LSPU Self-paced Learning Module (SLM)

Course	**Bachelor of Science in Computer Science**
Sem/AY	**First Semester/2023-2024**
Module No.	**1**
Lesson Title	**Introduction to Programming Languages**
Week Duration	**2**
Date	August 28-September 01, 2023 September 4-8, 2023 September 11-15, 2023 September 18, 22, 2023
Description of the Lesson	This course provides students the fundamental features and concepts to different programming languages. Topics include overview of programming languages, Introduction to language translation, type systems, data and execution control, declaration and modularity, and syntax and semantics. Laboratory will be used to demonstrate each of the concepts using different programming languages.

Learning Outcomes

Intended Learning Outcomes	Students should be able to meet the following intended learning outcomes: • Understand fundamental programming concepts, such as variables, data types, operators, and control structures (e.g., loops and conditionals). • Analyze and write code in different programming languages, demonstrating an understanding of their syntax and rules. • Identify and fix common programming errors and bugs, using debugging techniques and tools. • Write code that is clear, well-organized, and properly documented to enhance readability and maintainability.
Targets/ Objectives	At the end of the lesson, students should be able to learn, practice and apply the ff: • Understand fundamental programming concepts, such as variables, data types, operators, and control structures (e.g., loops and conditionals). • Develop algorithmic thinking skills to solve problems systematically and logically. • Identify and fix common programming errors and bugs, using debugging techniques and tools. • Understand the concept of modular programming and create reusable functions or modules to promote code reusability.

Student Learning Strategies

Online Activities (Synchronous/ Asynchronous)

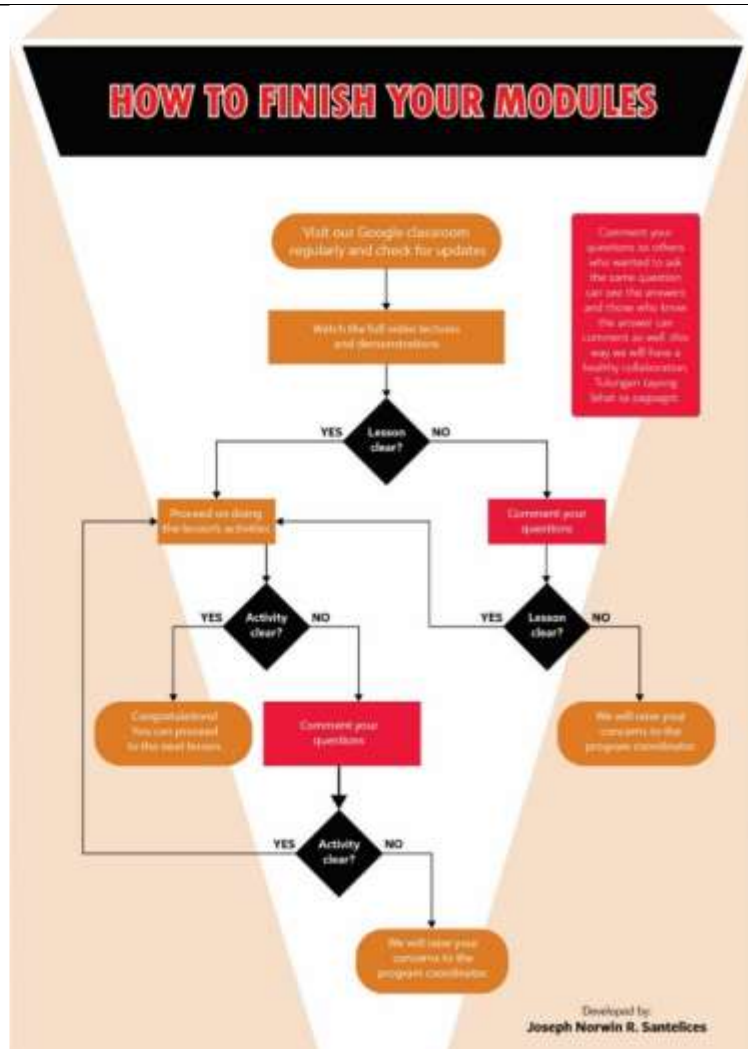
A. Online Discussion via Google Classroom.
You will be directed to join in the online classroom where you will have access to the video lectures, course materials, and evaluation tools 24/7. To have access on the Online Lectures, refer to this link: ___________________.

To ensure a healthy collaboration, you may post your inquiries on a particular topic and the instructor including all the members of the class can acknowledge the inquiry.

Monday	BSCS 3IS	Lecture	8:00-10:00
Tuesday	BSCS 3GV	Lecture	8:00-10:00
Wednesday	BSCS 3IS	Laboratory	7:00-10:00
Thursday	BSCS 3GV	Laboratory	10:00-1:00

(For further instructions, refer to your Google Classroom and see the schedule of activities for this module)

B. Learning Guide Questions:
Refer to the following diagram as your guide through the course.



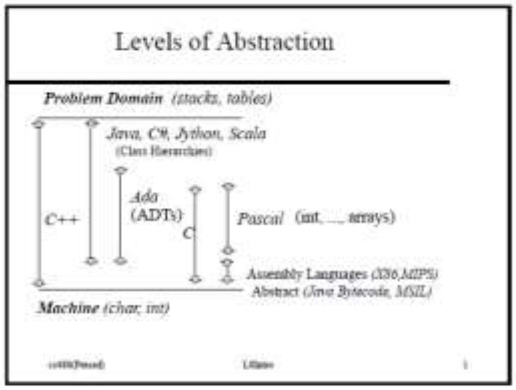
Note: *The insight that you will post on online discussion forum using Learning Management System (LMS) will receive additional scores in class participation*

