CS 3110

Interpreters

Prof. Clarkson
Fall 2019

Today's music: *Step by Step* by New Kids on the Block

CLICKER QUESTION 1

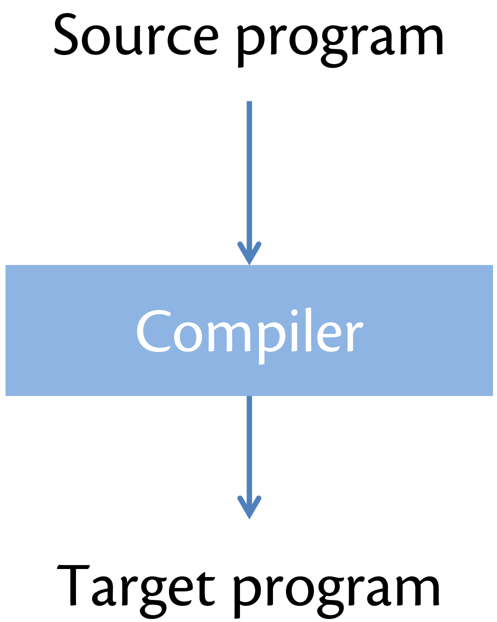
Review

Previously in 3110:

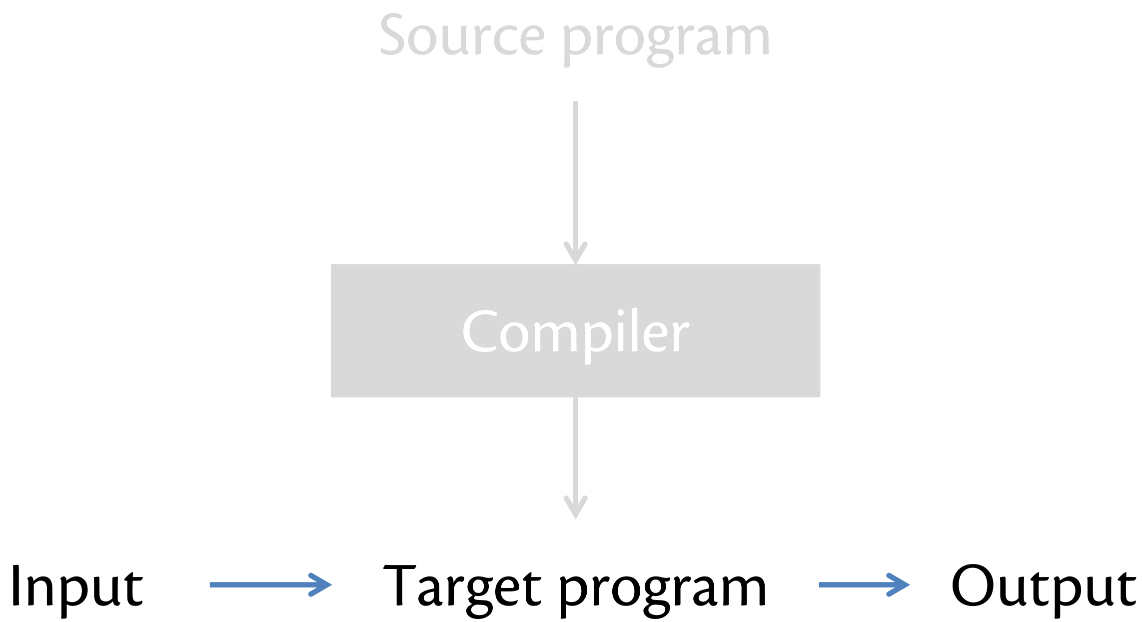
- functional programming
- modular programming
- efficiency

Today:

