CSC324: Principles of Programming Languages

Lecture 1

Friday 8 May, 2020

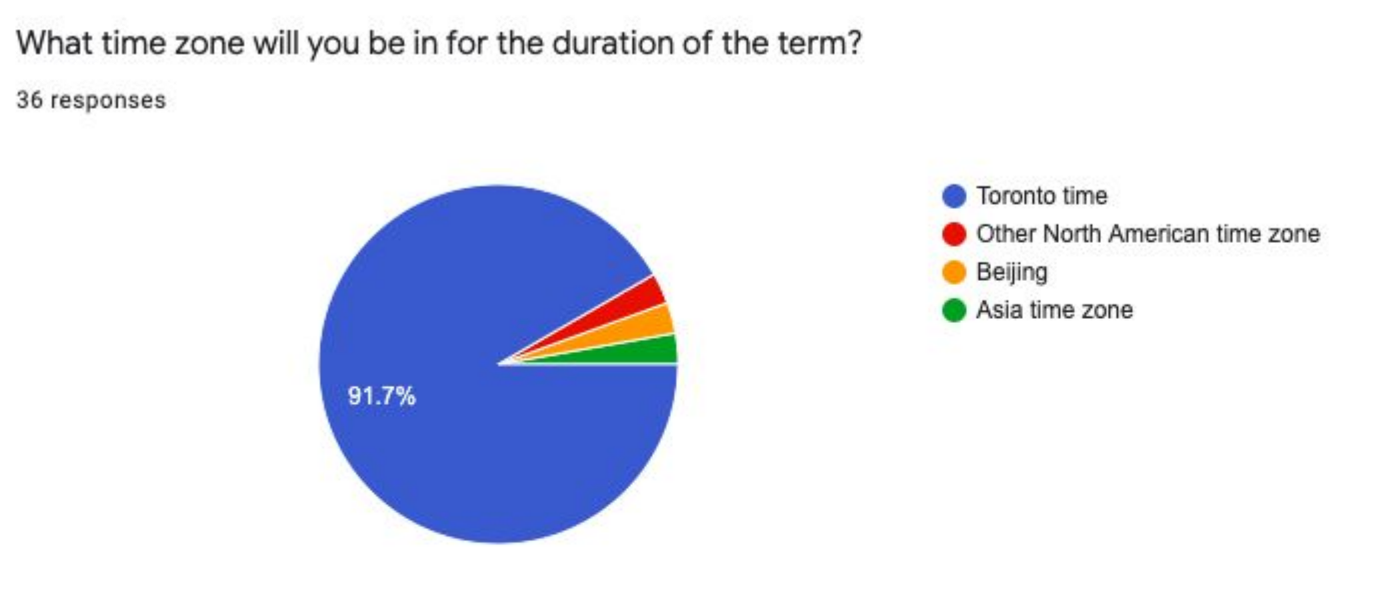
Course announcements

- Hope you are enjoying learning the basics of Racket and Haskell! (If you haven't started yet, no time like the present to start… check out the software page and Exercise 1/Lab 1 to get started.)

- Sorry that many of you had audio issues on Wednesday. I have raised those issues with instructional support.

- Moving to Piazza this weekend. Will make a Quercus announcement when it's ready to go.

Questionnaire results



Questionnaire results

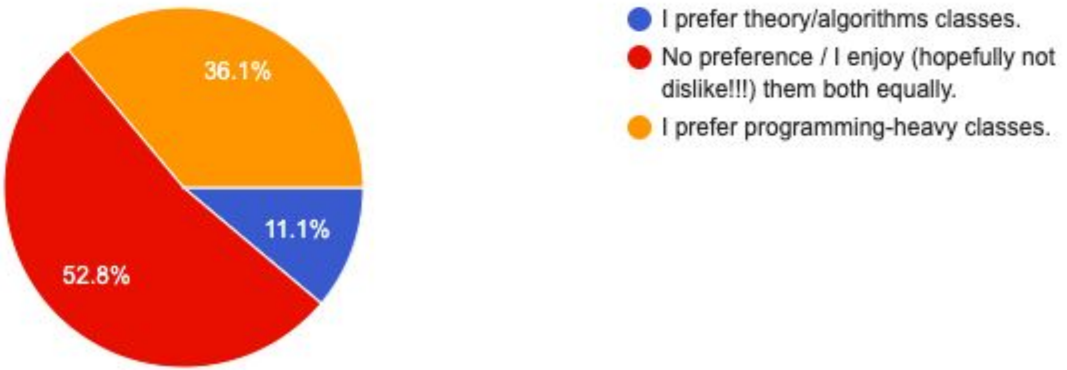
Do you have any past experience with functional programming or the programming languages we'll be covering in this class? (note: past experience is ABSOLUTELY NOT expected, but this will help me guide the pace of the course.)

36 responses

Questionnaire results

In your past, have your interests leaned towards theory/algorithms classes, or have you enjoyed programming-heavy classes more? (This will help me guide possible extra topics in the course.)

