CSE216
Programming Abstraction

Instructor: Zhoulai Fu

State University of New York, Korea

- After-class assignments (homework) will be mostly coding exercises, to be given after we have learned more about coding

- In-class assignments (quizzes) will be given on a weekly basis, unless we are behind schedule for our course objectives

- Format of the quiz will remain paper-based

- Attendance checks will be done on a weekly basis

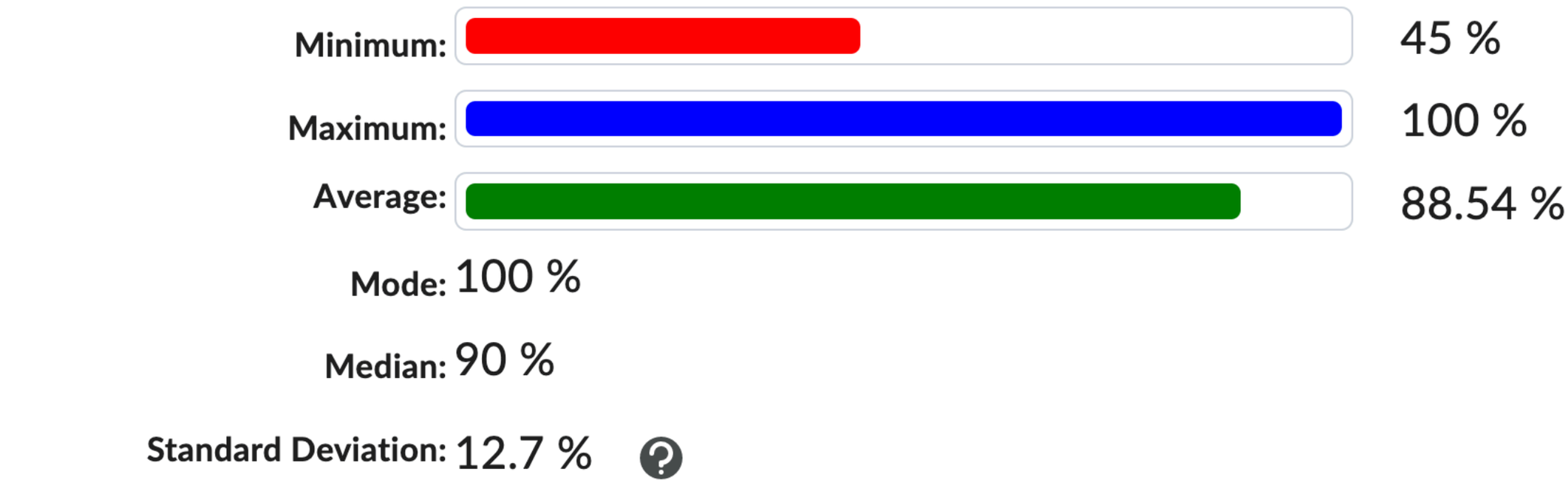
Today

- Quiz week 02 review
- Context-free Grammars, derivation and ambiguity
- Lambda calculus 1

Quiz week 02 review

Quiz Week 02 statistics

Number of submitted grades: 24 / 24



- The total score is 100. The last four exercises are counted as 5 points each, instead of 10. My apologies for the confusion.

Programming paradigms

Consider the following program in Python:

```
numbers = [1, 2, 3, 4, 5, 6]
sum = 0
for number in numbers:
    if number % 3 == 0:
        sum += number
print(sum)
```

1. (points = 10) What will be output if we run this program?

- Answer: **9**

2. (points = 10) What is the major paradigm used in the code? Choose from (a-d) below:
   (a) functional (b) object-oriented (c) imperative

- Answer: **c: imperative**

Lambda functions

In Python, lambda functions are defined using the lambda keyword followed by a comma-separated list of arguments (if any), followed by a colon and an expression. Here's an example:

```
add = lambda x, y: x + y
print(add(9, 3))
```

In this example, we define a lambda function `add` that takes two arguments `x` and `y`, and returns their sum. We then call the `add` function with arguments 9 and 3, which returns the sum 12.