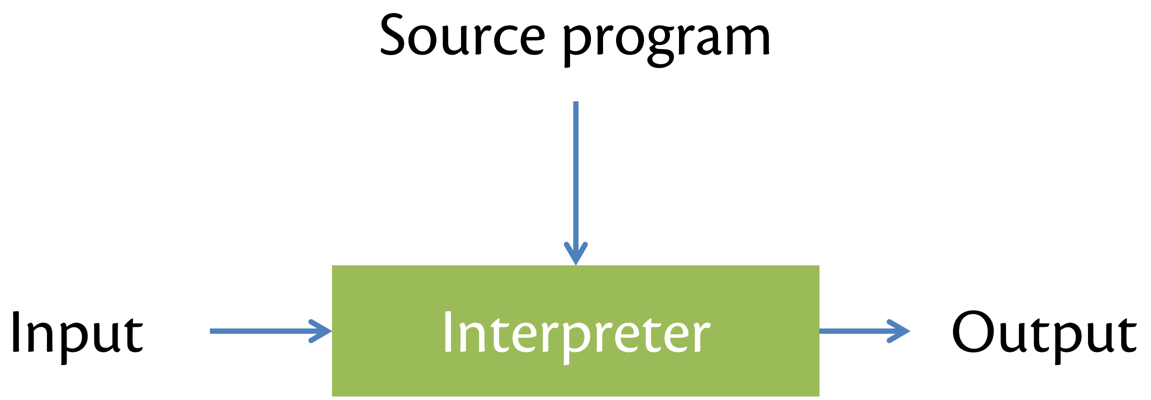
- new unit of course: interpreters



code as data: the compiler is code that operates on data; that data is itself code



the compiler goes away; not needed to run the program



the interpreter stays; needed to run the program

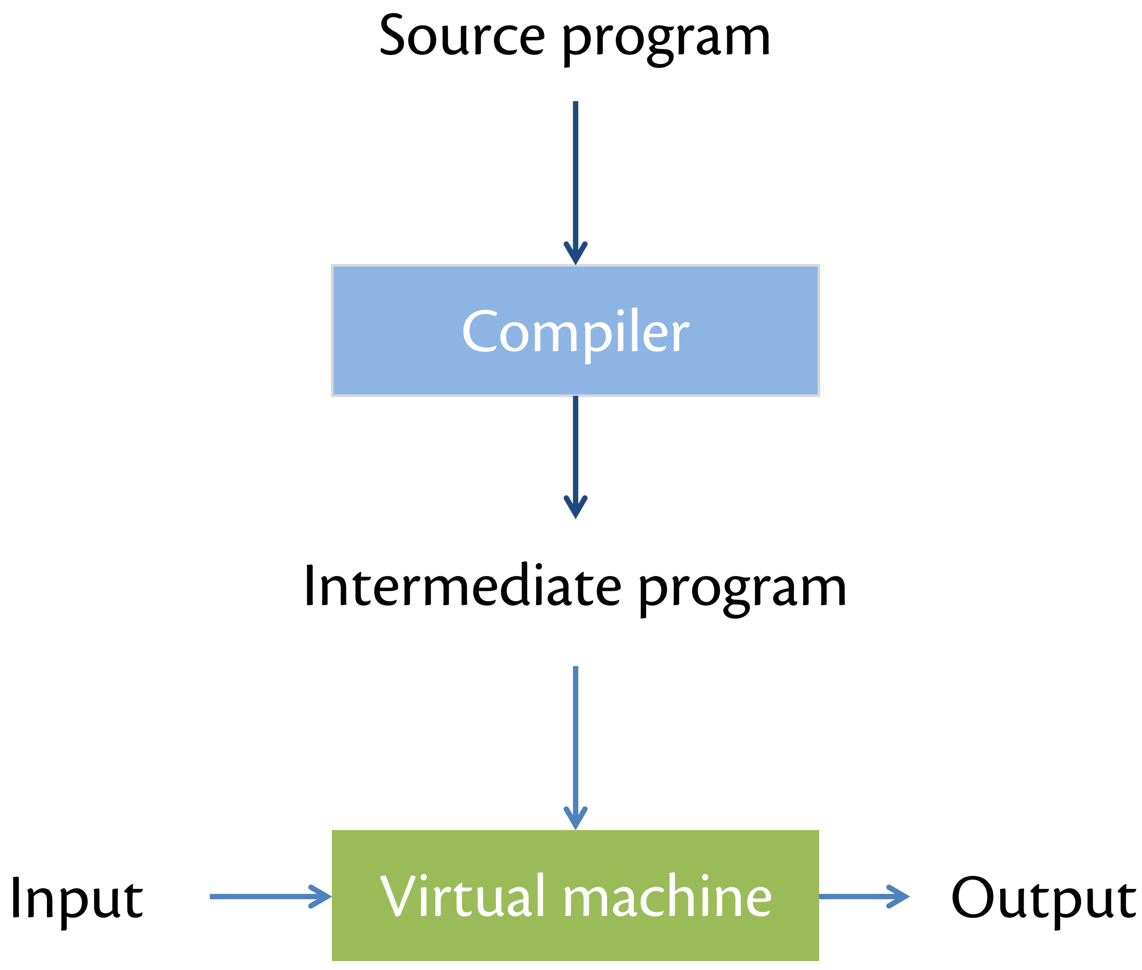
Compilers:

