CS 320: Principles of Programming Languages

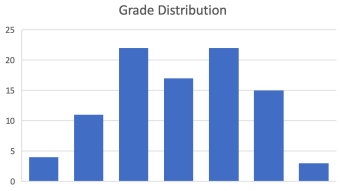
Mark P Jones
Portland State University

Spring 2019

Midterm Review

TL; DR; How did I do?

- In theory: 48 points available on the test (6 questions x 8 points)
- In practice: not expected to answer everything; graded out of 40
- Multiply raw score by 2.5 to obtain a percentage shown on D2L
- Did I pass the midterm? (~ percentage>= 40, raw>=16)
- Did I get an A? (~ percentage>=70, raw>=28)
- Can I still pass the class? YES



1) Syntax and Semantics:

a) For each of the following items, explain what the term refers to and why it would be important and/or relevant in the design of a practical programming language:

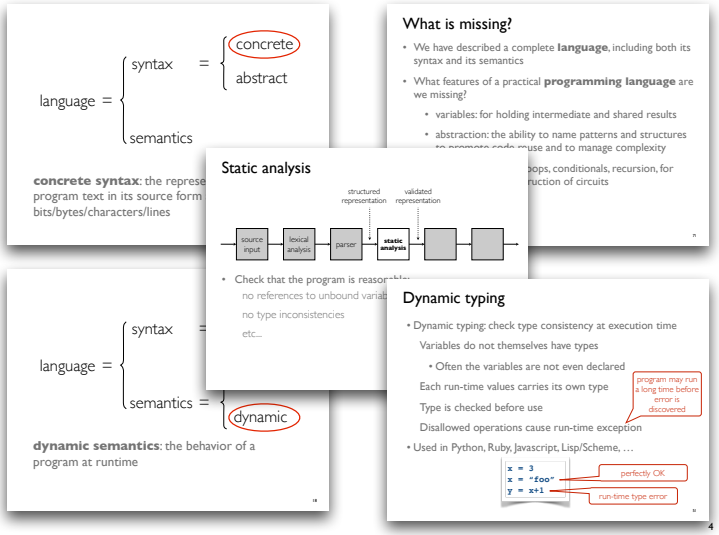
i) concrete syntax: **(1)**

ii) abstraction: **(1)**

iii) dynamic semantics: **(1)**

iv) static analysis: **(1)**

v) dynamic typing: **(1)**



1) ... continued

a) For each of the following items, explain <u>what the term refers to</u> and why it would be important and/or relevant in the design of a practical programming language:

i) concrete syntax: **(1)**
the representation of a program text in its source form

ii) abstraction: **(1)**
the ability to name patterns and structures

iii) dynamic semantics: **(1)**
the behavior of a program at runtime

iv) static analysis: **(1)**
validate abstract syntax/check that the program is reasonable

v) dynamic typing: **(1)**
check type consistency at execution time

1) ... continued

a) For each of the following items, explain what the term refers to and <u>why it would be important and/or relevant in the design of a practical programming language</u>:

i) concrete syntax: **(1)**
so we know how to write programs

ii) abstraction: **(1)**
to promote reuse and manage complexity

iii) dynamic semantics: **(1)**
so we can predict how programs will behave

iv) static analysis: **(1)**
to detect program errors at compile-time

v) dynamic typing: **(1)**
to increase flexibility (with potential cost of run-time errors)

1) ... continued

b) Construct a regular expression that describes a plausible syntax for floating point literals.

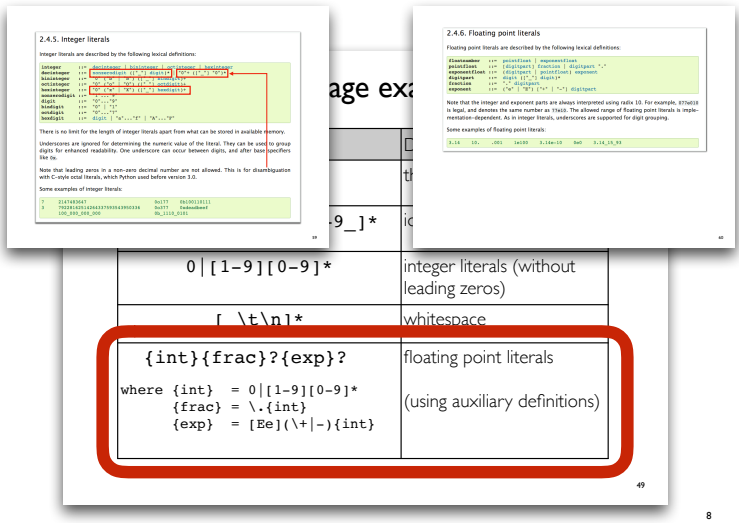
Any reasonable syntax will be accepted, so long as it allows

