

UMass Boston Computer Science
CS450 High Level Languages (section 2)
Interpreters and “eval”

Wednesday, November 6, 2024

Logistics

- HW 9 “out”
 - submit: in-class work 11/4 and 11/6
 - due: Mon 11/11 12pm (noon) EST
- HW 10 – extension of “hw9”
 - Out: Tue 11/5
 - due: Mon 11/18 12pm (noon) EST
- no lecture: Veteran’s Day Mon 11/11

Syntax vs Semantics (Spoken Language)

Syntax

- Specifies: **valid language constructs**
 - E.g., sentence = (subject) noun + verb + (object) noun

“the ball threw the child”

- Syntactically: **valid!**
- Semantically: ???

Semantics

- Specifies: “meaning” of language (constructs)

Syntax vs Semantics (Programming Language)

Syntax

- Specifies: valid language constructs
 - E.g., valid **Racket** program: s-expressions
 - Valid **python** program: follows python grammar (including whitespace!)

Semantics

- Specifies: “meaning” of language (constructs)

Syntax vs Semantics (Programming Language)

Syntax

- Specifies: valid language constructs
 - E.g., valid **Racket** program: s-expressions
 - Valid **python** program: follows python grammar (including whitespace!)

Semantics

- Specifies: “meaning” of language (constructs)

Q: What is the “meaning” of a program?

A: The result of “running” it!

... but how does a program “run”?

Programs run on CPUs

```
00000000 0000 0001 0001 1010 0010 0001 0004 0128
00000010 0000 0016 0000 0028 0000 0010 0000 0020
00000020 0000 0001 0004 0000 0000 0000 0000 0000
00000030 0000 0000 0000 0010 0000 0000 0000 0204
00000040 0004 8384 0084 c7c8 00c8 4748 0048 e8e9
00000050 00e9 6a69 0069 a8a9 00a9 2828 0028 fdfc
00000060 00fc 1819 0019 9898 0098 d9d8 00d8 5857
00000070 0057 7b7a 007a bab9 00b9 3a3c 003c 8888
00000080 8888 8888 8888 8888 288e be88 8888 8888
00000090 3b83 5788 8888 8888 7667 778e 8828 8888
000000a0 d61f 7abd 8818 8888 467c 585f 8814 8188
000000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988
000000c0 8a18 880c e841 c988 b328 6871 688e 958b
000000d0 a948 5862 5884 7e81 3788 1ab4 5a84 3eec
000000e0 3d86 dcb8 5cbb 8888 8888 8888 8888 8888
000000f0 8888 8888 8888 8888 8888 8888 8888 0000
00001000 0000 0000 0000 0000 0000 0000 0000 0000
*
00001300 0000 0000 0000 0000 0000 0000 0000 0000
000013e0
```

Machine code



cpu



“result”

“low level”

Q: What is the “meaning” of a program?

A: The result of “running” it!

... but how does a program “run”?

Programmers don’t write machine code!

Running Programs: `eval`

```
;; eval : Program -> Result  
;; “runs” a given “program”, producing a “result”
```

More generally:

An **interpreter**, i.e., an “**eval**” function, turns a “**program**” into a “**result**”

(But programs are usually not directly interpreted either)

More commonly, a high-level program is first **compiled** to a lower-level language (and then interpreted)

Q: What is the “meaning” of a program?

A: The result of “running” it!

... but how does a program “run”?

From
Lecture 1

“high” level
(easier for humans
to understand)

“declarative”

“imperative”

NOTE: This hierarchy is approximate

English	
Specification langs	Types? pre/post cond?
Markup (html, markdown)	tags
Database (SQL)	queries
Logic Program (Prolog)	relations
Lazy lang (Haskell, R)	Delayed computation
Functional lang (Racket)	Expressions (no stmts)
JavaScript, Python	“eval”
C# / Java	GC (no alloc, ptrs)
C++	Classes, objects
C	Scoped vars, fns
Assembly Language	Named instructions
Machine code	Binary

More commonly, a
high-level program is
first **compiled** to a
lower-level language
(and then **interpreted**)

(runs on cpu)

“high” level
(easier for humans
to understand)

surface language

“declarative”
compiler

target language

More commonly, a
high-level program is
first **compiled** to a
lower-level language
(and then **interpreted**)
(runs on cpu)

“imperative”