- primary job is *translation*
- better performance

vs.

Interpreters:

- primary job is *execution*
- easier implementation


Source program
Compiler
Intermediate program
Input
Virtual machine
Output

Architecture

Two phases:

- **Front end:** translate source code into *abstract syntax tree* (AST) then into *intermediate representation* (IR)
- **Back end:** translate AST into machine code

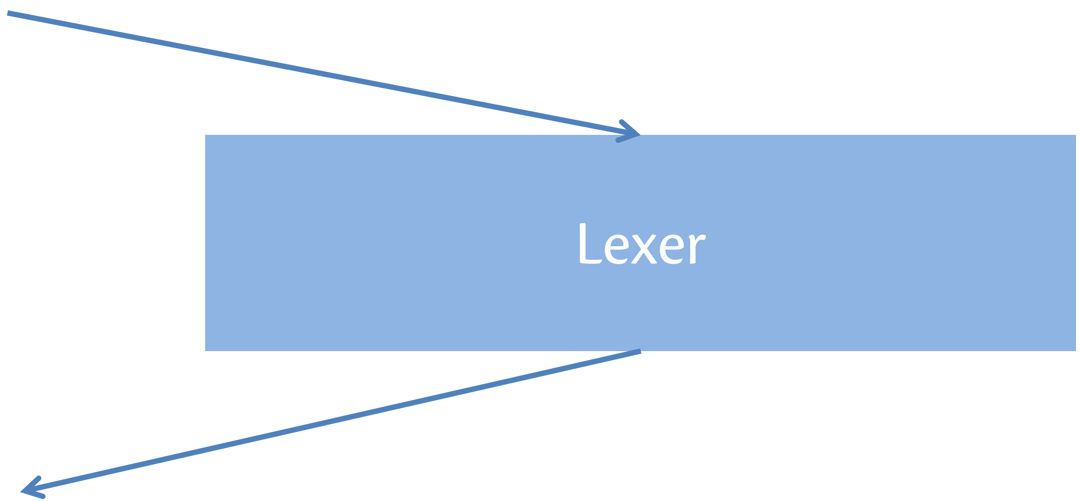
Front end of compilers and interpreters largely the same:

- *Lexical analysis* with lexer
- *Syntactic analysis* with parser
- *Semantic analysis*