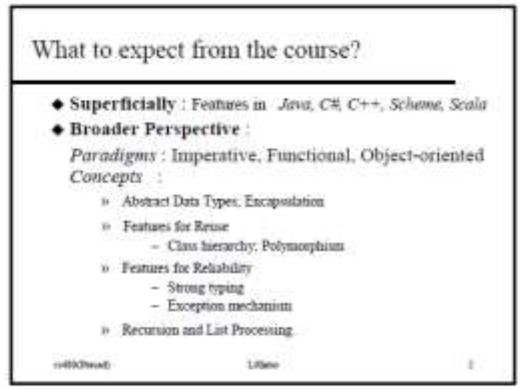
Offline Activities (e-Learning/Self-Paced)

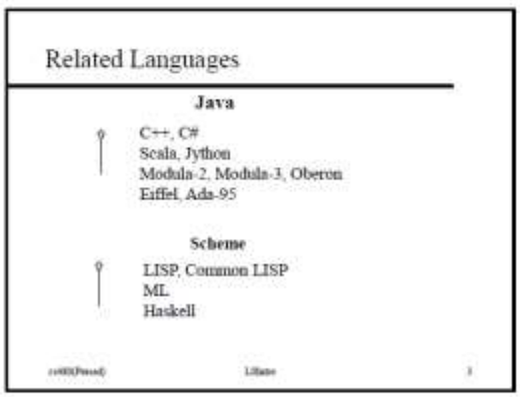
Introduction to Programming Languages

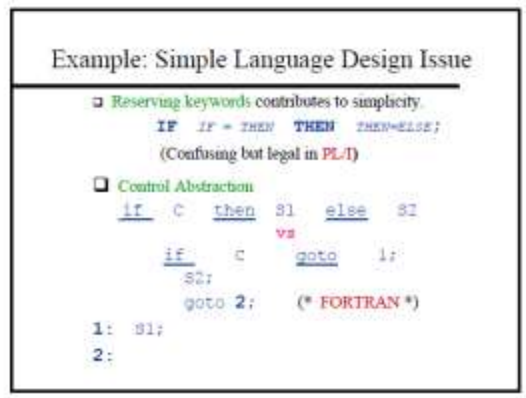
Programming languages are not very different from spoken languages. Learning any language requires an understanding of the building blocks and the grammar that govern the construction of statements in that language. This unit will serve as an introduction to programming languages, taking you through the history of programming languages. We will also learn about the various universal properties of all programming languages and identify distinct design features of each programming language. By the end of this unit, you will have a deeper understanding of what a programming language is and the ability to recognize the properties of programming languages.

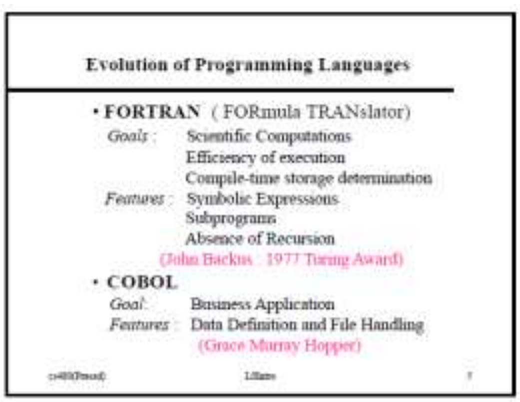
Evolution of Programming Languages

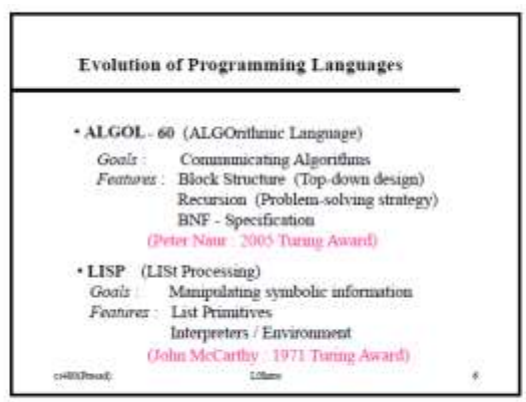












C.A.R Hoare On Algol-60

- Here is a language so far ahead of its time, that it was not only an improvement on its predecessors, but also on nearly all its successors.
- I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.

(C. A. R. Hoare : 1980 Turing Award)

Evolution of Programming Languages

- PL / 1

  FORTRAN + COBOL + SNOBOL+ ... + concurrency + ...

" When FORTRAN has been called infantile disorder, full PL /1, with its growth characteristics of a dangerous tumor, could turn out to be a fatal disease."

---- E. W. Dijkstra (1972 Turing Award Lecture)

Evolution of Programming Languages

- SIMULA (SIMULAtion LAnguage)

  Features : Data Abstraction, Class Hierarchies. (Inheritance)

  (O. J. Dahl, K. Nygaard : 2001 Turing Award)
- C

  Goal : Systems Programming

  Features : Coding language for Unix. Portability.

  (D. Richie and K. Thompson : 1983 Turing Award)

On C and C++

- C makes it easy to shoot yourself in the foot, C++ makes it harder, but when you do, it blows away your whole leg. -- Bjarne Stroustrup
- The last good thing written in C was Franz Schubert's Symphony number 9.
- C is quirky, flawed, and an enormous success. -- Dennis M. Ritchie.

Evolution of Programming Languages

- Pascal

  Goal : Structured Programming, Compiler writing.