3. (points = 10) Define an lambda function in python that takes a two input numbers and returns their distance. You can use the python function `abs` for the absolute value function. For example, `abs(-4.2)` returns 4.2. Write out the lambda expression. It should start with the key word "lambda".

- Answer: **lambda x, y: abs(x-y)**

A starter for onject-oriented programming

Consider the following code in Java

```
public class Customer {
    private String name;
    private SeniorityLevel level;

    public Customer(String name, SeniorityLevel level) {
        this.name = name;
        this.level = level;
    }

    public String getName() {
        return name;
    }

    public SeniorityLevel getLevel() {
        return level;
    }

    public void setLevel(SeniorityLevel level) {
        this.level = level;
    }
}

enum SeniorityLevel {
    NEW, REGULAR, VIP
}
```

Now, your co-worker Ji-Ho is working on web development to design a frontend for customers to operate on their accounts. Assume Ji-Ho's code has access to Custom objects.

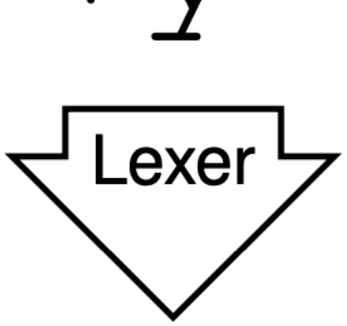
7. (points = 10) Is there any possibility that Ji-Ho's code can change a customer's name? **No**

8. (points = 10) Is there any possibility that Ji-Ho's code can change a customer's seniority level? **Yes**

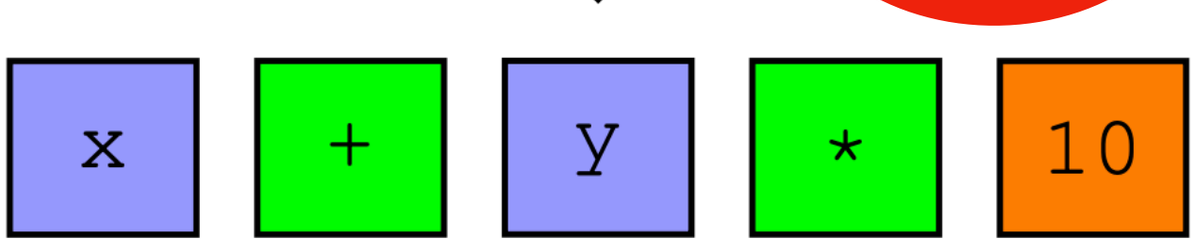
Context-free grammar

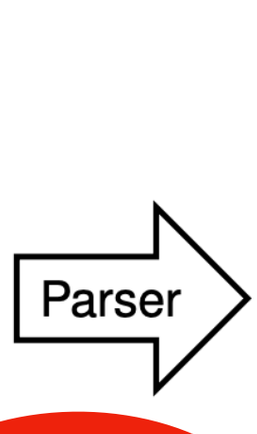
Parsing

"x +y*10"