Front end

Character stream:

```
if x=0 then 1 else fact(x-1)
```

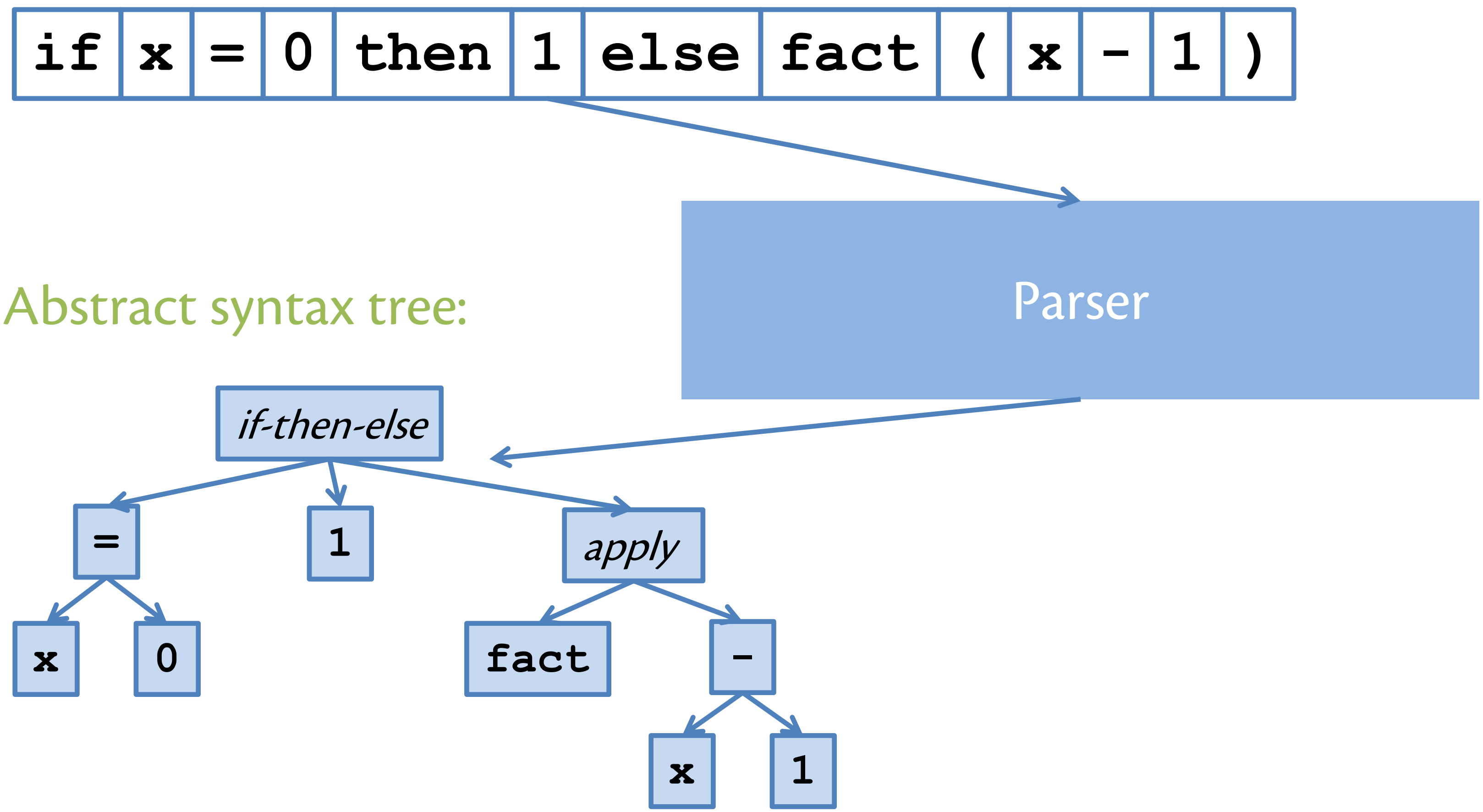


Token stream:

`if`	`x`	`=`	`0`	`then`	`1`	`else`	`fact`	`(`	`x`	`-`	`1`	`)`

Front end

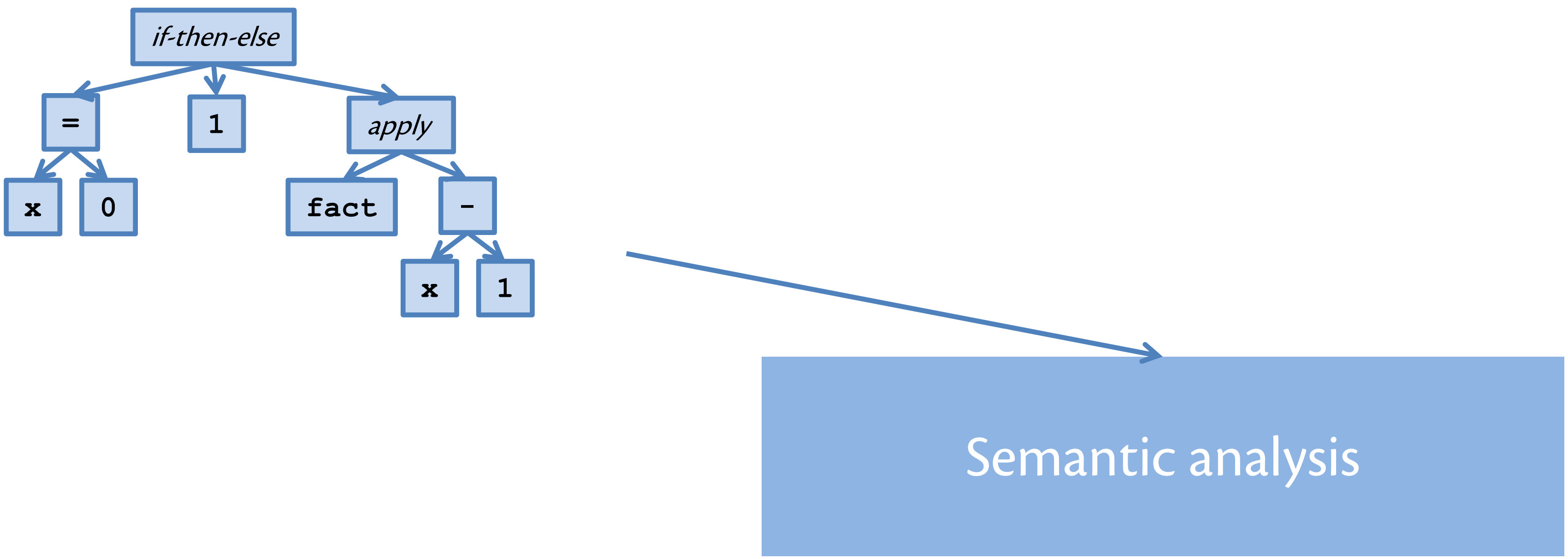
Token stream:



Abstract syntax tree:

Front end

Abstract syntax tree:



- accept or reject program
- create *symbol tables* mapping identifiers to types
- *decorate* AST with types
- etc.

Next

Might translate AST into a *intermediate representation* (IR) that is a kind of abstract machine code

Then:

- **Interpreter** executes AST or IR
- **Compiler** translates IR into machine code

Implementation

Functional languages are well-suited to implement compilers and interpreters

- **Code** easily represented by tree data types
- **Compilation/execution** easily defined by pattern matching on trees

Extended demo: A calculator

- 22
- 11 + 11
- (10 + 1) + (5 + 6)
- 2 * 11
- 2 + 2 * 10
- 2 * 2 + 10
- 2 * 2 * 10
- etc.

- Integers
- Addition
- Multiplication
- Parentheses
- Whitespace

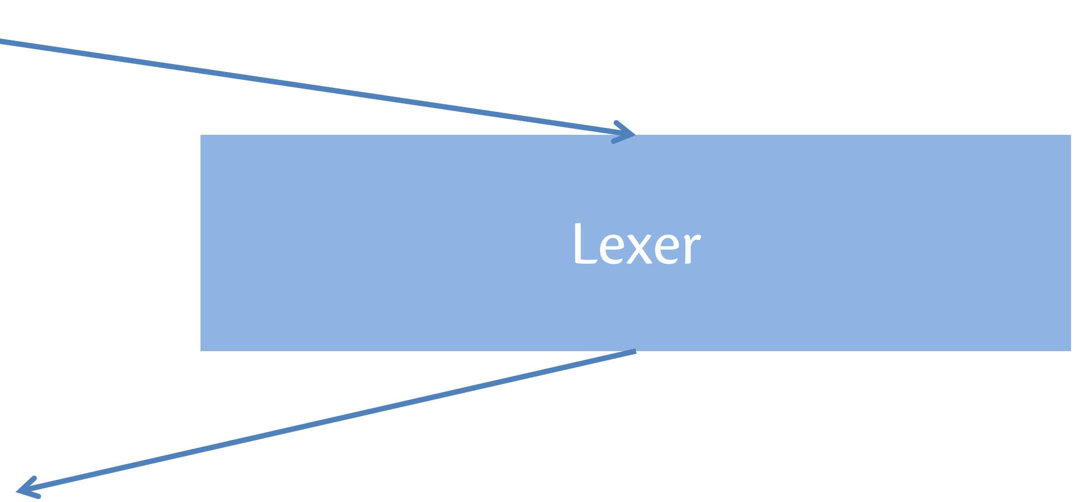
Goal: transform input string to output string

