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/420-spring-17](http://cs.uwlax.edu/~jmaraist/420-spring-17)
  – 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:
  - $S \rightarrow \text{hello goodbye}$
- There may be other nonterminals, with rules that refer to each other
  - $S \rightarrow T \text{ goodbye}$
  - $T \rightarrow \text{hello}$
- Use a vertical bar to separate alternative choices, or give multiple rules for a nonterminal
  - $S \rightarrow T \text{ goodbye}$
    - $T \rightarrow \text{bonjour} | \text{gruessgott} | \text{hola}$
  - $S \rightarrow T \text{ goodbye}$
    - $T \rightarrow \text{bonjour}$
    - $T \rightarrow \text{gruessgott}$
    - $T \rightarrow \text{hola}$
- Extended BNF (EBNF) includes the Kleene star and plus notations
Exercise 2.11. Consider this grammar $G$, with start symbol $R$ [Sipser]:

$$
R \rightarrow RXR \mid S
$$

$$
S \rightarrow aTb \mid bTa
$$

$$
T \rightarrow TXT \mid X \mid \varepsilon
$$

$$
X \rightarrow a \mid b
$$

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 $a^*(ba^*ba^*)^*$. [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 generated 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, babbb, 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)

\[
\text{Expr} \rightarrow \text{Expr} \ ^ \ \text{Expr} \mid \text{Expr} \ * \ \text{Expr} \mid \text{Expr} \ / \ \text{Expr} \\
\mid \text{Expr} \ + \ \text{Expr} \mid \text{Expr} \ - \ \text{Expr} \mid - \ \text{Expr} \\
\mid ( \ \text{Expr} \ ) \mid 0 \mid 1 \mid \ldots
\]

• 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

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

• 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

Expr → Expr + Product | Expr - Product | Product
Product → Product * Power | Product / Power | Power
Power → Power ^ Basic | Basic
Basic → ( Expr ) | - Basic | 0 | 1 | ...

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.

Expr → Expr + Product | Expr - Product | Product
Product → Product * Power | Product / Power | Power
Power → Power ^ Basic | Basic
Basic → ( Expr ) | - Basic | 0 | 1 | ...

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
    Prev FP
    z 2

Frame for g
    Prev FP
    x 1
    y 3

Top of stack
```

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?

```plaintext
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

```
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.

Then $f$ 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;
}
```

```java
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 $y$
- $z$ 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
```

```
\[
\begin{array}{c}
\text{main} \\
\text{Prev FP} \\
\end{array}
\quad \Rightarrow 
\begin{array}{c}
\text{f} \\
\text{Prev FP} \\
\end{array}
\quad \Rightarrow 
\begin{array}{c}
\text{main} \\
\text{Prev FP} \\
\end{array}
\]
```

- 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.

Exercise 3.9. Trace the evaluation of this main routine under both call-by-reference and call-by-copy-in/copy-out parameter-passing semantics.

```plaintext
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

```shell
> 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

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:

V result;
if (map.containsKey(k)) {
    result = map.get(k);
} else {
    result = EXPR;
    map.put(k, result);
}

Scala's equivalent to HashMap includes an extra method where the second parameter is call-by-name (indicated by the =>):
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 3.14159265358979323846264338327950L
#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

```
(defmacro 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

```
(defmacro swap (x y)
  (let ((tmp (gensym)))
    '(let ((,tmp ,x))
      (setf ,x ,y
           ,y ,tmp)))
)
```

```
(defmacro 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

Exercise 3.10. Scott, Exercise 3.23
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