i) for optional fractional and exponent parts; and

ii) for the ability to include a single underscore between any adjacent pair of digits (for the purpose of improving readability for long numbers).

Your use of regular expressions should be limited to the notations shown in the course slides.

Note, in particular, that you are encouraged (but not required) to use definitions of the form "`name=regexp`" in your answer and then to use "`{name}`" as an abbreviation for the specified "`regexp`" in subsequent parts of your answer. **(3)**



1) ... continued

b) Construct a regular expression that describes a plausible syntax for floating point literals.

underscores à la Python

```
{int}{frac}?{exp}?
  where int  = 0|[1-9](_? [0-9])*
        frac = \. {int}
        exp  = [Ee](+|-)?{int}
```


2) Programming Language Fundamentals:

a) Suppose that A is a set of symbols. Explain what (if anything) is meant by each of the following:

i) L is a *language* over A **(1)**

L is a subset of A* (i.e., a set of finite strings of symbols from A)

ii) L is a *regular language* over A **(1)**

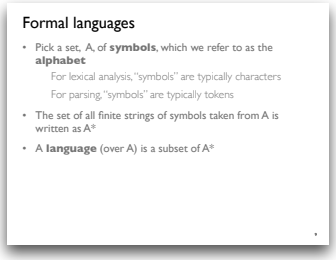
L can be recognized by a DFA whose transitions are labeled with symbols from A

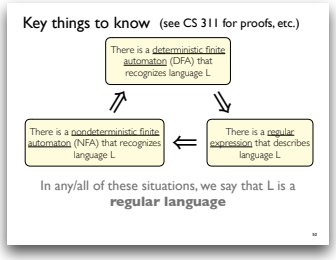
iii) L is a *context-free language* over A **(1)**

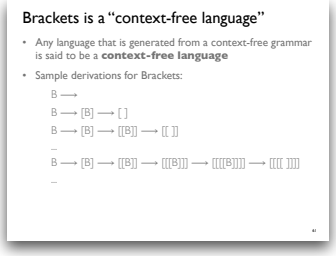
L is generated from a context-free grammar with terminals A

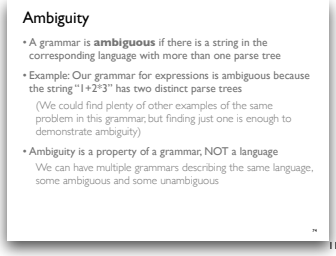
iv) L is an *ambiguous language* over A **(1)**

Nonsense: ambiguity is a property of grammars, not languages!









2) ... continued

b) The following shows a fragment from the Haskell implementation of a parser for the "`Prop`" language (with prefix notation syntax) that we have explored in lectures and labs:

```
parseProp :: [Token] -> (Prop, [Token])
parseProp (TAND : ts)
   = case parseProp ts of
      (p, ts1) -> case parseProp ts1 of
                    (q, ts2) -> (AND p q, ts2)
```

- Explain (i) what is meant by the type signature declaration in the first line ...

  The type signature indicates that the parser is a function that takes a list of tokens as inputs and returns a Prop AST and a list of unused tokens

+ Lab 3 (+ 1, 2, and 4)

2) ... continued

b) The following shows a fragment from the Haskell implementation of a parser for the "Prop" language (with prefix notation syntax) that we have explored in lectures and labs:

```
parseProp :: [Token] -> (Prop, [Token])
parseProp (TAND : ts)
   = case parseProp ts of
      (p, ts1) -> case parseProp ts1 of
                    (q, ts2) -> (AND p q, ts2)
```

- ... and (ii) what is the distinction between the symbols "TAND" and "AND" used here. **(2)**
  - TAND represents the "AND" keyword/token
  - AND is a constructor for the Prop AST type