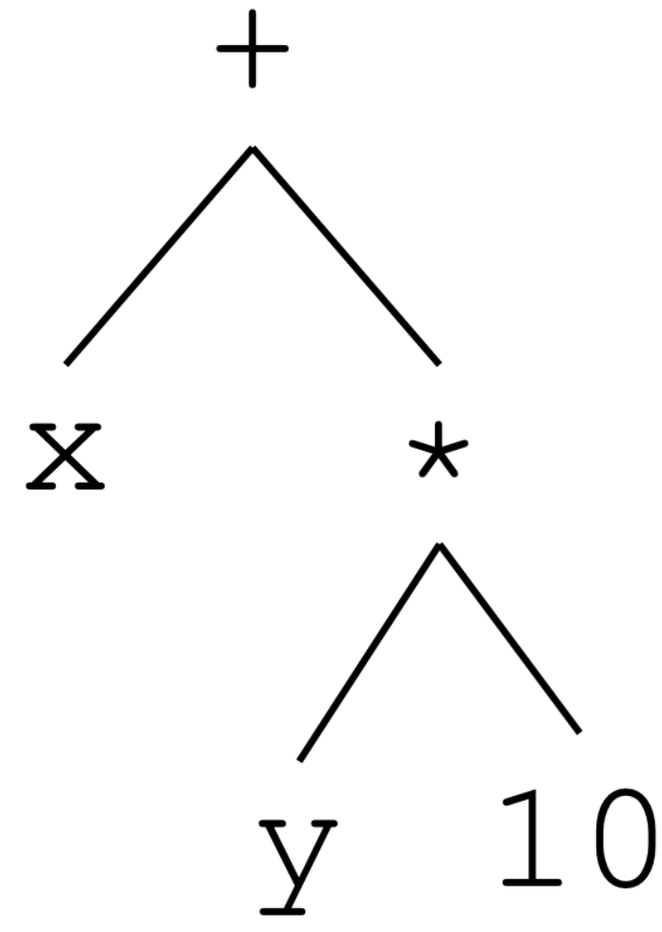


Regular expression Specification





Context-free grammar specification



Formal grammar

- Formal grammar is a set of rules **for generating the structure of a language** or a formal language. It is a mathematical framework used to describe the syntax of a language, where syntax refers to the set of rules governing how words and phrases are combined to form sentences.

- Formal grammars are typically expressed using a set of **symbols and production rules** that define how those symbols can be combined to form valid sentences or expressions.

- There are **different types of formal grammars, including context-free grammars, regular grammars, and context-sensitive grammars**, each with its own set of rules and restrictions. Formal grammar is an essential tool for analyzing and understanding the structure of languages, both natural and artificial, and it has applications in fields such as computational linguistics, natural language processing, and artificial intelligence.

Context-free grammar: Example 1

1. $S \rightarrow aSb$
2. $S \rightarrow ba$

- We start with S, and can choose a rule to apply to it.

- If we choose rule 1, we obtain the string aSb. If we then choose rule 1 again, we replace S with aSb and obtain the string aaSbb. If we now choose rule 2, we replace S with ba and obtain the string aababb, and are done.

- We can write this series of choices more briefly, using symbols: S => aSb => aaSbb => aababb

- The language of the grammar is the infinite set {a^n ba b^n | n>0} where n is is repeated times.

- This grammar is *context-free,* with only single nonterminals appear as left-hand sides)

Example 2: arithmetic expressions (ambiguous)

<expr> ::= <expr> + <expr>

| <expr> - <expr>

| <expr> * <expr>

| <expr> / <expr>

| (<expr>)

| <number>

<number> ::= 0 | 1 | 2 | ... | 9

This grammar is **ambiguous** because it allows for multiple parse trees to represent the same expression. For example, the expression 2 + 3 * 4 can be parsed as (2 + 3) * 4 or 2 + (3 * 4)

Ambiguous grammar

- An ambiguous grammar is a type of formal grammar that can produce **multiple parse trees** or interpretations for the same input sentence or sequence of symbols. In other words, the grammar can be interpreted in different ways, leading to more than one possible derivation or meaning for a given input.

- This can be problematic in various contexts because it can make it difficult to determine the correct meaning or parse tree of a sentence or sequence of symbols. For example, in programming languages, an ambiguous grammar can lead to errors or unexpected behaviors in code interpretation or compilation.

- To avoid ambiguity, it is often necessary to **use unambiguous grammars** or to add rules or constraints to the ambiguous grammar to disambiguate the interpretations.

Example 3: arithmetic expressions (unambiguous)