  Features :
  - Rich set of data types for efficient algorithm design
    - E.g., Records, sets, ...
  - Variety of "readable" *single-entry single-exit* control structures
    - E.g., *for-loop, while-loop*, ...
  - Efficient Implementation
    - Recursive descent parsing

  (N. Wirth : 1984 Turing Award)

On Type System; Efficiency

- Type security is intended not so much to inspire programmers as to protect them from their own not inconsiderable frailties.
- More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity. -- William A. Wulf

Other Languages

- Functional
  - Common LISP, Scheme
  - ML, Haskell

  (Robin Milner : 1991 Turing Award (for ML))
- Logic
  - Prolog
- Object-oriented
  - Smalltalk, Eiffel, Java, C#

  (Alan Kay : 2003 Turing Award (for SmallTalk))
  - C++, Ada-95, CLU

  (Barbara Liskov: 2009 Turing Award (for CLU))
  - Modula-3, Oberon
- Application specific languages and tools

Modern Scripting Languages

- Multiparadigm Constructs = OOP style + Functional style

  (+ Ease of prototyping (Interpreter-based))
- *Examples*: Python, Ruby, PERL, PHP, ..., JPython, JRuby, ...CAML, F# Scala, ...

Lamdba Caculus

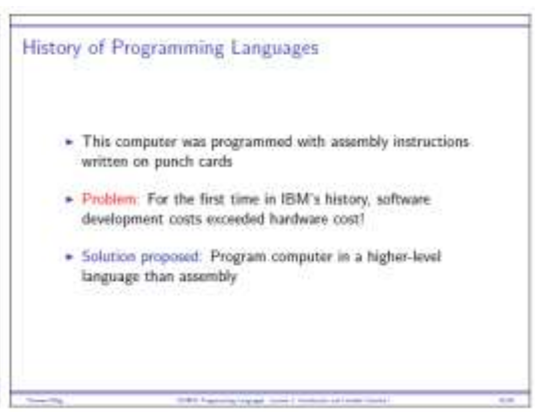
What is this course about?

This course is about programming languages.

- We will study different ways of specifying programs.
- We will learn how to give (precise) meaning to programs.
- We will see how to use programming languages to prevent run-time errors.
- We will explore these concepts in real-world languages.

Why should you take this course?

- Understanding programming languages means that you will be able to program in any existing or future programming language almost immediately.
- You will be able to choose the right language for the right problem.
- You will have techniques to give precise semantics to any string, not just programs.
- You will have a much easier time getting (and keeping) jobs.



FORTRAN I

- Enter John Backus
- Translation from higher-level language to assembly had already been tried before...
- And did not work out (at all)
- But team lead by John Backus produced first practical programming language called FORTRAN and a compiler to translate it to assembly

Impact of FORTRAN

- Within 2 years: 80% of programs written for the IBM 704 were written in FORTRAN
- This is even though FORTRAN I is a pretty awful language (by today's standards)
- After this: Almost all programming done in (increasingly) higher level languages
- Programming languages have greatly improved programmer productivity, enabling software that would never haver been possible otherwise

Language Goals:

- In the beginning, overarching concern when developing languages was performance
- As hardware got faster, many different goals emerged: Reliability, Security, Ease of Use, Re-usability, etc
- This resulted in thousands of actual programming languages

Language Evolution

Language Design Today

- We understand pretty well how to design good programming languages
- However, many bad languages are still designed
- After this class, you will be able to recognize bad programming languages

Lambda Calculus

- There are many programming languages we could talk about
- But pretty much all real languages are complex, large and obscure many important issues in irrelevant details
- We want: "as simple as possible" language to study properties of programming languages
- This language is known as lambda calculus

Lambda Calculus

- There are only four expressions in lambda calcus:
- Expression 1: constants
  - 1, 7, "yourName" are all valid expressions in lambda calculus
- Expression 2: identifiers
  - Will usually use x, y, etc for those
- Expression 3: lambda abstraction
  - written as $\lambda x.e$
- Expression 4: application
  - written as $e_1\ e_2$

Lambda Calculus Syntax

- Or, more concisely, the syntax of a lambda calculus expression as context-free grammar is given by:

$$e = c \mid \text{id} \mid \lambda \text{id}.e \mid e_1\ e_2$$

- This is a production that defines the left hand side (here an expression e)
- Observe that this production is recursive
- With this production, we can now check if any expression is valid lambda calculus

Lambda Calculus Syntax

- Consider the expression: $A = (\lambda x.x)\ 3$
- Now, recalling the syntax

$$e = c \mid \text{id} \mid \lambda\text{id}.e \mid e_1\ e_2$$

we can give a derivation proving that A is valid

- $e \to e_1\ e_2 \to e_1\ 3 \to (\lambda x.e)\ 3 \to (\lambda x.x)\ 3$
- Any expression for which we can find a derivation is syntactically valid lambda calculus

Are we done?

- We can now decide if any string is lambda calculus
- But we have no idea (yet) what these expressions mean!
- Just because we defined a syntax, this does not mean we have given meaning to expressions
- Giving meaning to syntax is called semantics
- Big chunk of this class: How to define syntax and semantics of programming languages

Lambda calculus semantics

- Let's define the meaning for each expression in our production:
  - Constant c: The meaning of c is the value of c
  - Identifier id: The meaning of id is id
  - Lambda $\lambda x.e$: The meaning: $\lambda x.e$
  - Application $\lambda x.e\ e_2$: The meaning: $e[e_2/x]$
- $e[e_2/x]$ is substitution. We replace all free occurrences of x by e_2 in expression e
- An occurrence of a variable is free if it is not bound by a λ
  Example: $(\lambda x.x)[2/x] = \lambda x.x$
- Upshot: We can define anonymous functions with binding operator λ.