LEXING

Lexer

Character stream:

```
(10 + 1) + (5 + 6)
```



Token stream:

(	10	+	1	)	+	(	5	+	6	)

Tokens

- integer literals: 0, 10, -22, etc.
- +
- *
- (
-)
- whitespace: irrelevant

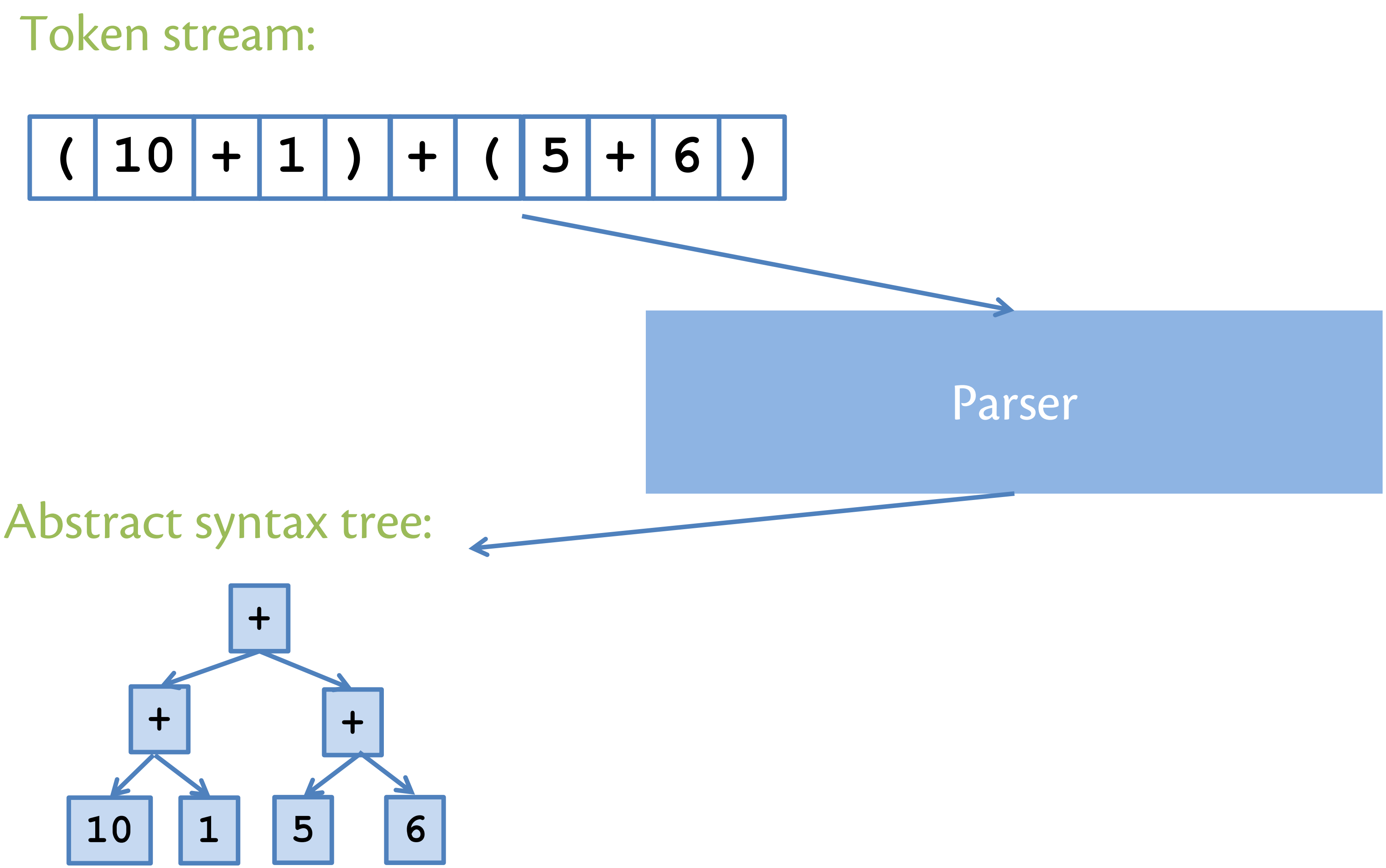
How to describe: regular expressions!