36 responses



Questionnaire results

- "Having the recorded lectures available for download would be extremely helpful if possible."
    - *downloadable/streamable through BB Collaborate*
- can you hold a Ice breaker time in lab so I can find my assignment partner.
    - Monday during lab session!
- Can the pace of the class be not too fast? That will be appreciated :))
    - 1) Let me know if I'm speaking too fast!
    - 2) I promise not to go faster than we need to :)

Today: **Syntax, Semantics, & Evaluation**

- Syntax and Semantics (30-35 min)

- Evaluation in the lambda calculus (10-15 min)

Syntax

<u>Definition</u>: The **syntax** of a computer language is the set of rules that defines the combinations of symbols that form a correctly-structured program" (Wikipedia)

Let's discuss the syntax of a language of **simple arithmetic.**

Examples (in the Python interpreter…)

Syntax of an arithmetic language

We saw some examples of valid "programs" in this language:

```
42

(6 * 8)

((1 + 2) / (3 * 4))
```

How do we define, formally, whether an input is a valid expression (and hence a valid "program") in this language?

Syntax: Grammars

Definition: a **grammar** is the set of rules that specify valid syntax

```
42

(6 * 8)

((1 + 2) / (3 * 4))
```

Definition: An **expression** is a syntactically-valid sequence of tokens

Syntax: Grammars

Definition: a **grammar** is the set of rules that specify valid syntax

```
42

(6 * 8)

((1 + 2) / (3 * 4))
```

In our simple arithmetic program, a program is valid if it is a valid arithmetic expression.

What rules can we say about each of the above "programs"?

Syntax: Grammars

LHS: "A prog is…"

```
<prog> = <expr>
```

RHS: "...an expr."

```
42

(6 * 8)

((1 + 2) / (3 * 4))
```

```
<expr> = ???
```

LHS: "An expr is…"

Syntax: Grammars

LHS: a parser **rule**

```
<prog> = <expr>
```

RHS: a production is one or more symbols

```
42

(6 * 8)

((1 + 2) / (3 * 4))
```

```
<expr> = ???
```

Syntax: Grammars

```
42

(6 * 8)

((1 + 2) / (3 * 4))
```

```
<prog> = <expr>

<expr> = … | "-1" | "0" | "1" | "2" | …
```

Syntax: Grammars

```
42

(6 * 8)

((1 + 2) / (3 * 4))
```

```
<prog> = <expr>

<expr> = NUMBER
```

Let's assume that parsing a number is axiomatic

Syntax: Grammars

a non-terminal symbol

```
<prog> = <expr>
```

a terminal symbol

```
<expr> = NUMBER
```

```
42

(6 * 8)

((1 + 2) / (3 * 4))
```

Syntax: Grammars

```
42

(6 * 8)

((1 + 2) / (3 * 4))
```

```
<prog> = <expr>

<expr> = NUMBER
```

Given the input "31337", the parser would:

- Determine that "31337" parses as a number
- Determine that a number parses as an expr
- Determine that an expr is a prog
- Concludes via transitivity "31337" is a syntactically-valid program

The substitution rule

Any time the right-hand side of a rule has been successfully parsed, it can be **substituted** by the left-hand side of the rule.

```
<prog> = <expr>


<expr> = NUMBER
```

Given the input "31337", the parser would:

- Determine that "31337" parses as a number
- Determine that a number parses as an expr
- Determine that an expr is a prog
- Concludes via transitivity "31337" is a syntactically-valid program

Syntax: Grammars

```
<prog> = <expr>
```

```
42

(6 * 8)

((1 + 2) / (3 * 4))
```

```
<expr> = NUMBER
```

Syntax: Grammars

```
42

(6 * 8)

((1 + 2) / (3 * 4))
```

```
<prog> = <expr>

<op> = "+" | "-" | "*" | "/"

<expr> = NUMBER
```

Syntax: Grammars

```
42

(6 * 8)

((1 + 2) / (3 * 4))
```

```
<prog> = <expr>

<op> = "+" | "-" | "*" | "/"

<expr> = NUMBER

       | "(" NUMBER <op> NUMBER ")"
```

...a first attempt...

```
( 6 * 8 )
```

```
<op> = "+" | "-" | "*" | "/"

<expr> = NUMBER

       | "(" NUMBER <op> NUMBER ")"
```

(6 * 8)

```
<op> = "+" | "-" | "*" | "/"

<expr> = NUMBER

       | "(" NUMBER <op> NUMBER ")"
```

(6 * 8)

```
<op> = "+" | "-" | "*" | "/"

<expr> = NUMBER

       | "(" NUMBER <op> NUMBER ")"
```

(6 * 8)

```
<op> = "+" | "-" | "*" | "/"

<expr> = NUMBER

       | "(" NUMBER <op> NUMBER ")"
```

(6 * 8)

```
<op> = "+" | "-" | "*" | "/"

<expr> = NUMBER

       | "(" NUMBER <op> NUMBER ")"
```

(6 * 8)

```
<op> = "+" | "-" | "*" | "/"

<expr> = NUMBER

       | "(" NUMBER <op> NUMBER ")"
```

```
( 6 * 8 )
```

```
<op> = "+" | "-" | "*" | "/"

<expr> = NUMBER

       | "(" NUMBER <op> NUMBER ")"
```

(6 * 8)

```
<op> = "+" | "-" | "*" | "/"

<expr> = NUMBER

       | "(" NUMBER <op> NUMBER ")"
```

```
(  *  6  8  )
```

```
<op> = "+" | "-" | "*" | "/"

<expr> = NUMBER

       | "(" NUMBER <op> NUMBER ")"
```

```
( * 6 8 )
```

```
<op> = "+" | "-" | "*" | "/"

<expr> = NUMBER

       | "(" NUMBER <op> NUMBER ")"
```

(* 6 8)

```
<op> = "+" | "-" | "*" | "/"

<expr> = NUMBER

       | "(" NUMBER <op> NUMBER ")"
```