Examples

- Meaning (or value) of $(\lambda x.x)\ 1$?
- $(\lambda x.x)\ 1 \to x[1/x] \to 1$
- $(\lambda x.(\lambda x.x)x)1 \to ((\lambda x.x)x)[1/x] \to (\lambda x.x)1 \to ...$
- Substitution is capture-avoiding: Does not replace variables bound by other λ's
- Convention: We assume that λ-bindings extend as far to the right as possible
- We read $\lambda x.\lambda y.xy$ as $(\lambda x.(\lambda y.xy))$ But use parenthesis to be safe

More Examples

- To make lambda calculus slightly more interesting, we will also allow arithmetic operators with their usual meaning
- We could give them precise semantics, but too boring. We all know their semantics
- $(\lambda x.5 * x)\ 1 \to (5 * x)[1/x] \to (5 * 1) \to 5$
- $(\lambda x.\lambda y.x + y)\ 3\ 5 \to ((\lambda y.x + y)[3/x])\ 5 \to (\lambda y.3 + y)\ 5 \to (3 + y)[5/y] \to (3 + 5) \to 8$

Properties of lambda expressions

- We have seen that to compute the value of lambda expressions, we only needed to define application: $\lambda x.e\ e_2$ as $e[e_2/x]$
- In lambda calculus, this is called β-reduction
- Confluence: Order of reductions is provably irrelevant
- Other property of lambda expressions: $\lambda x.e \Leftrightarrow \lambda y.(e[y/x])$
- This is called α-reduction
- Simply encodes that the name of lambda bound variables is irrelevant
- Analogy: $\int_0^\infty e^{-x}\,dx = \int_0^\infty e^{-y}\,dy$

Expression Equivalence

- Using $\alpha-$ and $\beta-$reductions, we can prove equivalence of expressions by computing their values using $\beta-$reduction and (if necessary) applying $\alpha-$reductions.
- Example: $e_1 = (\lambda x.x + 1)$ and $e_2 = (\lambda z.z + 1)$.
- Using $\alpha-$reduction, we can rewrite $e_1' = (\lambda x.x + 1) \rightarrow^{\alpha} (\lambda z.z + 1)$
- Have now proven that e_1 and e_2 are equivalent

What else?

- Lambda calculus looks very far from a real programming language.
- On the face of it, many features missing.
  - Multi-argument functions
  - Declarations
  - Conditionals
  - Named Functions
  - Recursion
  - ...
- Next: How to express these features in basic lambda calculus

Multi-argument functions

- How can we express adding two numbers?
- Recall earlier example: $(\lambda x.\lambda y.x + y)3\ 5$
- Here, we first reduce to $(\lambda x.\lambda y.x + y)\ 3\ 5 \rightarrow ((\lambda y.x + y)[3/x])\ 5 \rightarrow (\lambda y.3 + y)\ 5$
- In other words, we partially evaluate λx, resulting in a new function $(\lambda y.3 + y)$.
- This is equivalent to having a λ-binding with multiple arguments
- This is known as Currying

Declarations

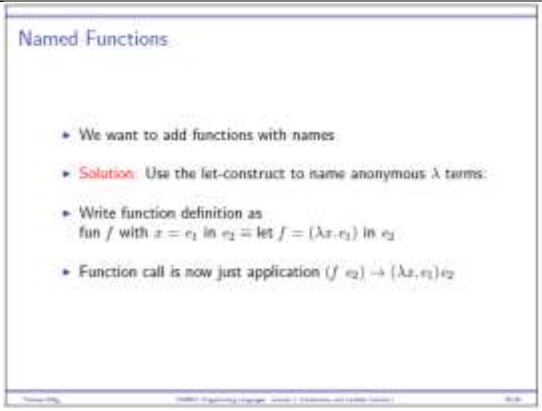
- We want to be able to give names to subexpressions
- Equivalence in typical programming languages: Local declarations
- Specifically, we want to add a let-construct of the following form to lambda calculus
- let $x = e_1$ in e_2
- Insight: Can define meaning of let-construct in in terms of basic lambda calculus

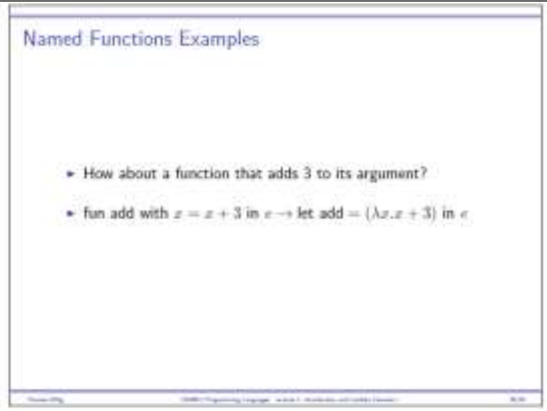
Declarations

- Any ideas?
- One possibility: let $x = e_1$ in e_2 means $e_2[e_1/x]$
- Or equivalently: let $x = e_1$ in e_2 means $(\lambda x.e_2)e_1$
- Why are these definitions equivalent?