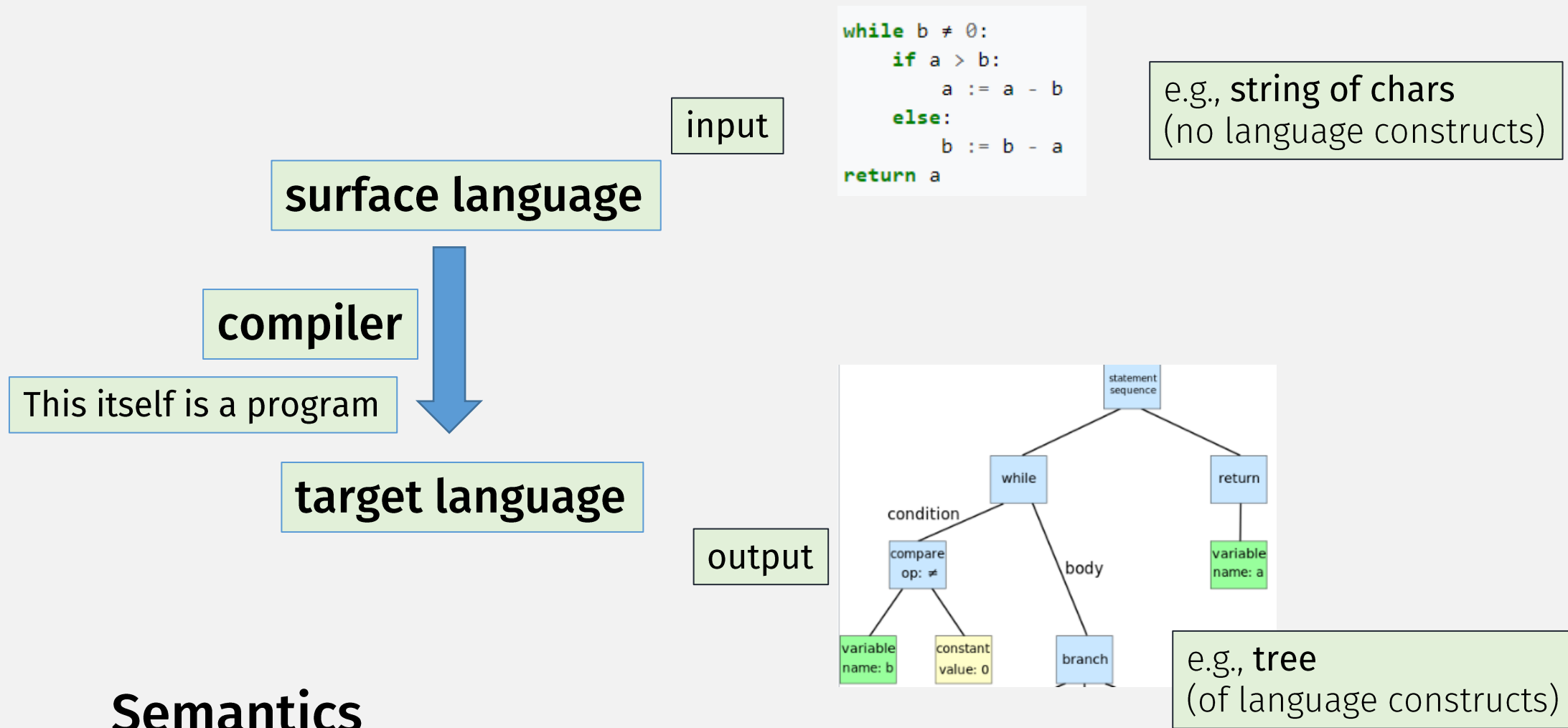
Specification langs
Markup (html, markdown)
Database (SQL)
Logic Program (Prolog)
Lazy lang (Haskell, R)
Functional lang (Racket)
JavaScript, Python
C# / Java
C++
C
Assembly Language
Machine code

Common **target** languages:

- bytecode (e.g., JS, Java)
- assembly
- machine code

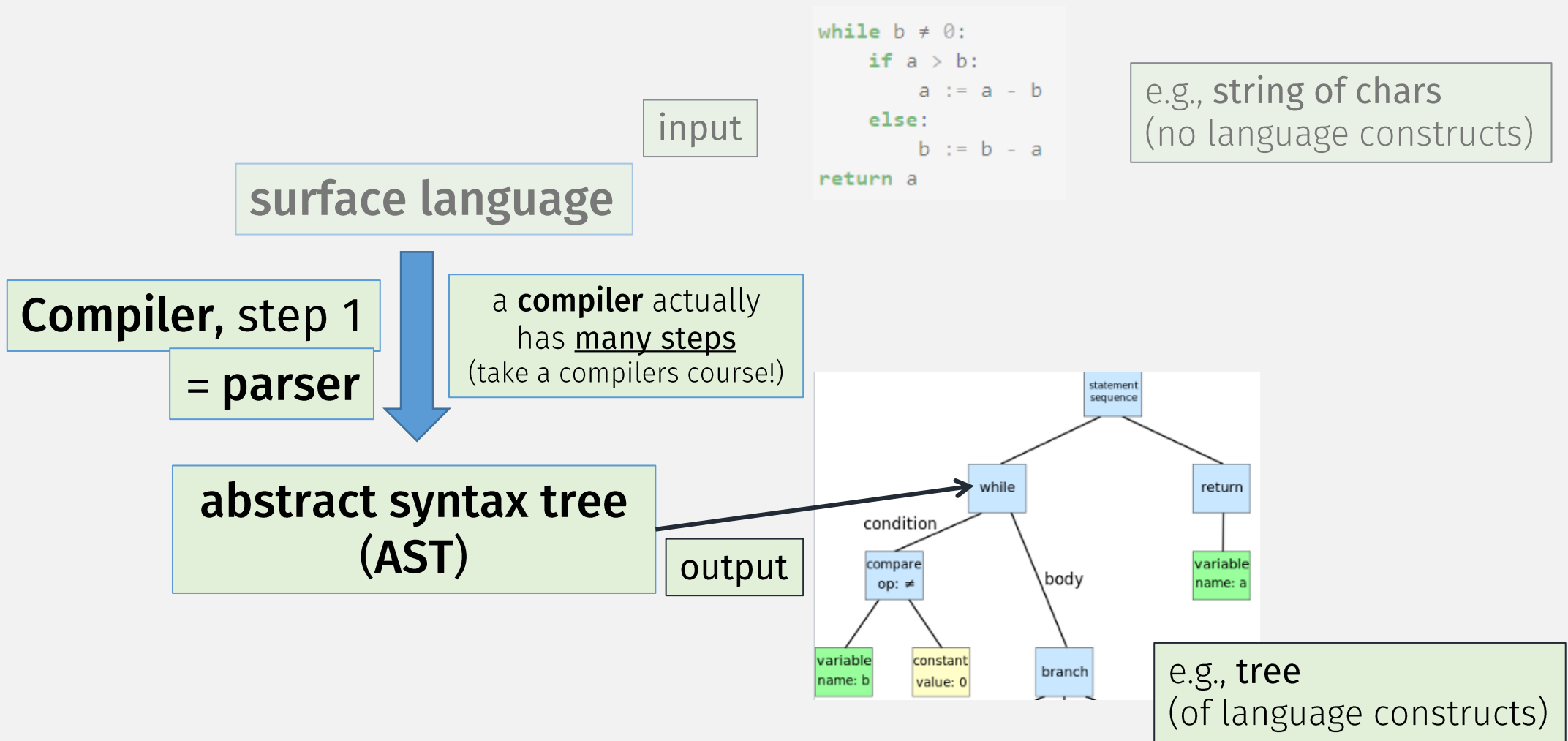
A **virtual machine** is just a
bytecode interpreter

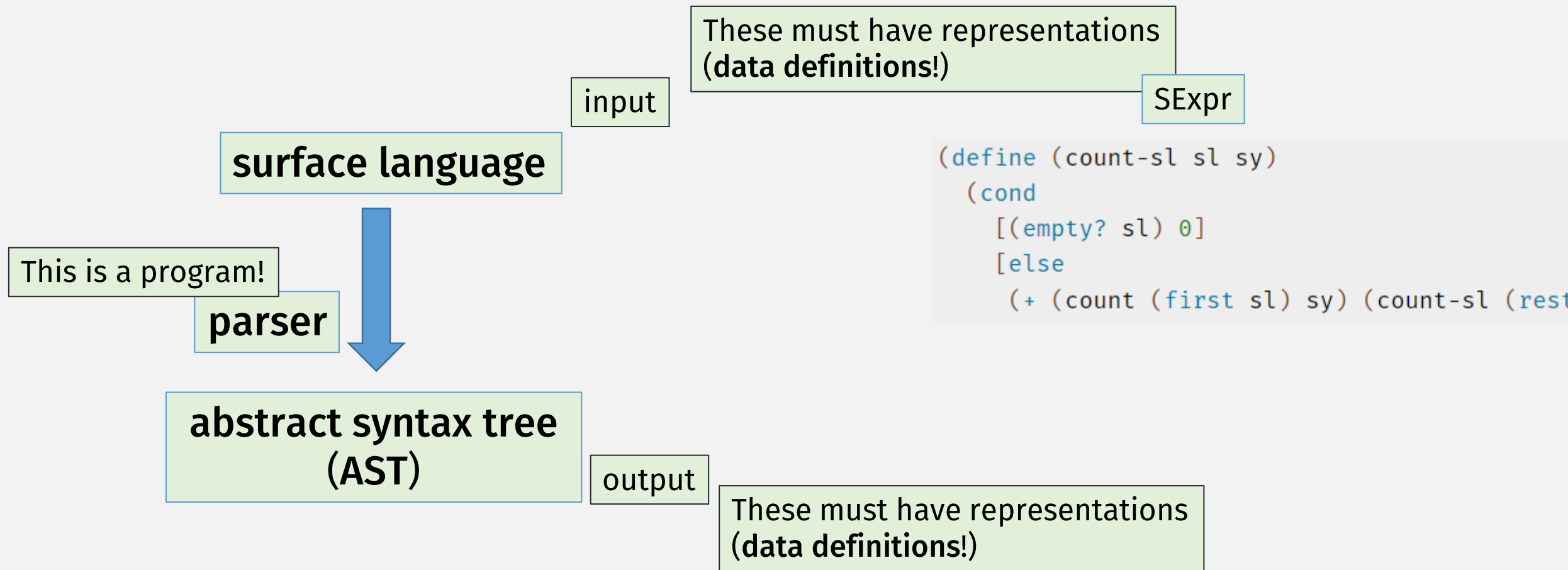
(A (hardware) **CPU** is just a
machine code interpreter)!