2) ... continued

Using text and/or a suitably labelled diagram, explain briefly how the equation for "parseProp" would work as part of a complete parser implementation. **(2)**

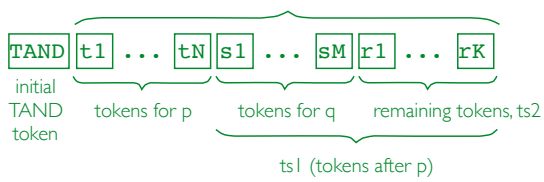
```
parseProp :: [Token] -> (Prop, [Token])
parseProp (TAND : ts)
   = case parseProp ts of
      (p, ts1) -> case parseProp ts1 of
                    (q, ts2) -> (AND p q, ts2)
```

- The first call to **parseProp** extracts a prop **p** from the tokens **ts** that come after the initial **TAND**, but does not use the tokens in **ts1**.
- The second call extracts a prop **q** from **ts1** but does not use the tokens in **ts2**.
- The overall result is to parse the prop **AND p q** without using the tokens in **ts2**.

2) ... continued

Using text and/or a suitably labelled diagram, explain briefly how the equation for "parseProp" would work as part of a complete parser implementation. **(2)**

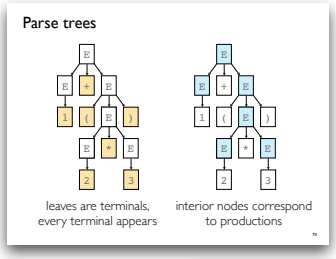
```
parseProp :: [Token] -> (Prop, [Token])
parseProp (TAND : ts)
   = case parseProp ts of
      (p, ts1) -> case parseProp ts1 of
                    (q, ts2) -> (AND p q, ts2)
```

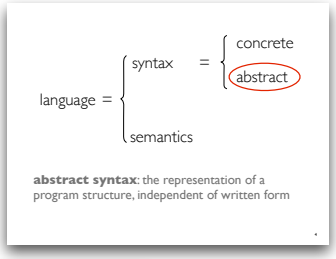


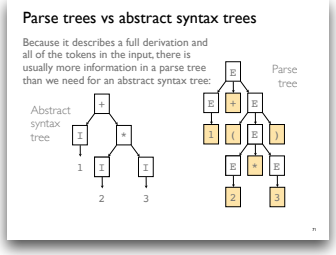
3) Parse Trees and Context Free Grammars:

Explain what is meant by parse trees and how they differ from abstract syntax trees. **(2)**

- A parse tree reflects the structure of a derivation with an interior node for each derivation step and a leaf node for every terminal in the input
- An abstract syntax tree captures the structure of a program or expression that is independent of written form (doesn't include all tokens, parentheses, punctuation, etc... and other grammar details)







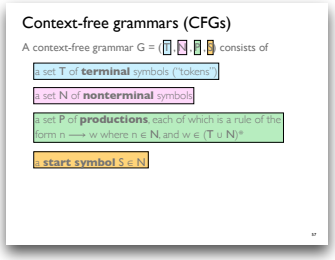
3) ... continued

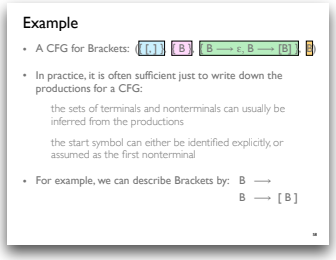
Consider a grammar for a variant of the Prop language that uses infix syntax for operators:

P ⟶ TRUE P ⟶ FALSE P ⟶ VAR P ⟶ (P)
P ⟶ NOT P P ⟶ P AND P P ⟶ P OR P

What are the sets of *terminals*, *nonterminals*, and *productions* for this grammar? **(1)**

- terminals: { TRUE, FALSE, VAR, (,), AND, OR, NOT }
- nonterminals: { P }
- productions: the seven rules shown above





Lab 2, HW2, Grammar Toolkit

3) ... continued

Identify a sentence in the language for this grammar that begins with NOT and that has two distinct parse trees. Draw diagrams to show the structure of those two parse trees. **(2)**

```
NOT VAR AND VAR                    (c.f., a | b*)
```

```
      P                              P
    /   \                          / | \
NOT     P                      P  AND  P
      / | \                    /  \      |
    P  AND  P              NOT    P     VAR
    |       |                      |
   VAR     VAR                    VAR
```