Conditionals

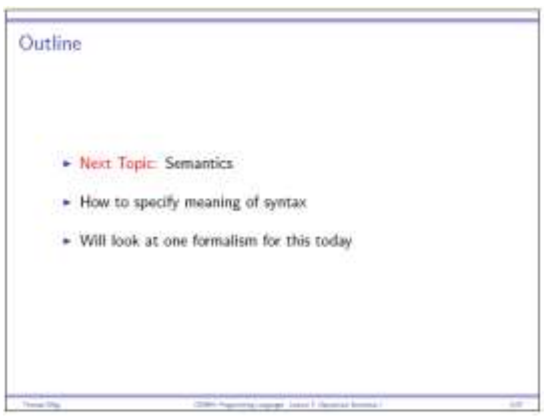
- Conditional: if x then e_1 else e_2
- Trick: We first define true and false as functions: let true $= (\lambda x \lambda y.x)$ let false $= (\lambda x \lambda y.y)$
- Recall: λ-bindings extend as far to the right as possible: $(\lambda x \lambda y.x) \equiv (\lambda x(\lambda y.x))$
- Then define conditional as: if p then e_1 else $e_2 \rightarrow (\lambda p \lambda e_1 \lambda e_2.p\ e_1\ e_2)$
- Here, p is a predicate, i.e. function evaluating to true or false
- Example predicates are EQZ, GTZ, etc.
- Observation: If we define numbers carefully in λ calculus, we can also define those precisely, but we won't in class

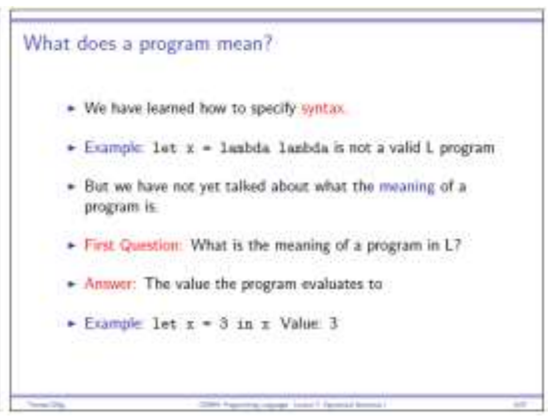


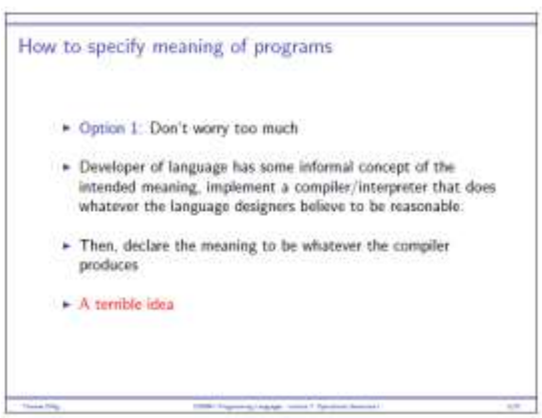


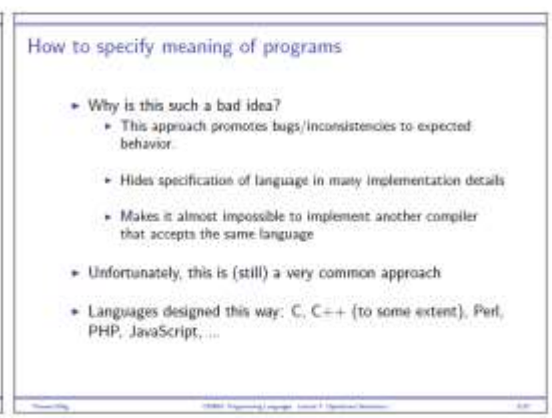
Semantics as the Meaning

Read these slides. While there are several forms of language semantics (axiomatic, denotational, and operational), we will focus on operational semantics in this course. Make sure that you understand the difference between eager versus lazy evaluation, and call-by-name vs call-by-value.









How to specify meaning of programs

- Option 2: Try to write out precisely the meaning of each language construct in documentation, then follow this description in implementation
- Example: Describe the meaning of !e in the L language:
- First attempt: "This evaluates to the head of e"
- What if e is not a list?
- Second attempt: "This evaluates to the head of e if e is a list, and to e otherwise"
- What if e is Nil? . . .

How to specify meaning of programs: Option 2

- Written language is, by nature, ambiguous. It is very difficult to fully specify the meaning of all language constructs this way
- Easy to miss cases
- Results in long, complicated and difficult to understand specifications, but an improvement over no specification

Written specification in practice

- Let's look at the ISO C++ standard: 879 pages, page 116:

Precisely Specifying Meaning

- Recall λ-calculus:
- To specify the meaning of expressions, we defined one single operation: β reduction
- Specifically, we wrote $\lambda x.e_1\ e_2 \rightarrow^{\beta} e_1[e_2/x]$
- Can read this as follows: If you see an expression of the form $\lambda x.e_1\ e_2$, you can compute its result as $e_1[e_2/x]$.

Operational Semantics

- Let's try the same in the language of arithmetic expression with the grammar:

$$S \rightarrow c \mid S_1 + S_2 \mid S_1 * S_2$$

- What is the meaning of an integer constant? The value of this integer
- More precisely: If we see an expression of the form c, its value is c
- We will write:

$$\vdash c : c$$

- Read as: "we can prove for any expression of the form c
- that the meaning of this expression is c"

Operational Semantics Cont.