<expr> ::= <expr> + <term> | <expr> - <term>

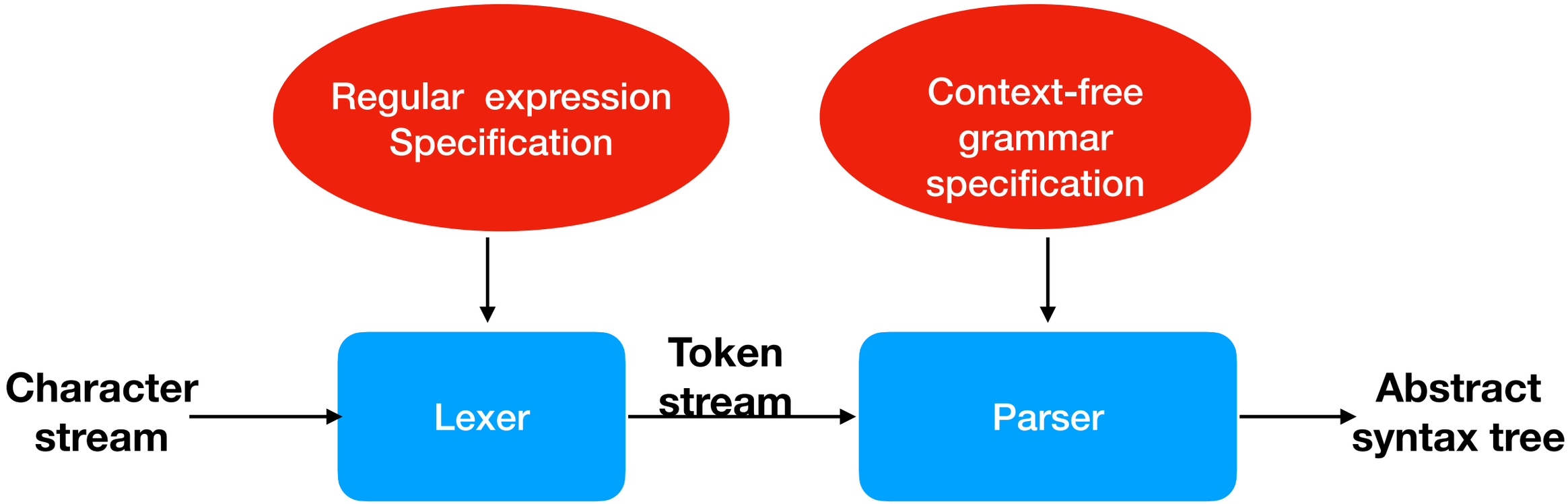
<term> ::= <term> * <factor> | <term> / <factor> | <factor>

<factor> ::= (<expr>) | <number>

<number> ::= 0 | 1 | 2 | ... | 9

In this revised grammar, the <expr> rule now includes a <term> component, which handles multiplication and division. The <term> rule includes a <factor> component, which handles parentheses and numbers. This approach ensures that the order of operations is clear and unambiguous