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;
}
```

Most (but not all) languages say that these are structurally equivalent.

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
- lsquare
- 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)
- [(False, '0'), (True, '1')]
- [(False, True), ('0', '1')]
- ([False, True], ['0', '1'])
- ([False, '0'], [True, '1'])
- [tail, init, reverse]

Exercise 5.6.  Write Haskell definitions which have the following types.

- [(Int,Int)]
- Int -> Int -> Bool -> Int
- Char -> (Char, Char)
- Int -> (Int -> Int) -> 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.

- second xs = head (tail xs)
- swap (x,y) = (y,x)
- pair x y = (x,y)
- double x = x*2
- 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,

\[
\text{showAreaOfCircle 12.3} \\
\quad \Rightarrow \ "The area of a circle with radius 12.3 cm is about 475.291552615999 cm^2"
\]

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,

\[
\text{sort2 :: Ord a =\rightarrow a =\rightarrow (a, a)}
\]

which accepts two Int values as arguments and returns them as a sorted pair, so that \( \text{sort2 5 3} \) 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

\[
\text{isLower :: Char \rightarrow Bool}
\]

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 \( 'a' \leq ch \) and \( ch \leq 'z'. \) 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]->[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,

\[
\text{mangle :: String \rightarrow String}
\]

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:

\[
\text{mangle "Hello" \Rightarrow " elloH"}
\]
\[
\text{mangle "I" \Rightarrow "I"}
\]
\[
\text{mangle " " \Rightarrow " "}
\]
Exercise 5.17. [Keller and Chakravarty] Write a module Divider with a function dividedBy which implements division on Int.

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

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.

\[ \text{dividedBy 5 10} \Rightarrow 2 \]
\[ \text{dividedBy 5 8} \Rightarrow 1 \]
\[ \text{dividedBy 3 10} \Rightarrow 3 \]

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

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

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

\[ \text{import Prelude hiding (length)} \]

to your module.

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

\[ \text{halve} : \text{[a]} \rightarrow (\text{[a]}, \text{[a]}) \]

Exercise 5.20. [Hutton Ex. 4.8.2, with solution] Define a function third.

\[ \text{third} : \text{[a]} \rightarrow \text{a} \]

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

Exercise 5.21. Write the function lastItem, which returns the last item in a list

Exercise 5.22. Write the function lastButOne, which returns the next-to-last item in a list

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

\[ \text{countOdds [1, 6, 9, 14, 16, 22]} = 2 \]

Hint: You can use the Prelude function \text{odd} :: \text{Int} \rightarrow \text{Bool}, which tests whether a number is odd.

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

\[ \text{removeOdd [1, 4, 5, 7, 10]} = [4, 10] \]

Exercise 5.25. Write the function isPalendrome, which checks if a list is a palendrome, the same backwards as forwards

Exercise 5.26. Write a module NeighborDups exporting a function noNeighborDups, which returns a list with consecutive duplicates removed
Exercise 5.27.  Write the function \texttt{lengthEncode}, for example,

\begin{verbatim}
lengthEncode "Aaabbcdddeeabb"
===> [ (1,'A'), (2,'a'), (3,'b'), (1,'c'),
        (2,'d'), (3,'e'), (1,'a'), (2,'b') ]
\end{verbatim}

Exercise 5.28.  Write the function \texttt{lengthDecode}, opposite of the above

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

Exercise 5.30.  Consider these declarations:

\begin{verbatim}
infixl 5 'test1'
infixl 7 'test2'
\end{verbatim}

Complete the definition of \texttt{test1} and \texttt{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 \texttt{test1} and \texttt{test2} behave differently with respect to each other? In a series of several applications of each?

Vary the declarations to use \texttt{infixr} and \texttt{infix} instead of \texttt{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

Exercise 5.31.  [Keller and Chakravarty] Write a function which, given a day, returns the data constructor representing the following day:

\begin{verbatim}
nextDay :: Day \rightarrow Day
\end{verbatim}

Use the definition of \texttt{Day} from this page.

Exercise 5.32.  [Thompson] Define a type \texttt{Month} as an algebraic type for the twelve months (use the full name of the month as constructors). Write a function \texttt{monthSeason} which maps a month to its member of the type \texttt{Season},

\begin{verbatim}
data Season = Winter | Spring | Summer | Fall
\end{verbatim}

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

\begin{verbatim}
data Shape = Circle Float
            | Rectangle Float Float
\end{verbatim}

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

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

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

3. Add cases to \texttt{area} and \texttt{perimeter} for \texttt{Triangle}. 

30
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 \texttt{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 \texttt{head} and \texttt{tail},

\begin{verbatim}
head :: [a] -> a
tail :: [a] -> [a]
\end{verbatim}

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

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

Exercise 5.36. [Keller and Chakravarty] Write a function \texttt{myLength}

\texttt{myLength :: [a] -> Int}

that, given a list \texttt{l}, returns the same result as \texttt{length l}. However, implement \texttt{myLength} without any explicit pattern matching on lists; instead, use the function \texttt{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

\begin{verbatim}
*Prelude> [ 2*x | x <- [1,2,3] ]
[2,4,6]
*Prelude> [(x,y) | x <- [1,2,3], y <-['a','b','c'] ]
[(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b'),
 (2,'c'),(3,'a'),(3,'b'),(3,'c')]
*Prelude> [ x | x <- [1..10], x 'mod' 3 == 1 ]
[1,4,7,10]
\end{verbatim}

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:

\begin{verbatim}
dividesByThree :: [Int] -> [Int]
dividesByThree xs = [ x | x <- xs 'mod' 3 == 0 ]
\end{verbatim}

Exercise 5.38. Use list comprehension notation to write the function \texttt{capVowelsFirst} that takes a list of strings, and return a list containing only the elements of the argument which start with a capital vowel.
6 Further topics

Time allowing, we will study additional topics at the end of the semester. Any additional notes and exercises will be distributed separately.