- How about the expression $S_1 + S_2$?
- $\vdash S_1 + S_2 : ?$
- Problem: To describe the meaning of $S_1 + S_2$, we need to know the meaning (value) of S_1 and S_2
- Solution: Use hypotheses: We want to say "Assuming S_1 evaluates to c_1 and S_2 evaluates to c_2, the value of $S_1 + S_2$ is $c_1 + c_2$"
- We write this as:

$$\frac{\begin{array}{c}\vdash S_1 : c_1 \\ \vdash S_2 : c_2\end{array}}{\vdash S_1 + S_2 : c_1 + c_2}$$

Inference Rules

- This notation is known as inference rule

$$\frac{\begin{array}{c}\text{Hypothesis 1}\\ \dots \\ \text{Hypothesis N}\end{array}}{\vdash \text{Conclusion}}$$

- This means "given hypothesis 1, ... N, the conclusion is provable"
- Example:

$$\frac{\begin{array}{c}\text{Miterm 1 grade} >= 70\\ \dots \\ \text{Final grade} >= 140\end{array}}{\vdash \text{Final grade: A}}$$

Inference Rules cont.

- A hypothesis in an inference rule may use other rules
- Example:

$$\frac{\begin{array}{c}\vdash S_1 : c_1\\ \vdash S_2 : c_2\end{array}}{\vdash S_1 + S_2 : c_1 + c_2}$$

- You can tell this by a $\vdash$ in at least one of the hypotheses.
- Such rules are called inductive since they define the meaning of an expression in terms of the meaning of subexpressions.
- Rules that to not have $\vdash$ in any hypothesis are base cases
- A system with only inductive rules is nonsensical

Operational Semantics

- Back to the rule for +:

$$\frac{\begin{array}{c}\vdash S_1 : c_1\\ \vdash S_2 : c_2\end{array}}{\vdash S_1 + S_2 : c_1 + c_2}$$

- Let's focus on the first hypothesis $\vdash S_1 : c_1$.
- Question: Can you write $S_1 = c_1$?
- Answer: Yes, but now your first hypothesis is: " Assuming S_1 is the integer constant c_1" $\Rightarrow$ this rule no longer applies if, for example, $S_1 = 2 * 3$
- Read $\vdash$ as "is provable by using our set of inference rules".

Operational Semantics and Order

- Important Point: This notation does not specify an order between hypothesis.
- This means that

$$\frac{\begin{array}{c}\vdash S_1 : c_1\\ \vdash S_2 : c_2\end{array}}{\vdash S_1 + S_2 : c_1 + c_2}$$

and

$$\frac{\begin{array}{c}\vdash S_2 : c_2\\ \vdash S_1 : c_1\end{array}}{\vdash S_1 + S_2 : c_1 + c_2}$$

have exactly the same meaning

Full Operational Semantics

- Here are the full operational semantics of the language

$$S \to c \mid S_1 + S_2 \mid S_1 * S_2$$

$$\vdash c : c$$

$$\frac{\vdash S_1 : c_1 \quad \vdash S_2 : c_2}{\vdash S_1 + S_2 : c_1 + c_2}$$

$$\frac{\vdash S_1 : c_1 \quad \vdash S_2 : c_2}{\vdash S_1 * S_2 : c_1 * c_2}$$

Using Operational Semantics

- Consider the expression $(21 * 2) + 6$
- Here is how to derive the value of this expression with the operational semantics:

$$\frac{\dfrac{\vdash 21 : 21 \quad \vdash 2 : 2}{\vdash 21 * 2 : 42} \quad \vdash 6 : 6}{\vdash (21 * 2) + 6 : 48}$$

- This is a formal proof that the expression $(21 * 2) + 6$ evaluates to 48 under the defined operational semantics
- Observe that these proofs have a tree structure: Each subexpression forms a new branch in the tree

Operational Semantics of L

- Let's try to give operational semantics to the L language:
- Start with integers: $\dfrac{\text{Integer } \mathtt{i}}{\vdash \mathtt{i} : i}$
- The $\mathtt{i}$ in the hypothesis and to the left of the colon is the syntactic number in the source code of L
- The i after the colon is the value of the integer i.
- This sounds nitpicky, but is important to understand this notation.

Operational Semantics of L

- Consider the (integer) plus expression in L:

$$\frac{\begin{array}{c}\vdash e_1 : i_1 \text{ (integer)} \\ \vdash e_2 : i_2 \text{ (integer)}\end{array}}{\vdash e_1 + e_2 : i_1 + i_2}$$

- Side remark: The hypothesis can be written in separate lines (but not when giving a derivation tree)
- Here, the hypotheses require that e_1 and e_2 evaluate to integers.
- Question: What happens if e_1 evaluates to a list?
- Answer: No rule applies and computation is "stuck". This means the L program does not evaluate to anything.
- In practice: This is a run-time error

Operational Semantics of L

- Integer minus:

$$\frac{\begin{array}{c}\vdash e_1 : i_1 \text{ (integer)} \\ \vdash e_2 : i_2 \text{ (integer)}\end{array}}{\vdash e_1 - e_2 : i_1 - i_2}$$

- Integer times:

$$\frac{\begin{array}{c}\vdash e_1 : i_1 \text{ (integer)} \\ \vdash e_2 : i_2 \text{ (integer)}\end{array}}{\vdash e_1 * e_2 : i_1 * i_2}$$

Operational Semantics of L

- On to the key construct: λ
- Let's write semantics for the simple application (e1 e2)
- Recall that this is only defined if e1 is a lambda expression.
- Hypothesis: $\vdash e_1 : lambda\ x.\ e_1'$
- Now, how do we evaluate (e1 e2) ? $\vdash e_1'[e_2/x] : e$
- Conclusion: $\vdash (e_1\ e_2) : e$
- Final rule:

$$\frac{\begin{array}{c}\vdash e_1 : lambda\ x.\ e_1' \\ \vdash e_1'[e_2/x] : e\end{array}}{\vdash (e_1\ e_2) : e}$$