Summary so far



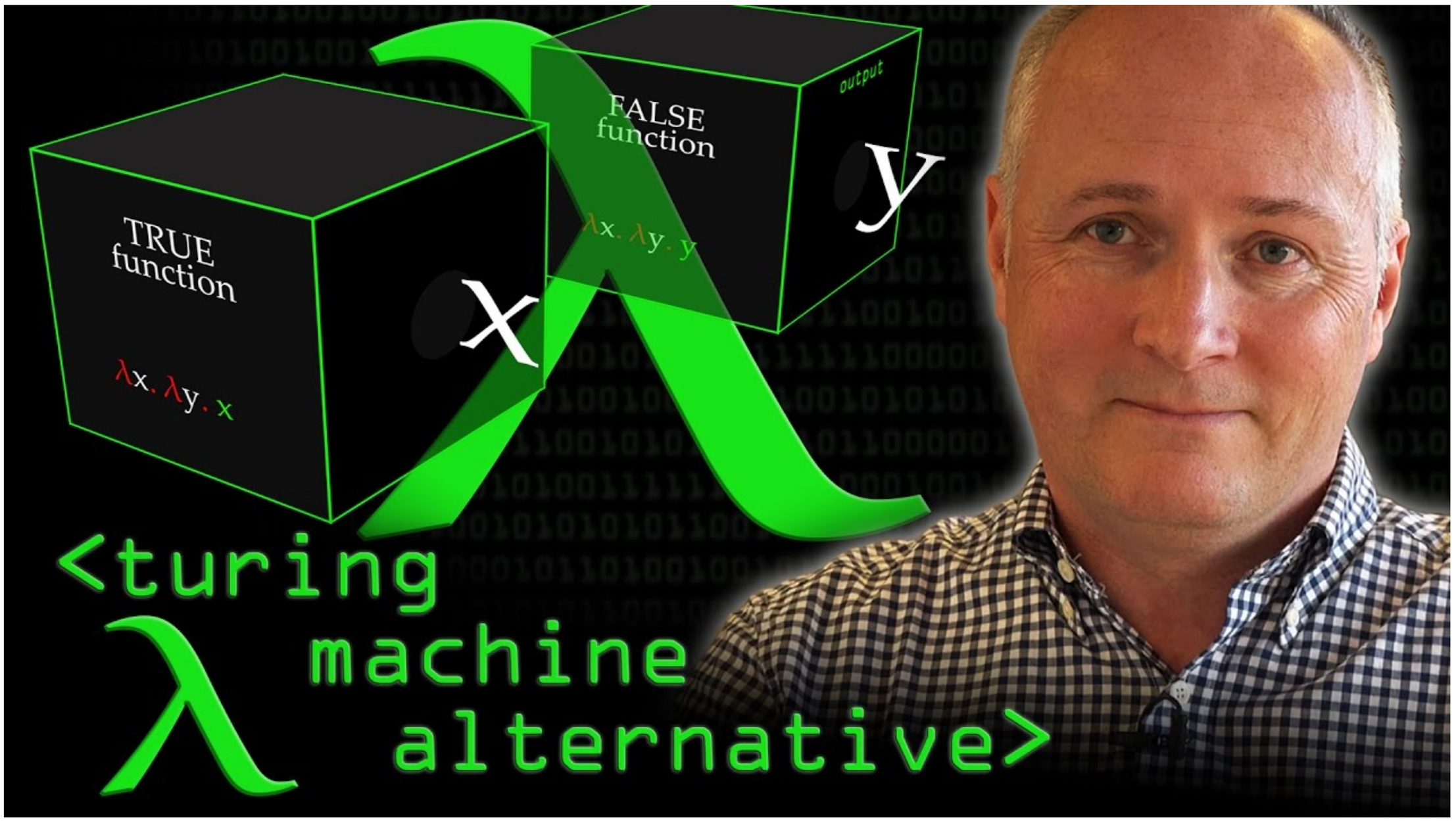
- A lexer converts a character stream to a token stream, using specification of regular expressions
- A parser converts a token stream to an abstract syntax tree, using specification of context-free grammar

Lambda calculus

Many slides adapted from CMU 15-252: More Great Theoretical Ideas in Computer Science
https://www.cs.cmu.edu/~venkatg/teaching/15252-sp20/notes/lambda-calculus-slides.pdf

Thanks!

A very nice Intro to lambda calculus



https://youtu.be/eis11j_iGMs

What is a computation/ algorithm?

Hilbert's 10th problem (1900):

Given a multivariate polynomial w/ integer coeffs,
e.g. $4x^2y^3 - 2x^4z^5 + x^8$,

"*devise a process according to which it can be determined in a finite number of operations*"
whether it has an integer root.

Mathematicians: "we should probably try to formalize what counts as an 'algorithm' ".

Gödel (1934):

Discusses some ideas for definitions of what functions/languages are "computable" but isn't confident what's a good definition.

Church (1936):

