• Subtypes of T must provide compliant concrete types for A
  – Subtypes can also impose more strict constraints
• This can be useful when specifying the internal structure of a class
  – Specify the types of internal data structures used within the class, and abstract methods in terms of these types
  – Different implementations can use different types internally
  – We’ll see later how we can mix-and-match parts of classes so long as these internal interfaces are compatible

7.5 Implicit

7.5.1 Implicit

Notes to be posted

8 Further topics

Time allowing, we will study additional topics at the end of the semester. Any additional notes and exercises will be distributed separately.

9 Hints and answers for selected exercises

Exercise [5.43](p. 37) Use a helper function which performs the actual recursion. The helper can return both whether the invariants are satisfied, and the sum of the black nodes in the subtree.

Exercise [6.1](p. 72) 1. \((\lambda x.(\lambda y.xy))\lambda z.(z)\)
  2. \(xy(xz)\)