Order of Evaluation

- What would change if we write:

$$\frac{\begin{array}{c}\vdash e_1'[e_2/x] : e \\ \vdash e_1 : lambda\ x.\ e_1'\end{array}}{\vdash (e_1\ e_2) : e}$$

- Answer: Nothing. The written order of hypotheses is irrelevant
- Observe: This rule does specify an order between hypothesis: $\vdash e_1 : lambda\ x.\ e_1'$ must be evaluated before $\vdash e_1'[e_2/x] : e$.
- This is the case because $\vdash e_1'[e_2/x] : e$ uses e_1' defined by hypothesis $\vdash e_1 : lambda\ x.\ e_1'$
- Important Point: Operational semantics can encode order, but not through syntactic ordering

The Lambda Rule

- Question: What would change if we write the hypothesis as

$$\frac{\begin{array}{c}e_1 = lambda\ x.\ e_1' \\ \vdash e_1'[e_2/x] : e\end{array}}{\vdash (e_1\ e_2) : e}$$

- Answer: This would still give semantics to `(lambda x.x 3)`, but no longer to `let y=lambda x.x in (y 3)`

The Lambda Rule cont.

$$\frac{\vdash e_1 : lambda\ x.\ e_1' \quad \vdash e_1'[e_2/x] : e}{\vdash (e_1\ e_2) : e}$$

- Observe that in this rule, we are not evaluating e_2 before substitution.
- Consider the following modified rule:

$$\frac{\vdash e_1 : lambda\ x.\ e_1' \quad \vdash e_2 : e_2' \quad \vdash e_1'[e_2'/x] : e}{\vdash (e_1\ e_2) : e}$$

- This also is a well-formed rule, but it gives a different meaning to the lambda expression

The Lambda Rule cont.

- Consider both rules:

$$\frac{\vdash e_1 : lambda\ x.\ e_1' \quad \vdash e_1'[e_2/x] : e}{\vdash (e_1\ e_2) : e} \qquad \frac{\vdash e_1 : lambda\ x.\ e_1' \quad \vdash e_2 : e_2' \quad \vdash e_1'[e_2'/x] : e}{\vdash (e_1\ e_2) : e}$$

- Consider the expression `(lambda x.3 4+"duck")`:
  - Rule 1 evaluates this expression to "3"
  - Rule 2 "gets stuck" and returns no value since adding an integer and string is undefined (we have not given a rule)
- Two reasonable ways of defining application, but different semantics!

Call-by-name vs. Call-by-value

- Not evaluating the argument before substitution is known as call-by name, evaluating the argument before substitution as call-by-value.
- Languages with call-by-name: classic lambda calculus, ALGOL 60, L
- Languages with call-by-value: C, C++, Java, Python, FORTRAN, ...
- Advantage of call-by-name: If argument is not used, it will not be evaluated
- Disadvantage: If argument is uses k times, it will be evaluated k times!

Call-by-name vs. Call-by-value

- Consider the following expression in L syntax: `(lambda x.x+x+x (77*3-2))`
- Under call-by-name semantics, we substitute `(77*3-2)` for x and reduce the problem of evaluating `(lambda x.x+x+x (77*3-2))` to evaluating `((77*3-2)+(77*3-2)+(77*3-2))`
- We compute the value of x three times
- Under call-by-value semantics, we first evaluate `(77*3-2)` to 229 and then evaluate 229+229+229

Semantics of the let-binding

- Let's try to define the semantics of the let-binding in L: `let x = e1 in e2`
- One possibility:

$$\frac{\vdash e_1 : e_1' \quad \vdash e_2[e_1'/x] : e}{\vdash \text{let } x = e_1 \text{ in } e_2 : e}$$

- What about the following definition?

$$\frac{\vdash e_2[e_1/x] : e}{\vdash \text{let } x = e_1 \text{ in } e_2 : e}$$

- Are these definitions equivalent?

Eager vs. Lazy Evaluation

- Evaluating e_1 before we know that it is used is called eager evaluation
- Waiting until we need it is lazy evaluation.
- These are analogous to call-by-name/call-by value in trade offs.

Definition of let bindings

- But currently there is one problem common to both the eager and lazy definition of the let binding.
- Consider the following valid L program:

```
let f =
  lambda x.  if x <= 0 then 1 else x*(f (x-1))
in (f 2)
```

- What happens if we use our definition of let on this expression? For brevity, let's use the lazy one here, but the same problem exists with the eager one.

$$\frac{\vdash (\mathtt{f}\ 2)[\mathtt{lambda}\ x.\mathtt{if}\ x <= 0\ \mathtt{then}\ 1\ \mathtt{else}\ x * (\mathtt{f}(x-1))/\mathtt{f}] : ?}{\vdash \mathtt{let}\ \mathtt{f} = \mathtt{lambda}\ x.\mathtt{if}\ x <= 0\ \mathtt{then}\ 1\ \mathtt{else}\ x * (\mathtt{f}(x-1))\ \mathtt{in}\ (\mathtt{f}\ 2) : ?}$$

Let Binding

- We have already seen this problem when studying lambda calculus.
- But this time, we want to solve it. After all, who wants to use the Y-combinator for every recursive function!
- Solution: Add an environment to our rules that tracks mappings between identifiers and values
- Specifically, write the let rule as follows:

$$\frac{\begin{array}{c} E \vdash e_1 : e_1' \\ E[x \leftarrow e_1'] \vdash e_2 : e \end{array}}{E \vdash \text{let } x = e_1 \text{ in } e_2 : e}$$