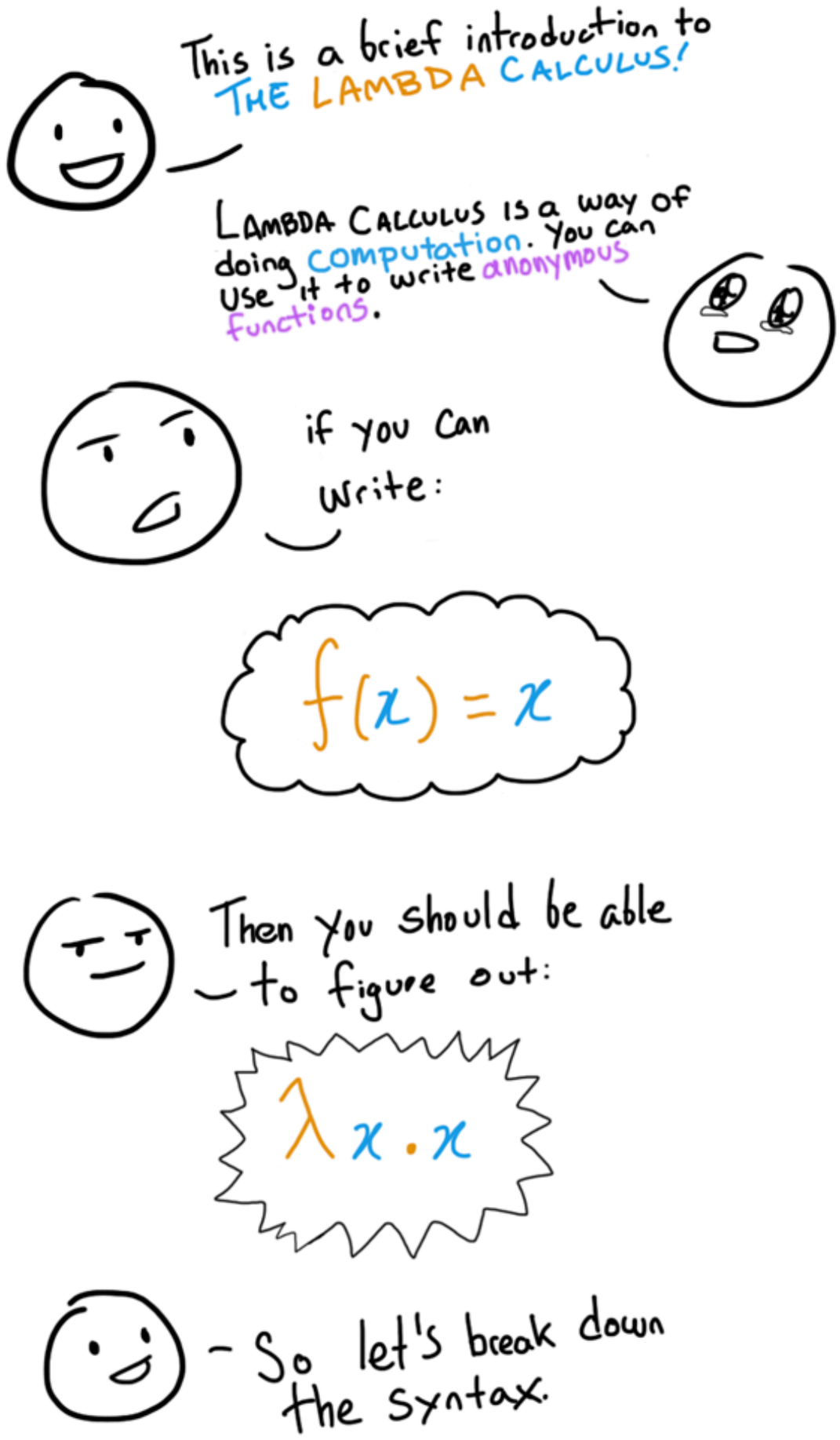
Invents lambda calculus,
claims it should be the definition.

Meanwhile… a certain British grad student in Princeton, unaware of all these debates…

Church-Turing thesis

Any natural / reasonable notion of computation realizable in the physical world can be simulated by a Turing machine (or equivalently, by lambda calculus).

In the following, we will discuss Alonzo Church's model of computing, **Lambda Calculus**.



Why study lambda calculus?

- Historically it preceded Turing machines.
- **Lambda calculus = Turing machine**
- Simplest known sufficiently powerful programming language.
- Modern functional programming languages (e.g. ML, Scheme, Haskell, F#, Scala) are direct descendants of lambda calculus.

- Syntax of Lambda Calculus
- Calculating with Lambda Calculus
- Uniqueness Theorems & Church-Rosser Property
- How to implement recursion in lambda calculus

Lambda calculus

- Syntax: expressions of form
  - EXPR::= x | lambda x. EXPR | EXPR EXPR
- Semantics: rules for **evaluating** the expression
  - Beta-reduction

“evaluating”?