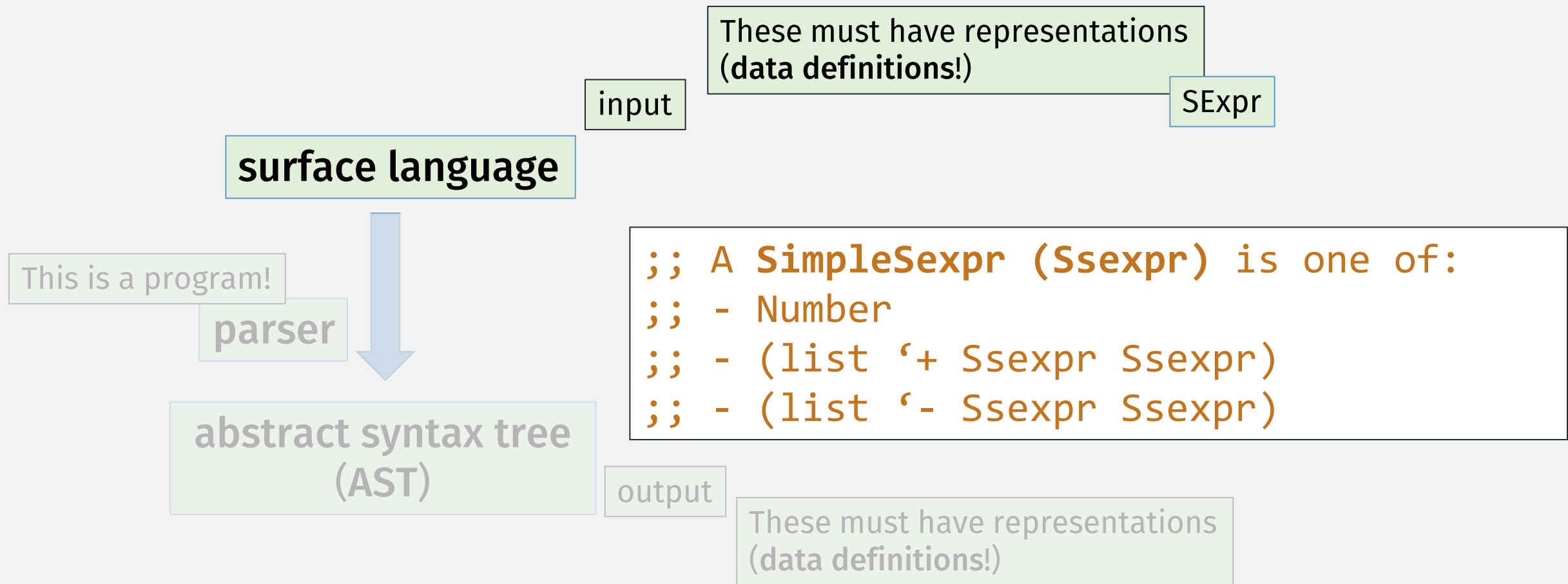


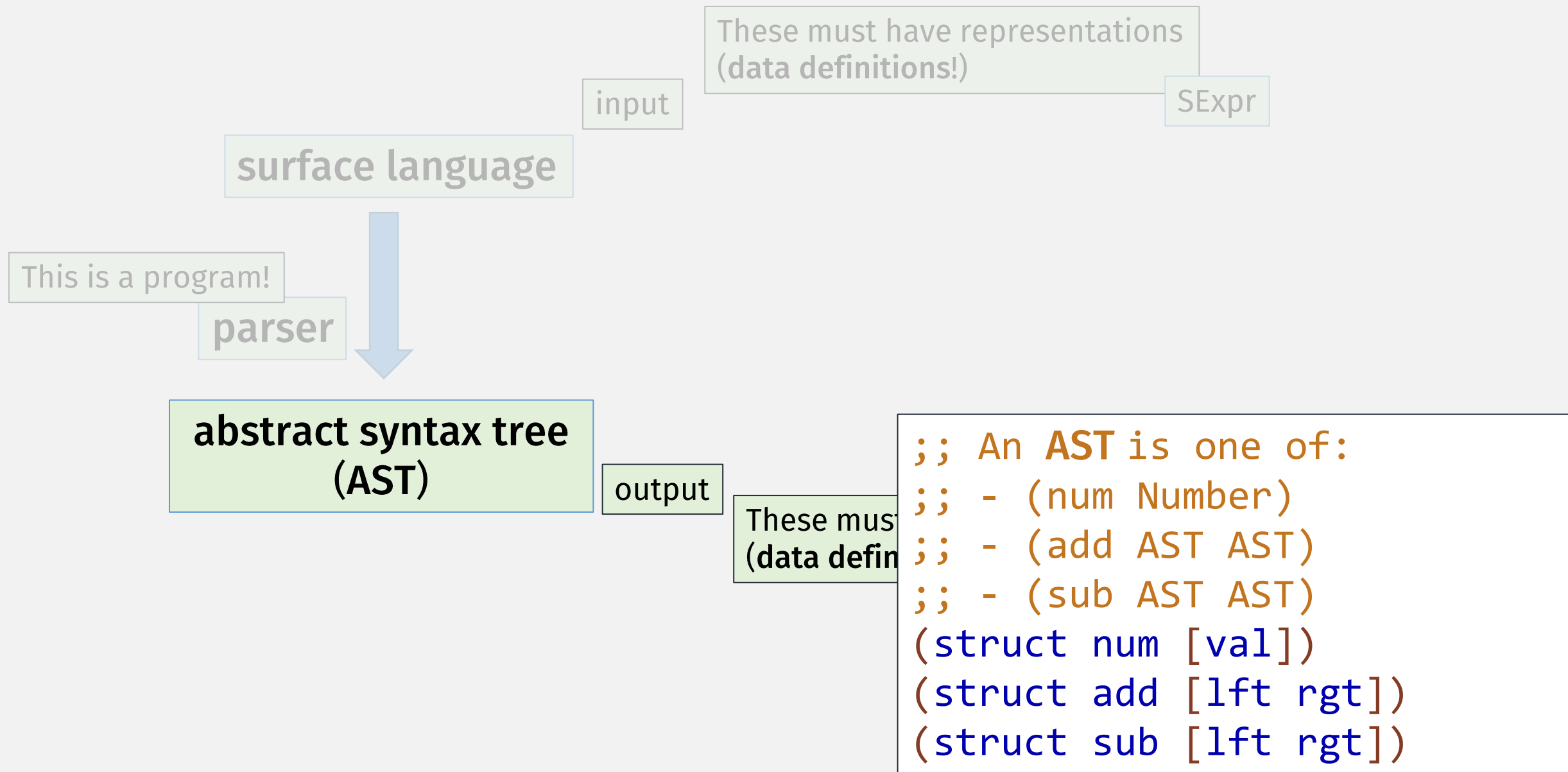
Semantics

- Specifies: meaning of language constructs
- So: to “run” a program, we need to construct the constructs first





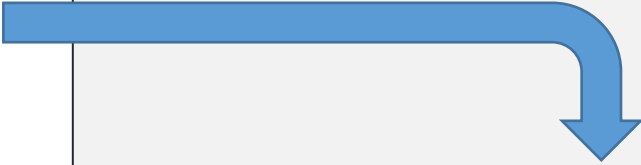




In-class Coding 11/4: parser

```
;; parse: SimpleSexpr -> AST
;; Converts a (simple) S-expression to a language AST
```

```
;; A SimpleSexpr (Ssexpr) is a:
;; - Number
;; - (list '+ Ssexpr Ssexpr)
;; - (list '- Ssexpr Ssexpr)
```