3) ... continued

Give a new grammar for the same language in which NOT has higher precedence than AND, which in turn has higher precedence that OR, and in which OR and AND both group to the left. Explain briefly how these properties are achieved. **(3)**

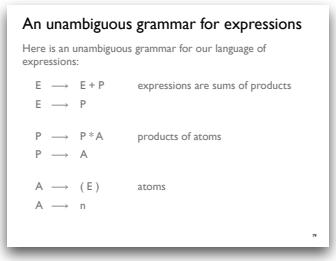
P ⟶ P OR C	P ⟶ C
C ⟶ C AND N	C ⟶ N
N ⟶ NOT N	N ⟶ A
A ⟶ TRUE	A ⟶ FALSE
A ⟶ VAR	A ⟶ (P)

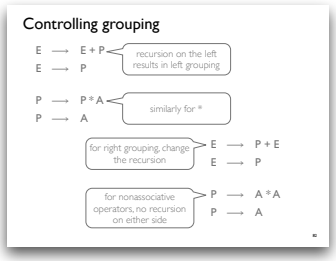
left recursion for P ⇒ OR groups to left

left recursion for C ⇒ AND groups to left

Every P is an OR of Cs, each of which is an AND of Ns, each of which is a (possibly negated) atom.

prec(OR) < prec(AND) < prec(NOT)



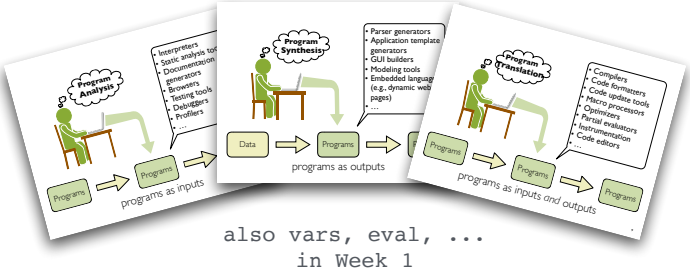


Lab 3 (ParseInfix in-class/video)
+ Homework 2, Q3

4) Programs As Data:

a) Why is it useful to think of programs as data, and not just as executables that can run on a computer? **(1)**

So that we can write interesting programs that compute with programs (for the purposes of analysis, synthesis, or translation)

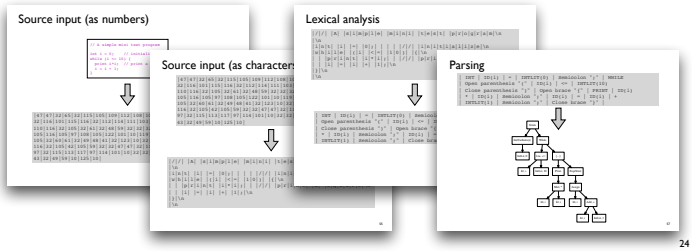


also vars, eval, ...
in Week 1

4) ... continued

b) Describe two different ways of representing programs as data, and comment on their relative strengths and weaknesses. **(2)**

1) ASTs: structured, good for computing over programs
2) character sequences: flat, good for programmers to read/write
3) [number/byte sequences: flat, good for executing programs]
4) [lists of tokens: flat, good for input to parser]



4) ... continued

c) In the context of systems that treat programs as data, explain what is meant by the terms *front end*, *middle end*, and *back end* ...

- a front end reads source programs and captures abstract syntax
- a middle end analyzes and manipulates the AST structures
- a back end generates output from AST structures

... and give two reasons why it is useful to think of these as separate components. **(2)**

- modular construction (easier to write, debug, test, maintain, ...)
- can reuse across multiple tools for the same language

General building blocks

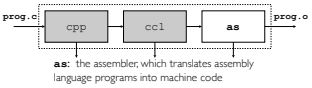
- A **front end** reads source programs (e.g., flat text files) and captures the corresponding abstract syntax in a collection of data structures (e.g., trees, graphs, arrays, ...)
- A **middle end** analyzes and manipulates the abstract syntax data structures of a program
- A **back end** generates output(e.g., a flat, binary executable file) from the abstract syntax data structures of a program
- Substantial parts of these components can be shared by multiple tools
  - Example: the `g++` and `gcc` compilers (for C++ and C, resp.) use the same middle and back end components
  - Example: the `ghc` (compiler) and `ghci` (interpreter) for Haskell use the same front and middle end components