(* 6 8)

Error parsing expression: expected NUMBER, got "*" instead

```
<exp
      | "(" NUMBER
```

Syntax: Grammars

```
42

(6 * 8)

((1 + 2) / (3 * 4))
```

```
<prog> = <expr>

<op> = "+" | "-" | "*" | "/"

<expr> = NUMBER

       | "(" NUMBER <op> NUMBER ")"
```

Syntax: Grammars

```
42

(6 * 8)

((1 + 2) / (3 * 4))
```

```
<prog> = <expr>

<op> = "+" | "-" | "*" | "/"

<expr> = NUMBER

       | "(" NUMBER <op> NUMBER ")"

       | "(" "(" NUMBER <op> NUMBER ")" <op>
```

Syntax: Grammars

```
42

(6 * 8)

((1 + 2) / (3 * 4))
```

```
<prog> = <expr>

<op> = "+" | "-" | "*" | "/"

<expr> = NUMBER

       | "(" NUMBER <op> NUMBER ")"

       | "(" "(" NUMBER <op> NUMBER ")" <op>

       | "(" "(" "(" NUMBER <op> NUMBER ")"

       | "(" "(" "(" "(" NUMBER <op> NUMBER
```

Life Lesson: Look For Recursive Structure!

Note that

$$\underbrace{\texttt{((1 + 2) / (3 * 4))}}_{\text{<expr>}}$$

Life Lesson: Look For Recursive Structure!

Note that

$$(\underbrace{(1 + 2)}_{\texttt{<expr>}} \; / \; \underbrace{(3 * 4)}_{\texttt{<expr>}})$$

Life Lesson: Look For Recursive Structure!

Note that

$$(\ (\underbrace{1}_{\text{NUMBER}} + \underbrace{2}_{\text{NUMBER}}) \ / \ (\underbrace{3}_{\text{NUMBER}} * \underbrace{4}_{\text{NUMBER}}) \ )$$

Life Lesson: Look For Recursive Structure!

Note that

$$(\ (\underbrace{1}_{\text{<expr>}} + \underbrace{2}_{\text{<expr>}})\ /\ (\underbrace{3}_{\text{<expr>}} * \underbrace{4}_{\text{<expr>}})\ )$$

Syntax: Grammars

```
42

(6 * 8)

((1 + 2) / (3 * 4))
```

```
<prog> = <expr>

<op> = "+" | "-" | "*" | "/"

<expr> = NUMBER

       | "(" NUMBER <op> NUMBER ")"
```

Syntax: Grammars

```
<prog> = <expr>

<op> = "+" | "-" | "*" | "/"

<expr> = NUMBER
       | "(" <expr> <op> <expr> ")"
```

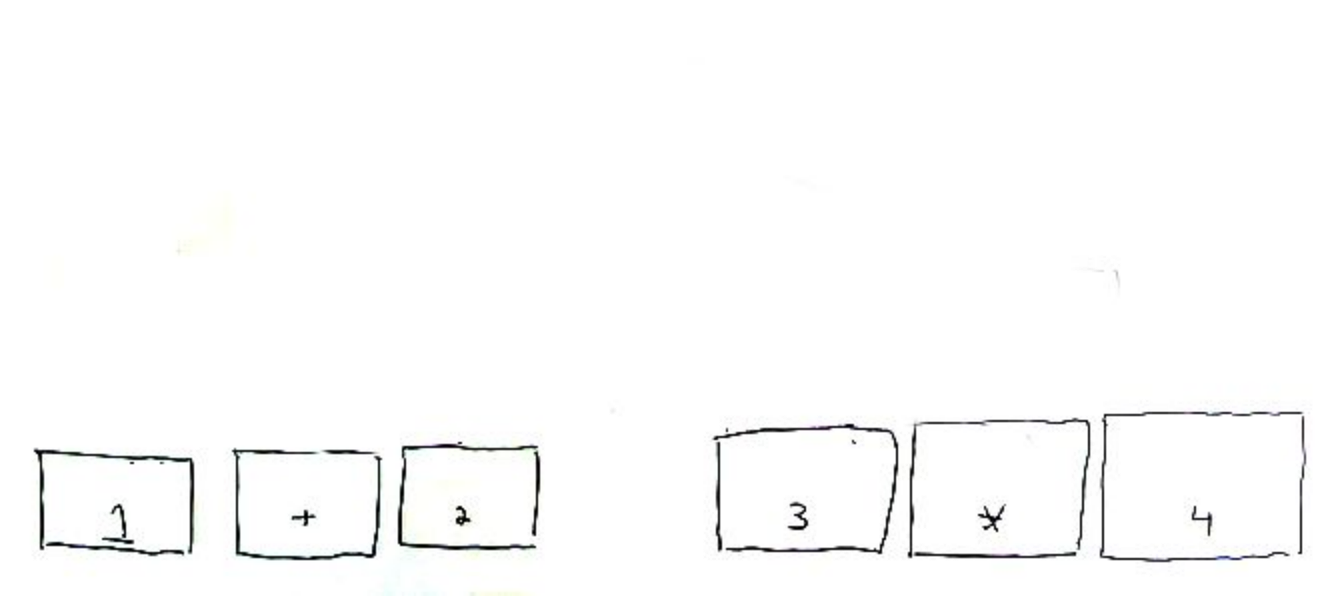
```
42

(6 * 8)

((1 + 2) / (3 * 4))
```