Tool: ocamllex (.mll files)

Demo

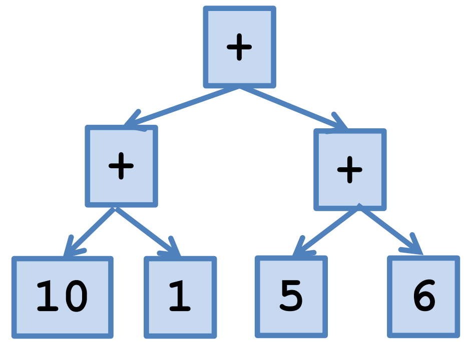
PARSING

Parser

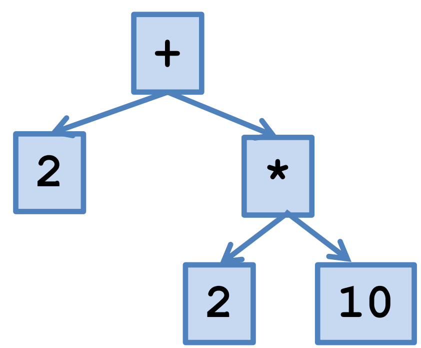


Concrete vs. abstract syntax

`(10 + 1) + (5 + 6)`



`2 + 2 * 10`



Parentheses: irrelevant

Operators: have precedence

CLICKER QUESTION 2

Grammar

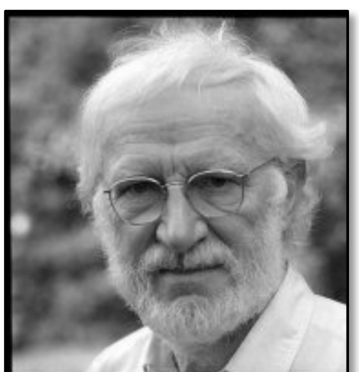
```
e ::= i
    | e1 bop e2
    | ( e )

bop ::= + | *

i ::= integers
```

Backus-Naur Form (BNF)

John Backus (1924-2007)
ACM Turing Award Winner 1977
"For profound, influential, and lasting contributions to the design of practical high-level programming systems"



Peter Naur (1928-2016)
ACM Turing Award Winner 2005
"For fundamental contributions to programming language design"

Grammar

```
e ::= i
    | e1 bop e2
    | ( e )

bop ::= + | *

i ::= integers
```

- How to describe: type for AST, and production rules
- Tool: ocamlyacc or menhir (.mly files)

Demo

AST vs BNF: note similarity

```
e ::= i | e1 bop e2 | ( e )

type expr =
  | Int of int
  | Binop of expr * expr



bop ::= + | *

type bop =
  | Add
  | Mult
```