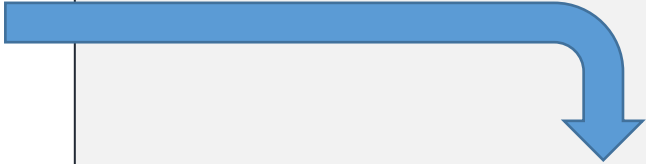


```
;; An AST is one of:
;; - (num Number)
;; - (add AST AST)
;; - (sub AST AST)
(struct num [val])
(struct add [lft rgt])
(struct sub [lft rgt])
```

In-class Coding 11/4: parser

```
;; parse: SimpleSexpr -> AST
;; Converts a (simple) S-expression to a language AST
```

```
;; A SimpleSexpr (Ssexpr) is a:
;; - Number
;; - (list '+ Ssexpr Ssexpr)
;; - (list '- Ssexpr Ssexpr)
```



```
(define (parse s)
  (match s
    [(? number?) ... ]
    [`(+ ,x ,y)
     ... (parse x) ... (parse y) ... ]
    [`(- ,x ,y)
     ... (parse x) ... (parse y) ... ])))
```

TEMPLATE

```
;; An AST is one of:
;; - (num Number)
;; - (add AST AST)
;; - (sub AST AST)
(struct num [val])
(struct add [lft rgt])
(struct sub [lft rgt])
```


In-class Coding 11/4: parser

```
;; parse: SimpleSexpr -> AST
;; Converts a (simple) S-expression to a language AST
```

```
;; A SimpleSexpr (Ssexpr) is a:
;; - Number
;; - (list '+ Ssexpr Ssexpr)
;; - (list '- Ssexpr Ssexpr)
```

```
(define (parse s)
  (match s
    [(? number?) (num s)]
    [`(+ ,x ,y)
     ... (parse x) ... (parse y) ... ]
    [`(- ,x ,y)
     ... (parse x) ... (parse y) ... ])))
```

```
;; An AST is one of:
;; - (num Number)
;; - (add AST AST)
;; - (sub AST AST)
(struct num [val])
(struct add [lft rgt])
(struct sub [lft rgt])
```

In-class Coding 11/4: parser

```
;; parse: SimpleSexpr -> AST
;; Converts a (simple) S-expression to a language AST
```

```
;; A SimpleSexpr (Ssexpr) is a:
;; - Number
;; - (list '+ Ssexpr Ssexpr)
;; - (list '- Ssexpr Ssexpr)
```

```
(define (parse s)
  (match s
    [(? number?) (num s)]
    [`(+ ,x ,y)
     (add (parse x) (parse y))]
    [`(- ,x ,y)
     ... (parse x) ... (parse y) ... ]))
```

```
;; An AST is one of:
;; - (num Number)
;; - (add AST AST)
;; - (sub AST AST)
(struct num [val])
(struct add [lft rgt])
(struct sub [lft rgt])
```

In-class Coding 11/4: parser

```
;; parse: SimpleSexpr -> AST
;; Converts a (simple) S-expression to a language AST
```

```
;; A SimpleSexpr (Ssexpr) is a:
;; - Number
;; - (list '+ Ssexpr Ssexpr)
;; - (list '- Ssexpr Ssexpr)
```

```
(define (parse s)
  (match s
    [(? number?) (num s)]
    [`(+ ,x ,y)
     (add (parse x) (parse y))]
    [`(- ,x ,y)
     (sub (parse x) (parse y))]))
```

```
;; An AST is one of:
;; - (num Number)
;; - (add AST AST)
;; - (sub AST AST)
(struct num [val])
(struct add [lft rgt])
(struct sub [lft rgt])
```

In-class Coding 11/4 #2: ~~eval-ast~~ run

```
;; eval-ast run : AST -> Result  
;; computes the result of given program AST
```

↓

```
;; A Result is one of:  
;;  
;; ???  
;;
```

```
;; An AST is one of:  
;; - (num Number)  
;; - (add AST AST)  
;; - (sub AST AST)
```

TEMPLATE

```
(define (eval-ast run p)  
  (cond  
    [(num? p) ... (num-val p) ... ]  
    [(add? p) ... (run (add-lft p)) ...  
                 ... (run (add-rgt p)) ... ]  
    [(sub? p) ... (run (sub-lft p)) ...  
                 ... (run (sub-rgt p)) ... ]))
```


In-class Coding 11/4 #2: run

```
;; run: AST -> Result
;; computes the result of given program AST
```

```
;; A Result is a:
;; - Number
```

```
(define (run p)
  (cond
    [(num? p) p]
    [(add? p) ... (run (add-lft p)) ...
              ... (run (add-rgt p)) ... ]
    [(sub? p) ... (run (sub-lft p)) ...
              ... (run (sub-rgt p)) ... ]))
```