Definition: **Parsing** is the process by which a string representation of an expression is transformed into a “more structured” representation

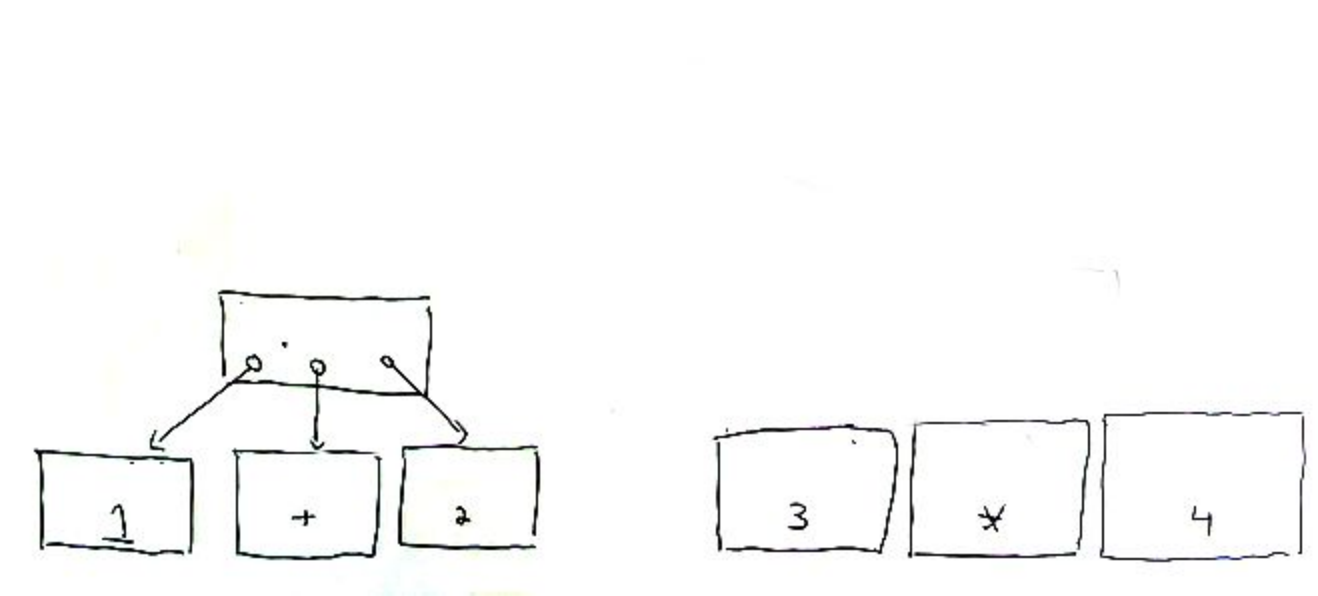
How could we design a data structure that could be the return value of the function

```
SomeReturnValueTBD ParseProgram(String input) { … } ?
```