Modularity

- Modularity is all about building large systems from collections of smaller components
- Modular implementations can be easier to write, test, debug, understand, and maintain than monolithic implementations
- For example:
  - Components can be developed independently
  - Some components can be reused in other contexts
  - Some components may even be useful as standalone tools

Combining compilers

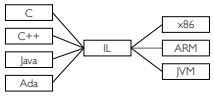
- The classic Unix C compiler, `cc`, is implemented by a pipeline of compilers:



as: the assembler, which translates assembly language programs into machine code

An intermediate language

- Alternatively: design a general purpose, shared "intermediate language":



- Now we only have n front ends and m back ends to write!
- The biggest challenge is to find an intermediate language that is general enough to accommodate a wide range of languages and machine types

4) ... continued

d) Identify the sequence of tokens in the following code fragment, taken from a larger Haskell program. Individual tokens can be represented by a suitably chosen name (your choice) followed, where necessary, by an appropriate attribute (e.g., like "`TOPEN`" and "`TVAR s`" in the lexer for `Prop`). What additional kinds of information might a Haskell parser need to make sense of this sequence of tokens? **(3)**

```
Calculate the area of a circle:
> area r    = pi * r * r  -- area = pi r^2
Approx value for pi:
> pi = 3.1415 :: Double
```


4) ... continued

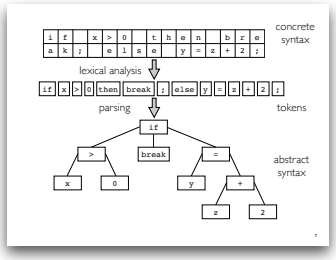
d) Identify the sequence of tokens in the following code fragment:

```
Calculate the area of a circle:
> area r    = pi * r * r  -- area = pi r^2
Approx value for pi:
> pi = 3.1415 :: Double
```

comments are *not* tokens

```
[TID "area", TID "r", TEQ,  TID "pi", TOP "*",
 TID "r", TOP "*", TID "r", TID "pi", TEQ,
 TFLOAT 3.1415, THASTYPE, TID "Double"]
```

A Haskell parser would need information about indentation so that it can tell where one definition ends and the next begins (e.g., here).



+ Lab 3: lexProp and parseProp

Homework 3

5) Types:

a) Explain what is meant by each of the following, including a corresponding example from a programming language of your choice in each case. [Notes: List only types, not values. Identify the programming languages that you are using if it is not immediately obvious. You may choose (but are not required to use) a different language for each part.] **(4)**

i) a primitive type
   cannot be broken into smaller parts/atomic/builtin — `Int`

ii) a sum type
   capture alternatives — `Bool`

iii) a product type
   groups multiple values into a composite — `(Int,Bool)`

iv) a recursive type
   definition is self-referential — `Tree`

Primitive types

- Primitive types cannot be broken down into smaller constituents
- A programming language will typically provide
  - literal syntax for writing/constructing values of primitive types
  - built-in operators and/or statement forms for manipulating values of primitive types
- Primitive types often (but not always) reflect basic machine-level types such as integers, floating point numbers, characters, etc...
  - The numeric types are superficially familiar from mathematics, but may have limited range or precision

Sums/variants

- **Variants** or **sum types** allow us to capture alternatives
- The sum type (T + S) contains all the values of type T and all the values of type S, with some way to distinguish between them
- Example: in Haskell, the type `Either A B` is a sum type that contains all the values of the form `Left x` (if x∈A) and all the values of the form `Right y` (if y∈B)
- The type `Either Bool Char` contains:
  `Left True`, `Right 'a'`, `Left False`, `Right 'z'`, ...
- We sometimes refer to `Left` and `Right` here as **tags**

Products

- A **product type** allows us to package multiple values up as a single, composite (or aggregate) value
- Different programming languages support different notions of products, trading off such features as:
  - Access methods: named, numbered, positional, ...
  - Accuracy of typing: heterogeneous vs. homogeneous (e.g., type `list` in Python vs `T[]` in Java)
  - ...