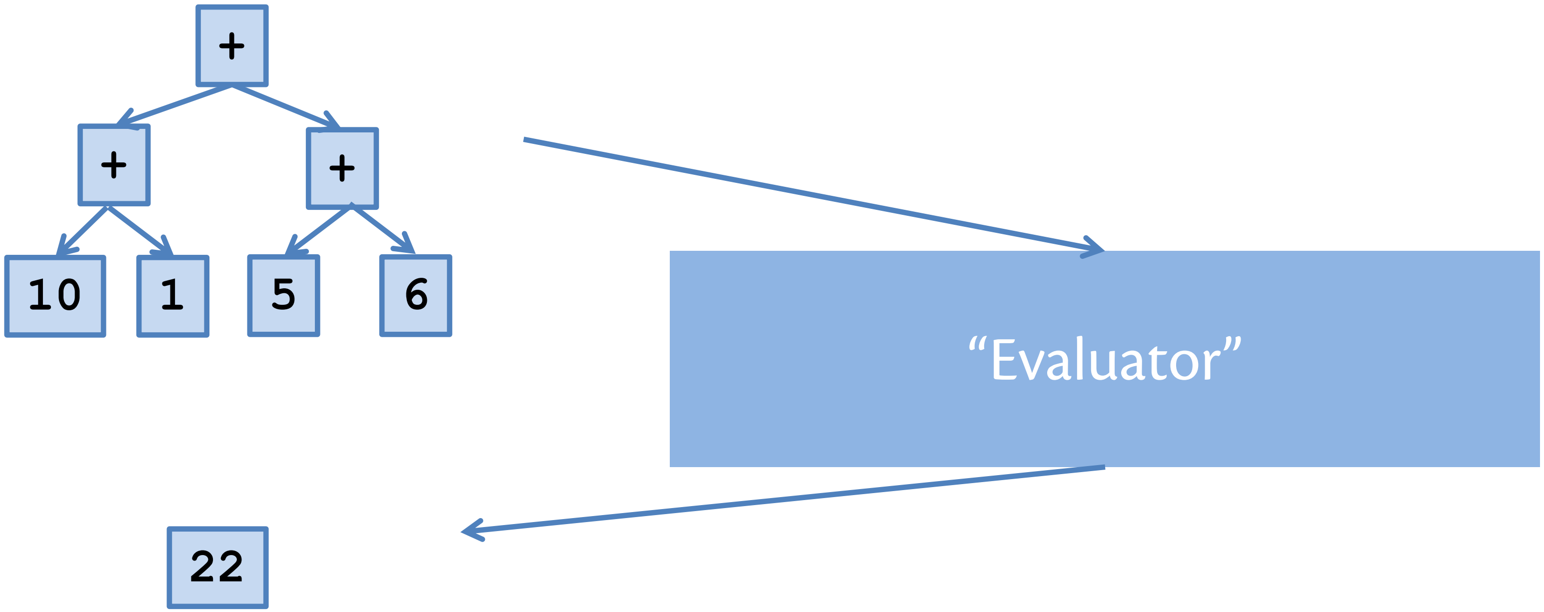
We will skip this until the end of this unit of course

TYPE CHECKING

EVALUATION

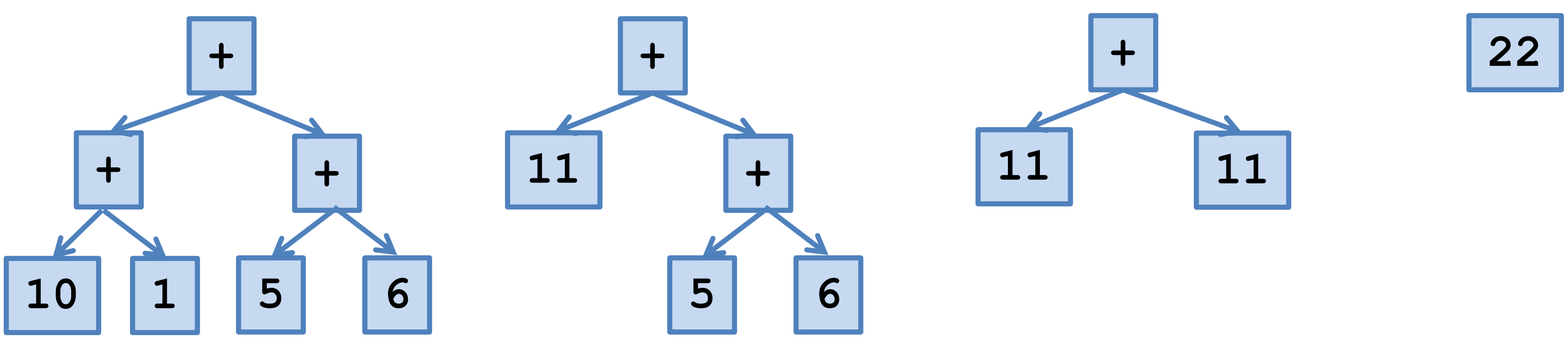
Evaluation

Abstract syntax tree:



Evaluation strategy

- An expression e takes a single *step* to a new expression e’ by simplifying some subexpression
- Expression keeps stepping until it reaches a *value*
- Values never step further



Demo

Upcoming events

- [last night] R7 due
- [tomorrow] A5 due

This is a step forward.

THIS IS 3110