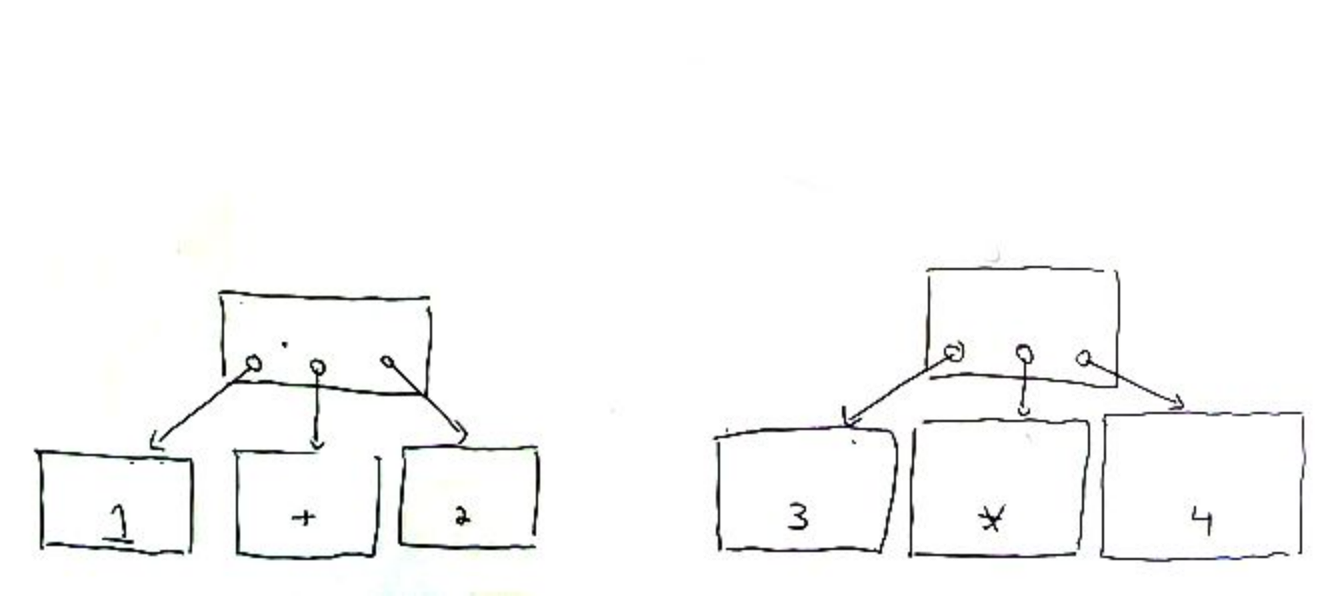
$$((1 + 2) / (3 * 4))$$



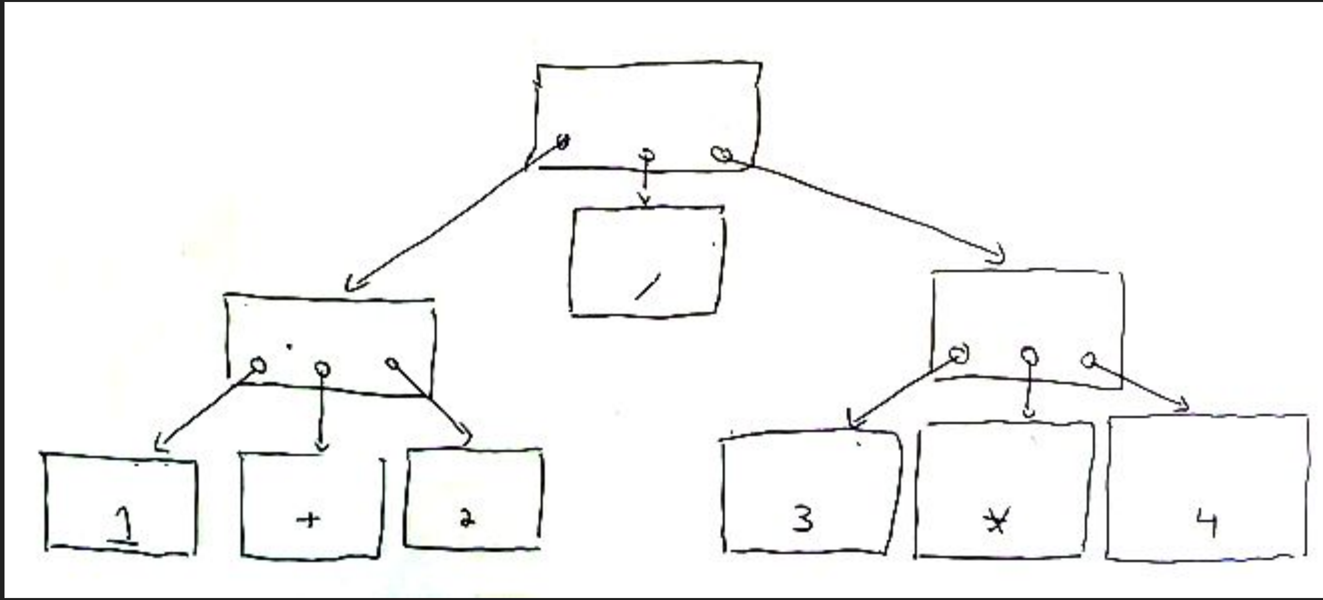
((1 + 2) / (3 * 4))



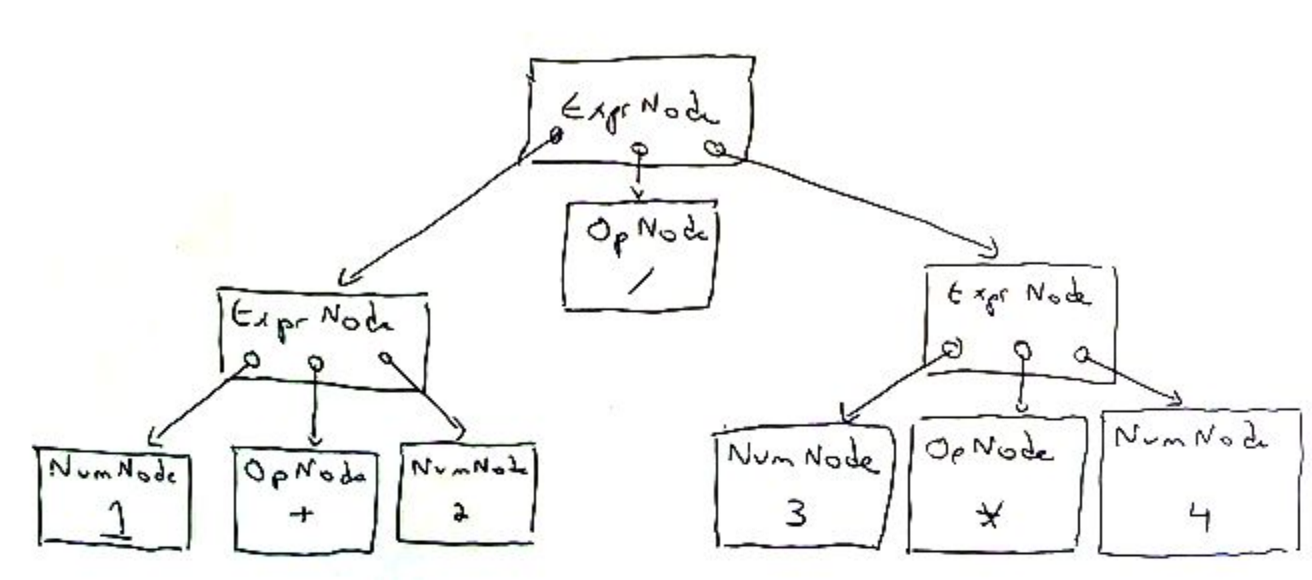
((1 + 2) / (3 * 4))



((1 + 2) / (3 * 4))



((1 + 2) / (3 * 4))



ExprNode
OpNode
/
ExprNode
ExprNode
NumNode
1
OpNode
+
NumNode
2
NumNode
3
OpNode
*
NumNode
4

Abstract Syntax Tree

Definition: An **abstract syntax tree (AST)** is a data-structure representation of a computer program. Each node in the tree corresponds to a production in the grammar.