Recursion

- Many interesting types can be defined once we allow recursive definitions
- For example, a list of integers is either:
  Empty; or else
  Nonempty, with an integer at its head and a (shorter) list of numbers as its tail.
- In the notations we have seen so far:

```
data IntList = Empty | NonEmpty Int IntList   // Haskell
struct IntList { int head; struct IntList* tail; }; // C
abstract class IntList {}                     // Java
class Empty    extends IntList {}
class NonEmpty extends IntList { int head; IntList tail; }
```

31

5) ... continued

b) In what ways does it make sense to think of a function type as a product type?

A function of type A ⟶ B behaves like an array of B values that are indexed by values of type A.

In what ways do function types differ from other forms of product type? **(2)**

- functions typically compute values on demand rather than storing computed results.
- functions can have infinite domains/index types.

32

Functions (exponentials)

- Conceptually, a function of type (A⟶B) that takes an argument of type A and returns a result of type B is like an array in which:
  - Components are all indexed by values of type A
  - All components store values of the same type B
  - Values are computed on demand rather than stored
- Functions can be used to build data structures:

```
type IntSet = Int -> Bool

intersect     :: IntSet -> IntSet -> IntSet
intersect s t  = \n -> s n && t n
```

Lambda expressions

Lambda expressions (anonymous functions) provide a way to write functions without giving them a name.

Haskell	`\x -> x + 1`
LISP	`(lambda (x) (+ x 1))`
Python	`lambda x: x + 1`

Java	`(x) -> x + 1`
C++ 11	`[] (int x) -> int { return x + 1; }`
Scala	`x => x + 1`

Lambda calculus (Alonzo Church, 1930s)

- A simple language for describing calculations with functions:

E ⟶	x	variables
E ⟶	c	constants
E ⟶	E E	function application
E ⟶	λx . E	lambda abstraction
E ⟶	(E)	parentheses

- Examples:
  - The identity function (λx.x)
  - The successor function (λx.x+1)
  - The doubling function (λx.x*2)
  - Function composition (λf.λg.λx.f(g x))
  - ("x+1" is "syntactic sugar" for "plus x 1", where plus is a constant representing the function that adds two numbers)

Evaluation in the lambda calculus

- The fundamental rule:
  - "beta conversion" ("function application"):
    (λx. E) E' = [E'/x] E
- Examples (of beta reduction):
  (λf.λg.λx.f(g x)) (λx.x+1) (λz.2*z) n
  = (λf.λg.λx.f(g x)) (λy.y+1) (λz.2*z) n ← renaming of bound variables
  = (λg.λx. (λy.y+1) (g x)) (λz.2*z) n
  = (λg.λx. g x+1) (λz.2*z) n
  = (λx. (λz.2*z) x+1) n
  = (λx. (2*x)+1) n
  = ((2*n)+1)

33

5) ... continued

c) Suppose that f = (λx. x+2) and g = (λy. 3*y). Stating and using the definition of "beta conversion" to justify your answer, calculate the value of g (f 4). **(2)**

Beta conversion: (λx.E) E' = [E'/x] E

Using Haskell: `f = \x -> x+2, g = \y -> 3*y`

```
g (f 4)  =  g (4 + 2)
         =  g 6
         =  3 * 6
         =  18
```

34

5) ... continued

c) Suppose that f = (λx. x+2) and g = (λy. 3*y). Stating and using the definition of "beta conversion" to justify your answer, calculate the value of g (f 4). **(2)**

Beta conversion: (λx.E) E' = [E'/x] E

g (f 4) = g ((λx.x + 2) 4)	definition of f
= g ([4/x](x + 2))	beta conversion
= g (4 + 2)	substitution
= g 6	arithmetic
= (λy. 3 * y) 6	definition of g
= [6/y] (3 * y)	beta conversion
= 3 * 6	substitution
= 18	arithmetic