Environments

- You can think of the environment as storing information to be used by other rules
- An environment maps keys to values
- Notation: $E[x \leftarrow y]$ means new environment with all mappings in E and the mapping $x \mapsto y$ added.
- If x was already mapped in E, the mapping is replaced
- Notation: $E(x) = y$ means bind value of key x in E to y. If no mapping $x \mapsto y$ exits in E, this "gets stuck"

Environments

- An environment adds extra information!
- In this rule:

$$\frac{\begin{array}{c} E \vdash e_1 : e_1' \\ E[x \leftarrow e_1'] \vdash e_2 : e \end{array}}{E \vdash \text{let } x = e_1 \text{ in } e_2 : e}$$

- Read the hypothesis $E \vdash e_1 : e_1'$ as: "Given environment E and expression e_1 and that it is provable that e_1 evaluates to e"
- Read the conclusion as: "Given environment E and expression let $x = e_1$ in e_2, this expression evaluates to e.

Environments

- Since we are no longer replacing the let-bound identifiers, we also need a base case for identifiers
- This will now use the environment:

$$\frac{\begin{array}{c} \text{Identifier id} \\ E(\text{id}) = e \end{array}}{E \vdash \text{id} : e}$$

- Adding the environment allows us now to be able to give (intuitive) meaning to recursive programs.

Environments Example

- Consider the L program `let x = 3 in x`
- Here is the proof that this program evaluates to 3:

$$\frac{E \vdash 3 : 3 \qquad \dfrac{\begin{array}{c} \text{Identifier x} \\ E[x \leftarrow 3](\text{x}) = 3 \end{array}}{E[x \leftarrow 3] \vdash \text{x} : 3}}{E \vdash \text{let } x = 3 \text{ in } x : 3}$$

Types

In this unit, you will learn about types, a method of enforcing levels of abstraction in programs. Data in programs come in many types: real number, integers, characters, lists, etc. A type error occurs when an operation is applied to an inappropriate data type. A type system consists of a set of types, and a set of programs to analyze types and type judgment. You will also learn about the basics of static typing, type checking and type inference.

Performance Tasks

Understanding Directed Assess

CMSC 308: PROGRAMMING LANGUAGES | 1
Hands-on Activity Rubric

Basic Adobe Photoshop Hands-on Activity

The activity is about proficiency with the functions and application in Programming Languages.

Score	COMPLIANCE *Project Guidelines and Compliance*	KNOWLEDGE *Ability to use required tools*	PRESENTATION *Demonstration of objective or competency in presentation*	SKILL *Uses a number of elements and principles*	
Excellent **90-100**	Project guidelines followed completely and all the required elements are present	Shows ability to use a variety of tools expertly and effectively -- at an excellent level.	The objective or competency was executed with thorough and precise judgments concluding in a carefully crafted product, presentation, or behavior.	Student used a skillful number of elements and principles to present their information.	
Proficient **80-89**	Project guidelines are mostly complete and all required elements are present.	Shows ability to use essential tools capably and effectively -- at a proficient level.	The objective or competency was executed with discernable and Correct judgments concluding in a proficiently crafted product, presentation, or behavior.	Student used an effective number of elements and principles to present their information.	
Adequate **70-79**	There is a missing important project requirement, or a guideline not followed	Shows ability to use basic tools competently and effectively -- at a satisfactory level.	The objective or competency was executed with apparent and acceptable judgments concluding in a satisfactorily crafted product, presentation, or behavior.	Student used a minimal number of elements and principles to present their information.	
Limited **60-69**	There are missing 2 or more important project requirements, or project guidelines not followed	Did not demonstrate ability to use basic tools competently and effectively at a satisfactory level.	The objective or competency was executed with superficial and vague Judgments concluding in an indistinctly crafted product, presentation, or behavior.	Student used an Inadequate number of elements and principles to present their information .	
Final Score					

This rubric came from ***http://barkerwchs.weebly.com/rubrics.html*** and was modified to suit the needs of the course.

Learning Resources

Website:

- *https://learn.saylor.org/course/view.php?id=79&sectionid=780*
- *https://learn.saylor.org/course/view.php?id=79&sectionid=781*
- *https://learn.saylor.org/course/view.php?id=79&sectionid=782*
- *https://learn.saylor.org/course/view.php?id=79&sectionid=783*
- *https://learn.saylor.org/course/view.php?id=79&sectionid=784*
- *https://learn.saylor.org/course/view.php?id=79&sectionid=785*
- *https://learn.saylor.org/course/view.php?id=79&sectionid=786*
- *https://www.computerscience.org/resources/computer-programming-languages/*

- **"Programming Language Pragmatics" by Michael L. Scott:**
  Author: Michael L. Scott/Publisher: Morgan Kaufmann/Edition: Fourth Edition/Publication Year: 2020/ISBN-13: 978-0124104099
- **"Eloquent JavaScript" by Marijn Haverbeke:**
  Author: Marijn Haverbeke/Publisher: No Starch Press/Edition: Third Edition/Publication Year: 2018/ISBN-13: 978-1593279509
- **"Python Crash Course" by Eric Matthes:**
  Author: Eric Matthes/Publisher: No Starch Press/Edition: Second Edition/Publication Year: 2019/ISBN-13: 978-1593279288
- Python.org
- Mozilla Developer Network (MDN)
- Codecademy