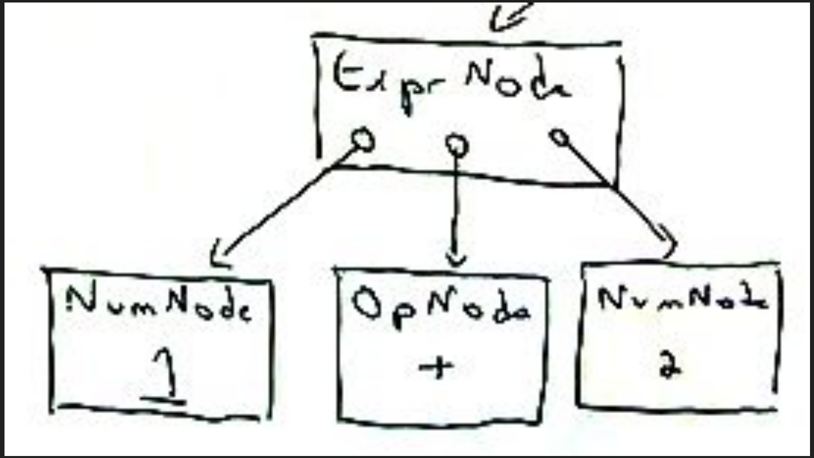
```
SyntaxTree ParseProgram(String input) { … }
```

Looking ahead to part 2: metaprogramming is the process of manipulating ASTs.

Abstract Syntax Tree

Definition: An **abstract syntax tree (AST)** is a data-structure representation of a computer program. Each node in the tree corresponds to a production in the grammar.

Recall that a tree datatype has **leaf nodes** and **interior nodes**: what can we say about the relationship between symbols in our grammar, and the kinds of nodes in the AST?

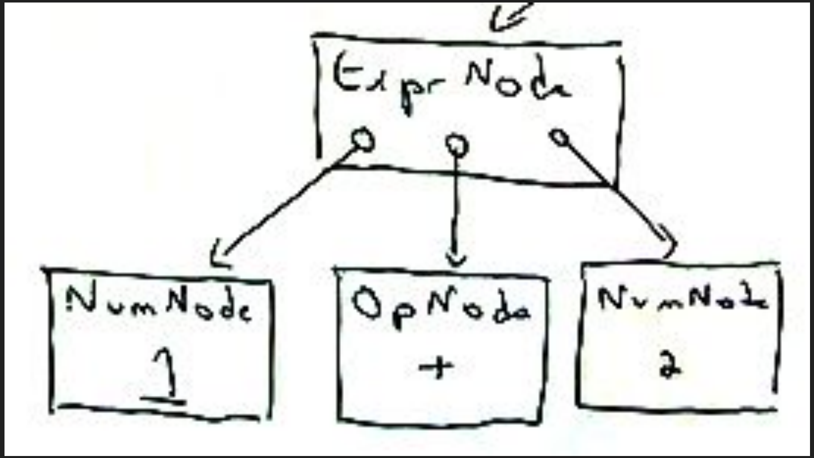


Abstract Syntax Tree

```
<prog> = <expr>

<op> = "+" | "-" | "*" | "/"

<expr> = NUMBER

       | "(" <expr> <op> <expr> ")"
```

Leafs in the AST correspond to **terminal symbols** in the grammar

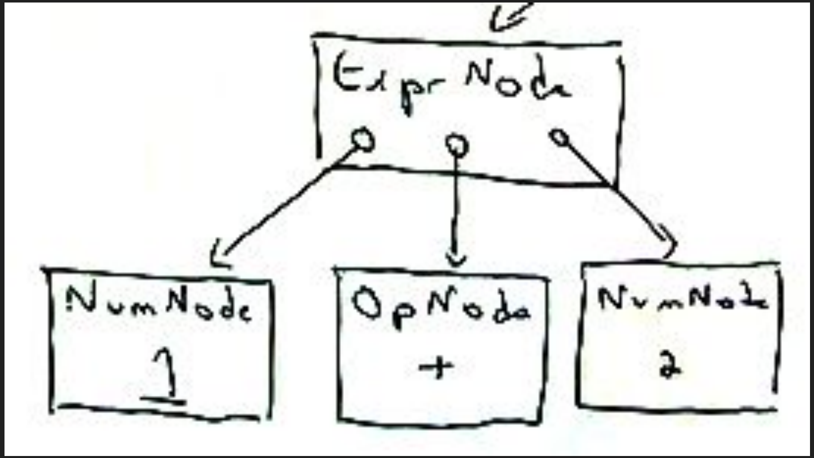
Interior nodes in the AST correspond to **non-terminal symbols in** the grammar



Abstract Syntax Tree

```
<prog> = <expr>

<op> = "+" | "-" | "*" | "/"

<expr> = NUMBER

       | "(" <expr> <op> <expr> ")"
```

```
interface ASTNode {...}

class OpNode implements ASTNode {
     private char opSymbol;
}

abstract class ExprNode implements ASTNode {...}

class NumNode extends ExprNode {
     private int value;
}

class BinaryOpNode extends ExprNode {
     private ExprNode lhs, rhs;
     private OpNode op;
}
```



Semantics

Parsing into an AST gives us the **syntactic form** of a program, but doesn't tell us anything about the *meaning* of the program.