For example: (+ 4 5)

+ is a function. We write functions in prefix form.

Another example: (+ (* 5 6) (* 8 3))

Evaluation proceeds by choosing a reducible expression and reducing it. (There may be more than one order.)

(+ (* 5 6) (* 8 3)) → (+ 30 (* 8 3)) → (+ 30 24) → 54

Function evaluation and currying

Function application is indicated by juxtaposition: f x

"The function f applied to the argument x"

What if we want a function of more than one argument?

We could invent a notation f(x,y), but there's an alternative. To express the sum of 3 and 4 we write:

((+ 3) 4)

The expression (+ 3) denotes the function that adds 3 to its argument.

Cont.

So all functions take one argument. So when we wrote (+ 3 4) this is shorthand for ((+ 3) 4). (The arguments are associated to the left.)

This mechanism is known as currying (in honor of Haskell Curry).

Parentheses can be added for clarity. E g:

((f ((+ 4) 3)) (g x)) is identical to f (+ 4 3) (g x)

Building functions and constants

- We will start out with a version of the lambda calculus that has constants like 0, 1, 2… and functions such as + * =.
- We also include constants TRUE and FALSE, and logical functions AND, OR, NOT.

Also conditional expressions, with these reduction rules:

IF TRUE E_1 E_2 → E_1

IF FALSE E_1 E_2 → E_2

Write your own functions - lambda abstraction

A way to write expressions that denote functions.

Example: (λx . + x 1)

λ means here comes a function

Then comes the parameter x

Then comes the .

Then comes the body + x 1

The function is ended by the) (or end of string)

(λx . + x 1) is the increment function.

(λx . + x 1) 5 ➞ 6

(We'll explain this more thoroughly later.)

Syntax summary

<exp> ::= <constant>	Built-in constants & functions
| <variable>	Variable names, e.g. x, y, z…
| <exp> <exp>	Application
| λ <variable> . <exp>	Lambda Abstractions
| (<exp>)	Parens

Example: λf.λn. IF (= n 0) 1 (* n (f (- n 1)))

Implied parentheses

Functions associate to the left: f g x stands for (f g) x

Lambdas associate to the right:
λx. x 3 stands for λx. (x 3),
i.e., function which takes x and applies it to 3

Note (λx. x) 3 equals 3 (identity function applied to 3)

Eg: λx.λy. x y (λz. z y) when fully paranthesized is

λx.λy. ((x y) (λz. (z y)))

Sometimes abbreviate λx.λy by λxy.

Bound and unbound variables

Consider this lambda expression:

(λx. + x y) 4

x is a *bound variable*, because it's inside the body of the λx expression.

y is an *unbound or free variable*, because it's not in a λy expression.

In this expression + x ((λx. + x 1) 4) The first x is free, and the second one is bound. In (+ x 1) , the x is free.

β-Reduction (aka λ expression evaluation)

Consider this lambda expression:

$$(\lambda x.\ +\ x\ 1)\ 4$$

This juxtaposition of $(\lambda x.\ +\ x\ 1)$ with 4 means to apply the lambda abstraction $(\lambda x.\ +\ x\ 1)$ to the argument 4. Here's how we do it:

The result of applying a lambda abstraction to an argument is an instance of the body of the lambda abstraction in which bound occurrences of the formal parameter in the body are replaced with copies of the argument.

I.e. we put the 4 in for the x in $+\ x\ 1$. The result is $+\ 4\ 1$.

$$(\lambda x.\ +\ x\ 1)\ 4 \rightarrow_{\beta} +\ 4\ 1$$

β-Reduction (some more examples)

$(\lambda x.\ +\ x\ x)\ 5 \rightarrow\ +\ 5\ 5 \rightarrow\ 10$

$(\lambda x.\ 3)\ 5 \rightarrow\ \ \ \ 3$

$(\lambda x.\ (\lambda y.\ -\ y\ x))\ 4\ 5 \rightarrow\ (\lambda y.\ -\ y\ 4)\ 5 \rightarrow\ -\ 5\ 4 \rightarrow 1$

(Note the currying – we peel off the arguments 4 then 5.)