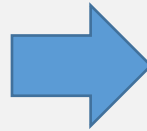
```
;; An AST is one of:
;; - (num Number)
;; - (add AST AST)
;; - (sub AST AST)
```



In-class Coding 11/4 #2: run

```
;; eval-ast: AST -> Result
;; computes the result of given program AST
```

```
;; An AST is one of:
;; - (num Number)
;; - (add AST AST)
;; - (sub AST AST)
```



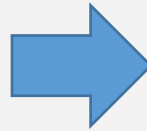
```
;; A Result is a:
;; - Number
```

```
(define (run p)
  (cond
    [(num? p) p]
    [(add? p) (+ (run (add-lft p))
                  (run (add-rgt p)))]
    [(sub? p) ... (run (sub-lft p)) ...
               ... (run (sub-rgt p)) ... ]))
```

In-class Coding 11/4 #2: run

```
;; run: AST -> Result
;; computes the result of given program AST
```

```
;; An AST is one of:
;; - (num Number)
;; - (add AST AST)
;; - (sub AST AST)
```



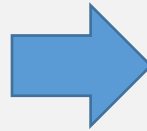
```
;; A Result is a:
;; - Number
```

```
(define (run p)
  (cond
    [(num? p) p]
    [(add? p) (+ (run (add-lft p))
                  (run (add-rgt p)))]
    [(sub? p) ... (run (sub-lft p)) ...
                ... (run (sub-rgt p)) ... ]))
```

In-class Coding 11/4 #2: run

```
;; run: AST -> Result
;; computes the result of given program AST
```

```
;; An AST is one of:
;; - (num Number)
;; - (add AST AST)
;; - (sub AST AST)
```



```
;; A Result is a:
;; - Number
```

```
(define (run p)
  (cond
    [(num? p) p]
    [(add? p) (+ (run (add-lft p))
                  (run (add-rgt p)))]
    [(sub? p) (- (run (sub-lft p))
                  (run (sub-rgt p)))]))
```


The “CS450 LANG” Programming Language

cs450lang.rkt

```
;; parse : Ssexpr -> AST
(define (parse ssexpr) ... )
;; run: AST -> Result
(define (run ast) ... )

(define (eval450 p) (compose run parse))

(define-macro (module expr ...)
  (eval450 expr) ...))
```

`((compose f g) x) = (f (g x))`

cs450lang-prog.rkt

#lang s-exp “cs450lang.rkt”

`(+ 1 2) ; => 3`

A program! written
in “CS450 LANG”!

“CS450 Lang” Demo

- See `cs450f24/in-class-11-06` github repository

The “CS450 LANG + STRINGS” PL

```
;; A Ssexpr is a:  
;; - Number  
;; - (list '+ Ssexpr Ssexpr)  
;; - (list '- Ssexpr Ssexpr)
```



```
;; A 450LangExpr (Expr) is a:  
;; - Number  
;; - String  
;; - (list '+ Expr Expr)  
;; - (list '- Expr Expr)
```

```
;; An AST is one of:  
;; - (num Number)  
;; - (add AST AST)  
;; - (sub AST AST)  
(struct num [val])
```

```
;; A 450LangAST (AST) is:  
;; - (num Number)  
;; - (str String)  
;; - (add AST AST)  
;; - (sub AST AST)  
(struct num [val])  
(struct str [val])  
(struct add [lft rgt])  
(struct sub [lft rgt])
```

Parsing “CS450 LANG + STRINGS” Programs

```
;; parse: Expr -> AST
```

```
;; Converts a “CS450 Lang” S-expression to AST
```

```
(define (parse s)
  (match s
    [(? number?) (num s)]
    [(? string?) (str s)]
    [`(+ ,x ,y) (add (parse x) (parse y))]
    [`(- ,x ,y) (sub (parse x) (parse y))])
```

```
;; A 450LangExpr (Expr) is a:
```

```
;; - Number
```

```
;; - String
```

```
;; - (list '+ Expr Expr)
```

```
;; - (list '- Expr Expr)
```

```
;; A 450LangAST (AST) is:
```

```
;; - (num Number)
```

```
;; - (str String)
```

```
;; - (add AST AST)
```

```
;; - (sub AST AST)
```

```
(struct num [val])
```

```
(struct str [val])
```

```
(struct add [lft rgt])
```

```
(struct sub [lft rgt])
```