Semantics

Parsing into an AST gives us the **syntactic form** of a program, but doesn't tell us anything about the *meaning* of the program.

Definition: The **semantics** of a programming language are the rules governing the *meaning* of programs written in that language

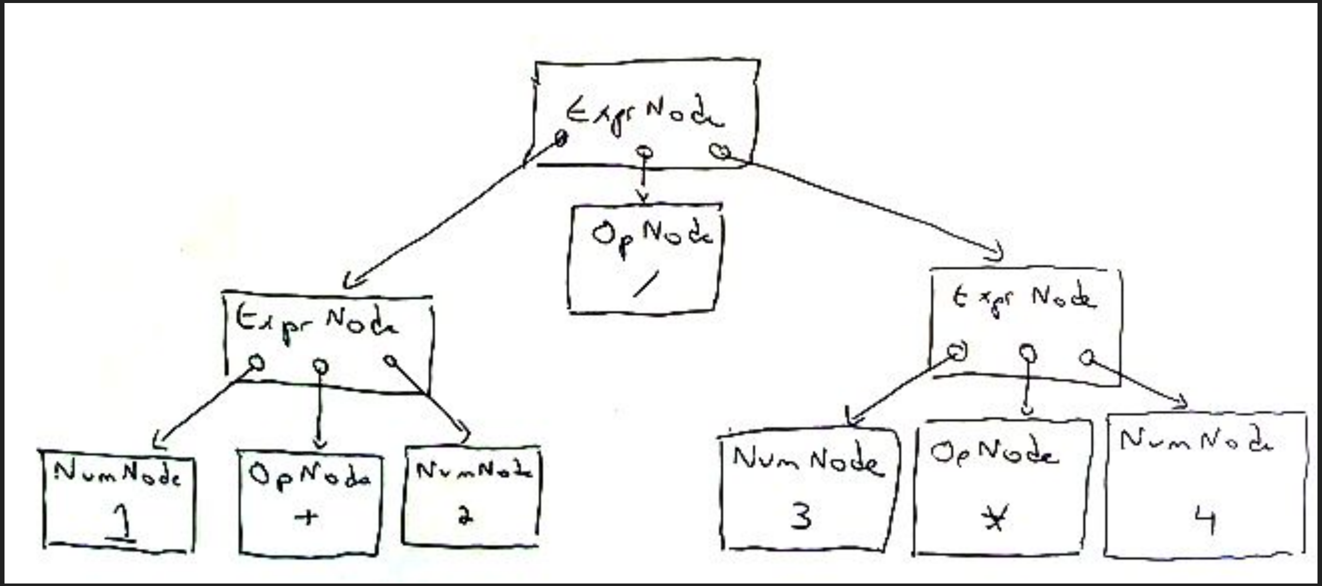
Semantics of an imperative programming language

Imperative languages are organised around **statements**, that abstract instructions that the computer should execute

The semantics in these language involve describing the meaning of things like the `while` and `return` keywords (changes control flow) and `global` (changes variable scoping)

```python
max_fib = 0

def fib(n):
    global max_fib
    a,b = 0,1

    i = 0
    while i < n:
        a, b = b, a+b
        i += 1
    max_fib = max(a, max_fib)
    return a

i = 0
while i < 15:
    print(i, fib(i), max_fib)
    i += 1

```

"expression-oriented programming"



Note that the root of the AST is a node designating that the tree represents an expression. This will be the standard model for the languages we cover in this course.

The semantics of "expression-oriented" languages

...we will consider only languages whose programs are *expressions*.

This allows us to simplify our discussion of semantics to mean "what value does the expression ultimately produce", and hence "what value does the program ultimately return to the user".

The semantics of the word "semantics"

We can actually mean a few different things here:

Definition: The **denotational semantics** of a programming language specifies the "value" or "output" from a program, given a formal definition

- For the purposes of this class, "formal definition" can mean "your intuition".
- How many different ways can you come up with to write an expression that ultimately evaluates to the integer `10`?

The semantics of the word "semantics"

We can actually mean a few different things here:

Definition: The **operational semantics** of a programming language describes the process by which the value of a program is computed (ie. *evaluated*).

Syntax and semantics for the **lambda calculus**

We'll conclude this lecture by applying what we've learned about studying the syntax and semantics of programming languages to the **lambda calculus**

Syntax and semantics for the **lambda calculus**

We'll conclude this lecture by applying what we've learned about studying the syntax and semantics of programming languages to the **lambda calculus**

The Lambda Calculus is a way of expressing computation invented by computer scientist Alonzo Church. Both its syntax and semantics were strong inspirations for the family of **functional programming languages**, which we will start using in earnest next week.

Alonzo Church

Alonzo Church (1903–1995)

lambda calculus

The lambda calculus is concerned with **denoting functions of one variable** (an "abstraction") and calling them with some value (a function **application)**.

Here's a program in the lambda calculus that returns whatever input it is supplied (the "identity function"):

$$(\lambda x . x)$$

lambda calculus

The lambda calculus is concerned with **denoting functions of one variable** (an "abstraction") and calling them with some value (a function **application**).

Here's a program in the lambda calculus that returns whatever input it is supplied (the "identity function"):

A function consuming one value called x

$$(\lambda \; x \; . \; x)$$

whose **body** simply produces its input

lambda calculus

The lambda calculus is concerned with **denoting functions of one variable** (an "abstraction") and calling them with some value (a function **application)**.