$(\lambda f.\ f\ 3)\ (\lambda x.\ +\ x\ 1) \rightarrow\ (\lambda x.\ +\ x\ 1)\ 3 \rightarrow\ +\ 3\ 1 \rightarrow 4$

$(\lambda x.\ (\lambda x.\ +\ (-\ x\ 1))\ \ x\ 3)\ 9 \rightarrow (\lambda x.\ +\ (-\ x\ 1))\ 9\ 3$

$\rightarrow\ +\ (\ -\ 9\ 1)\ 3$

$\rightarrow\ 11$

β-Reduction (contd)

(λx. (λx. + (- x 1)) x 3) 9 → (λx. + (- x 1)) 9 3
→ + (- 9 1) 3
→ 11

In this last example we could have applied the reduction in a different order.

(λx. (λx. + (- x 1)) x 3) 9 → (λx. + (- x 1) 3) 9
→ + (- 9 1) 3
→ 11

β-Reduction (ordering)

So, to evaluate an expression we search for reductions, and make them. When the process stops (there are no more reductions to apply), the expression is said to be in ***normal form***. This is the *value* of the original expression.

Wait… there are different orders in which to do the reductions. Does the order matter? If so, we have a problem.

β-Reduction (ordering)

Let D be this expression: (λx. x x). What is the value of this expression (D D)?

(D D) = (λx. x x) (λx. x x) → (λx. x x) (λx. x x) → …

It never terminates!

Now consider this expression: (λx. 3) (D D)

If we apply β-reduction to the leftmost part first, we get 3. If we keep applying it to (D D) we go into an infinite loop.

Former is *call-by-name* (lazy evaluation)
Latter is *call-by-value* (eager evaluation)

On the ordering of reductions

Church-Rosser Theorem : No expression can be converted into two distinct normal forms.

(Modulo renaming .. I.e (λx. + x x) and (λy. + y y) the same.)

Note: evaluations for some ordering may not terminate,
but will never terminate leading to a different normal form.

Implementing recursion in λ calculus

We claimed that lambda calculus was powerful.

We've seen how to define expressions.

But the language does not seem to support loops or recursive calls.

All functions are anonymous. There is no mechanism for naming a function, then calling it by its name.

But one can get around this problem.

Implementing recursion

Suppose we wanted to write a factorial function which takes a number n and computes n! Here's an attempt to get started:

λn.if (= n 0) 1 (* n (f (- n 1)))

This does not work. Because what is the unbound variable f ? It would work if we could somehow make f be the very function above.

Implementing recursion

Idea: To give us access to that function, how about passing it in as another parameter?

(λf.λn.if (= n 0) 1 (* n (f (- n 1)))))

- But we need to pass this function to itself (recursively).
- So inner recursive call should be (f f (- n 1)).

Implementing recursion

To give us access to a function itself, we pass it in as another parameter.

FACT = (λf.λn.if (= n 0) 1 (* n (f f (- n 1)))))

(FACT is just shorthand for that string of characters.)

Now if we write

FACT FACT 5

This will work, because the β-reduction substitutes FACT for f, resulting in a function call FACT FACT 4. Etc.

Implementing recursion

Another example. The Fibonacci numbers

FIB = λf.λn.if (= n 0) 0 (if (= n 1) 1 (+ (f f (- n 1)) (f f (- n 2)))))

Now FIB FIB n computes the nth Fibonacci number.

A general method for recursion: Y-combinator

There's a systematic way to do this. We'll just touch upon this, but you can see http://en.wikipedia.org/wiki/Fixed-point_combinator

Let H be the factorial function we attempted to use before:

$$H = \lambda f.\lambda n.\text{if } (= n\ 0)\ 1\ (* n\ (f\ (- n\ 1)))$$

Let Y be the following expression.

$$Y = \lambda h.(\ \lambda x.\ (h\ (x\ x))\ \ \lambda x.\ (h\ (x\ x))\ )$$

Turns out Y H is the factorial function that we wanted.

Exercise: ***Check this!***