35

6) Programming Paradigms:

a) For each of the following, give a single sentence, high-level summary of the corresponding paradigm and explain how a program entry point might be described:

i) procedural programming: **(1)**

ii) object-oriented programming: **(1)**

iii) functional programming: **(1)**

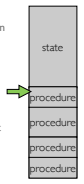
36

Summary: Language paradigms

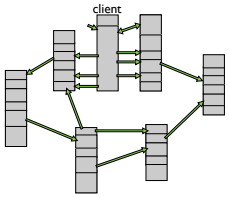
- Languages can be (partially) classified according to paradigms such as functional, procedural, object-oriented, …
- Functional programming emphasizes the evaluation of expressions and the use of functions as first-class values
- Procedural programming views programs as collections of procedures with associated state
- Object-oriented programming views programs as collections of stateful objects that interact by sending messages
- There are no firm boundaries: for example, you can do functional programming in a procedural language
- Many modern language designs are multi-paradigm

Procedural programming

- A form of imperative programming in which the code is structured as a collection of procedure/subroutine definitions
- Allows abstraction, reuse, and modular construction of software
- One particular procedure is chosen as the **entry point** (e.g., "main")
- Programs run by executing the entry point procedure
- Procedures "call" other procedures in a LIFO (i.e., stack-like) manner



Using multiple ~~libraries~~ objects



Computing by calculating

- An old dictionary definition for "computer":
  "a person who makes calculations, especially with a calculating machine"
- A basis for functional programming:
  - Using a wide range of types of values, from numbers and strings to tuples, lists, trees, functions, and more
  - Scaling to large problems

37

6) ... continued

i) procedural programming: **(1)**
computations are described by collections of procedures that can call on one another and that operate on a shared program state.

ii) object-oriented programming: **(1)**
computations are described in terms of stateful objects that communicate with one another by sending messages/calling each other's methods.

iii) functional programming: **(1)**
computations are described by the evaluation of expressions (``computing by calculating'').

38

6) ... continued

i) procedural programming: **(1)**
The entry point is a specific procedure in the program

ii) object-oriented programming: **(1)**
The entry point might be described by a combination, either of a specific object and method, or by a specific class and static method

iii) functional programming: **(1)**
The entry point is the expression to evaluate

39

6) ... continued

b) Ignoring details of syntax, and based only on material that we have covered in this course, what are the most important distinctions:

i) between lists in Haskell and lists in Python: **(2)**

1) all elements of a list in Haskell must have same type

2) lists in Python are immutable

ii) between functions in Haskell and functions in Python: **(2)**

1) functions in Haskell are pure (no side-effects)

2) functions in Haskell can be partially applied

3) functions in Haskell are statically typed (known argument and return types, modulo polymorphism)

40

6) ... continued

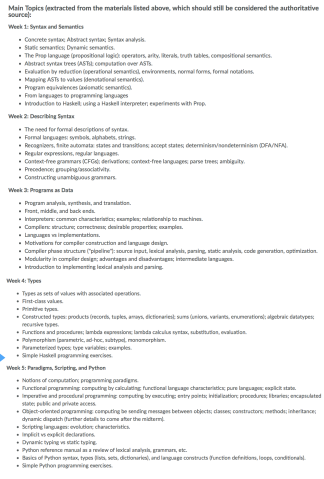
c) Why might it be useful for a person to learn about new programming paradigms if they expect or plan to spend their career developing deep expertise in a single, widely-used programming language (such as C, C++, Java, Python, Haskell, or Rust, for example)? **(1)**

- To reinforce your understanding of fundamental concepts, highlighting details that you might otherwise take for granted
- To discover fundamentally new ways to think about programming and problem solving
- To prepare for learning new languages as your career evolves

41

How to Study?

- Everyone is different; find what works for you ...
- Use the resources provided:
  - Slides, Lab materials
  - HW/Midterm sample solutions
  - Notes on Haskell
  - [D2L discussions]
- Just "reading" may not be enough
- Prepare your own topic list
- Handwriting recommended!
- Allocate time for review
- Ask questions ...



Main Topics (extracted from the materials listed above, which should still be considered the authoritative source):

Week 1: Syntax and Semantics

- Concrete syntax; Abstract syntax; Syntax analysis.
- Static semantics; Dynamic semantics.
- The Prop language (propositional logic): operators, arity, literals, truth tables, compositional semantics.
- Abstract syntax trees (ASTs); computation over ASTs.
- Evaluation by reduction (operational semantics), environments, normal forms, formal notations.
- Mapping ASTs to values (denotational semantics).
- Program equivalences (axiomatic semantics).
- From languages to programming languages
- Introduction to Haskell; using a Haskell interpreter; experiments with Prop.