Here's how to **apply the previous function** to **some value:**

$$\underbrace{(\lambda x . x)}_{\text{The abstraction...}}\ 5$$

Applied to this value

The grammar of the lambda calculus

Note: We are glossing over a key insight into the lambda calculus for now. For these next few examples, assume that we have arithmetic operations as you would expect in a language like Python.

The lambda calculus does, indeed, support things like the natural numbers, but the way that they're constructed is non-obvious, so we will gloss over the details for now. For details, if you're interested, see "Implementing primitives in the Lambda Calculus", linked to on the course page.

The grammar of the (simplified) lambda calculus

We've said that a program in the lambda calculus is an **expression**. Therefore:

```
<expr> = IDENTIFIER
```

The grammar of the (simplified) lambda calculus

We've said that a program in the lambda calculus is an **expression**. Therefore:

```
<expr> = IDENTIFIER
       | "(" "λ" IDENTIFIER "." expr ")"  # function abstraction
```

The grammar of the (simplified) lambda calculus

We've said that a program in the lambda calculus is an **expression**. Therefore:

```
<expr> = IDENTIFIER
       | "(" "λ" IDENTIFIER "." expr ")"  # function abstraction
       | "(" expr expr ")"  # application: applies 2nd expr to 1st
```

The semantics of the (simplified) lambda calculus

We evaluate programs in the lambda calculus by **substitution** (so-called β-reduction). Recall that

- Every function ("abstraction") in the lambda-calculus takes exactly one arg
- The "application" part of the grammar hands exactly one expression to the fn

Consider the following (slightly cheating) expression:

$$(\lambda\ x\ .\ x+1)\ 5$$

The semantics of the (simplified) lambda calculus

We evaluate programs in the lambda calculus by **substitution** (so-called β-reduction).

Therefore, we **substitute the argument passed to the abstraction inside the body of the function, and then recursively evaluate the new expression**

$$(\lambda \ x \ . \ x+1) \ 5$$

The semantics of the (simplified) lambda calculus

We evaluate programs in the lambda calculus by **substitution** (so-called β-reduction).

Therefore, we **substitute the argument passed to the abstraction inside the body of the function, and then recursively evaluate the new expression**

$$(\lambda x . x+1)\ 5$$

$$5+1$$

The semantics of the (simplified) lambda calculus

We evaluate programs in the lambda calculus by **substitution** (so-called β-reduction).

Therefore, we **substitute the argument passed to the abstraction inside the body of the function, and then recursively evaluate the new expression**

```
( λ x . x+1) 5

5+1 => 6
```

Something slightly more interesting...

$$(\lambda x . \lambda y . \sqrt{(x^2 + y^2)})$$

Something slightly more interesting...

$$(\underbrace{\lambda\ x}_{\text{The abstraction is a function of one argument}} \ . \ \underbrace{\lambda\ y\ .\ \sqrt{}(x\hat{}2 + y\hat{}2))}_{\text{That returns another function, also of one argument}}$$

Something slightly more interesting...

$$(\lambda x . \lambda y . \sqrt{(x^2 + y^2)})$$

Even though x is not a variable in the inner lambda expression, it is "in scope" in the outer one"

How do we evaluate this?

```
( λ x . λ y . √(x^2 + y^2))
```

How do we evaluate this?

$$(\lambda x . \lambda y . \sqrt{(x^2 + y^2)}) \underbrace{3 \ 4}_{}$$

Calling the function with "two arguments"

How do we evaluate this?

$$(\lambda x . \lambda y . \sqrt{(x^2 + y^2)})\ 3\ 4$$

How do we evaluate this?

$$(\lambda x . \lambda y . \sqrt{(x^2 + y^2)})\ 3\ 4$$

$$(\lambda y . \sqrt{(3^2 + y^2)})\ 4$$

How do we evaluate this?

$$(\lambda x . \lambda y . \sqrt{(x^2 + y^2)})\ 3\ 4$$

$$(\lambda y . \sqrt{(3^2 + y^2)})\ 4$$

The abstraction is a function of one argument

All occurrences of x have been replaced in the previous application; `3` can be thought of as a "constant" now.

How do we evaluate this?

$$(\lambda x . \lambda y . \sqrt{(x^2 + y^2)})\ 3\ 4$$

$$(\lambda y . \underbrace{\sqrt{(3^2 + y^2)}\ 4}_{})$$

The abstraction is a function of one argument

All occurrences of x have been replaced in the previous application; \`3\` can be thought of as a "constant" now.

How do we evaluate this?

$$(\lambda x . \lambda y . \sqrt{(x^2 + y^2)})\ 3\ 4$$

$$(\lambda y . \sqrt{(3^2 + y^2)})\ 4$$

$$\sqrt{(3^2 + 4^2)}$$

How do we evaluate this?

$$(\lambda x.\ \lambda y.\ \sqrt{(x^2 + y^2)})\ 3\ 4$$

$$(\lambda y.\ \sqrt{(3^2 + y^2)})\ 4$$

$$\sqrt{(3^2 + 4^2)}$$

$$5$$

We have intentionally skipped over how to define things like numbers and booleans, or describe recursion, in the lambda calculus; right now we are only interested in **syntax and evaluation**

We will return periodically to the lambda calculus throughout this class, as we learn new concepts, though! Stay tuned for more fun with lambdas………..

Next week:

We'll start writing actual code!

First lab session on Monday