Week 2: Describing Syntax

- The need for formal descriptions of syntax.
- Formal languages: symbols, alphabets, strings.
- Recognizers, finite automata: states and transitions; accept states; determinism/nondeterminism (DFA/NFA).
- Regular expressions, regular languages.
- Context-free grammars (CFGs); derivations; context-free languages; parse trees; ambiguity.
- Precedence; grouping/associativity.
- Constructing unambiguous grammars.

Week 3: Programs as Data

- Program analysis, synthesis, and translation.
- Front, middle, and back ends.
- Interpreters: common characteristics; examples; relationship to machines.
- Compilers: structure; correctness; desirable properties; examples.
- Languages vs implementations.
- Motivations for compiler construction and language design.
- Compiler phase structure ("pipeline"): source input, lexical analysis, parsing, static analysis, code generation, optimization.
- Modularity in compiler design; advantages and disadvantages; intermediate languages.
- Introduction to implementing lexical analysis and parsing.

Week 4: Types

- Types as sets of values with associated operations.
- First-class values.
- Primitive types.
- Constructed types: products (records, tuples, arrays, dictionaries); sums (unions, variants, enumerations); algebraic datatypes; recursive types.
- Functions and procedures; lambda expressions; lambda calculus syntax, substitution, evaluation.
- Polymorphism (parametric, ad-hoc, subtype), monomorphism.
- Parameterized types; type variables; examples.
- Simple Haskell programming exercises.

Week 5: Paradigms, Scripting, and Python

- Notions of computation; programming paradigms.
- Functional programming: computing by calculating; functional language characteristics; pure languages; explicit state.
- Imperative and procedural programming: computing by executing; entry points; initialization; procedures; libraries; encapsulated state; public and private access.
- Object-oriented programming: computing be sending messages between objects; classes; constructors; methods; inheritance; dynamic dispatch (further details to come after the midterm).
- Scripting languages: evolution; characteristics.
- Implicit vs explicit declarations.
- Dynamic typing vs static typing.
- Python reference manual as a review of lexical analysis, grammars, etc.
- Basics of Python syntax, types (lists, sets, dictionaries), and language constructs (function definitions, loops, conditionals).
- Simple Python programming exercises.

42

CS 320 Objectives

Upon the successful completion of this course, students will be able to:

1. Explain the distinction between language syntax and semantics ✓
2. Describe the similarities and differences between interpreters and compilers. ✓
3. Explain the phase structure of a typical compiler and the role of each phase. ✓
4. Use regular expressions and context-free grammars to describe the syntax of simple programming languages. ✓

CS 320 Objectives, continued

5. Illustrate the features and characteristics of different programming paradigms, including procedural, functional, and object-oriented programming. ✓
6. Explain the concepts of binding, scope, block structure, and lifetime, and apply them to resolving variable uses to their binding sites in a variety of languages. ?
7. Describe and apply the basic concepts of type systems, including primitive types, compound, and recursive types, abstract data types, and type equivalence models. ✓
8. Describe the strengths and limitations of static and dynamic typing disciplines. ✓

CS 320 Objectives, continued

9. Describe and apply the basic concepts of data abstraction, encapsulation, object-oriented classes, and modules. ?
10. Explain basic approaches and applications for the formalization of programming language semantics. ?

How to succeed in this class (1)

- Attend lectures and pay attention in class.
- Allocate time for study (~8 hours a week outside class)
- Check the D2L site on a regular basis.
- Start your assignments early so that you have plenty of time to ask for help if you need it.
- Seek out, read, and review other resources.
- Study for the tests (and make sure you get enough sleep, etc. before the tests).
- Do not attempt to cheat!
- Tell us if there are things we can do to help you learn the material more effectively!

How to succeed in this class (2)

- If you don't understand something or don't know what is expected, ask for help.
- Asking questions is an essential part of learning.
- Don't be shy, ask away!
- If you have a question, chances are high that others are wondering the same thing; you'll be helping others!
- Don't worry about derailing the class; we'll take questions "offline" if necessary.
- And suggestions that might improve the class are always welcome too!
- Your feedback really can make a difference.

Any questions?