Running “CS450 Lang + Strings” Programs

```
;; run: AST -> Result  
;; computes the result of given program AST
```

```
;; A Result is a:  
;; - Number  
;; - String
```

```
;; An AST is one of:  
;; - (num Number)  
;; - (str String)  
;; - (add AST AST)  
;; - (sub AST AST)
```

```
(define (run p)  
  (cond  
    [(num? n) n]  
    [(str? s) s]  
    [(add? p) (???? (run (add-lft p)) (run (add-rgt p)))]  
    [(sub? p) (???? (run (sub-lft p)) (run (sub-rgt p)))]))
```



Running “CS450 Lang + Strings” Programs

```
;; run: AST -> Result  
;; computes the result of given program AST
```

```
;; A Result is a:  
;; - Number  
;; - String
```

What should happen when two strings are added???

e.g., What is the “meaning” of (+ “hello” “world!”)

```
(define (run p)  
  (cond  
    [(num? n) n]  
    [(str? s) s]  
    [(add? p) (???? (run (add-lft p)) (run (add-rgt p)))]  
    [(sub? p) (???? (run (sub-lft p)) (run (sub-rgt p)))]))
```

Running “CS450 Lang + Strings” Programs

```
;; run: AST -> Result
;; computes the result of given program AST
```

```
;; A Result is a:
;; - Number
;; - String
```

What should happen when two strings are added???

e.g., What is the “meaning” of (+ “hello” “world!”)

```
(define (run p)
  (cond
    [(num? n) n]
    [(str? s) s]
    [(add? p) (450+ (run (add-lft p)) (run (add-rgt p)))]
    [(sub? p) (???? (run (sub-lft p)) (run (sub-rgt p)))])
```

Running: “CS450LANG” Programs: “450+”

```
;; 450+: Result Result -> Result  
;; “adds” two CS450Lang Result values together
```

```
;; A 450LangResult (Result) is either:  
;; - Number  
;; - String
```

TEMPLATE

```
(define (450+ x y)  
  (cond  
    [(number? x) ... ]  
    [(string? x) ... ])))
```

or

```
(define (450+ x y)  
  (cond  
    [(number? y) ... ]  
    [(string? y) ... ])))
```


Two-Argument Templates

- Sometimes ... a fn must **process two arguments simultaneously**
- **This template** should **combine templates of both args**
 - (This is only possible if the data defs are simple enough)

```
;; 450+: Result Result -> Result
;; “adds” two CS450Lang Result values together
```

(2-argument) TEMPLATE

```
(define (450+ x y)
  (cond
    [(and (number? x) (number? y)) ... ]
    [(and (number? x) (string? y)) ... ]
    [(and (string? x) (number? y)) ... ]
    [(and (string? x) (string? y)) ... ]
```

(see why this is typically not recommended?)

```
;; 450+: Result Result -> Result  
;; “adds” two CS450Lang Result values together
```

```
(define (450+ x y)  
  (cond  
    [(and (number? x) (number? y)) ... ]  
    [(and (number? x) (string? y)) ... ]  
    [(and (string? x) (number? y)) ... ]  
    [(and (string? x) (string? y)) ... ]
```

```
;; 450+: Result Result -> Result  
;; “adds” two CS450Lang Result values together
```

```
(define (450+ x y)  
  (cond  
    [(and (number? x) (number? y)) (+ x y)]  
    [(and (number? x) (string? y)) ... ]  
    [(and (string? x) (number? y)) ... ]  
    [(and (string? x) (string? y)) ... ]
```

Let's look at other languages!

```
:: 450+: Result Result -> Result  
;; “adds” two CS450Lang Result values together
```

```
(define (450+ x y)  
  (cond  
    [(and (number? x) (number? y)) (+ x y)]  
    [(and (number? x) (string? y)) ... ]  
    [(and (string? x) (number? y)) ... ]  
    [(and (string? x) (string? y)) ???])
```

JavaScript Semantics Exploration: “plus”

- `repljs.com`

```
;; 450+: Result Result -> Result
;; “adds” two CS450Lang Result values together
;; (following js semantics!)
```

```
(define (450+ x y)
  (cond
    [(and (number? x) (number? y)) (+ x y)]
    [(and (number? x) (string? y)) ... ]
    [(and (string? x) (number? y)) ... ]
    [(and (string? x) (string? y)) ???])
```

```
;; 450+: Result Result -> Result
;; “adds” two CS450Lang Result values together
;; (following js semantics!)
```

```
(define (450+ x y)
  (cond
    [(and (number? x) (number? y)) (+ x y)]
    [(and (number? x) (string? y)) ... ]
    [(and (string? x) (number? y)) ... ]
    [(and (string? x) (string? y)) (string-append x y)]
```

```
;; 450+: Result Result -> Result  
;; “adds” two CS450Lang Result values together  
;; (following js semantics)
```

```
(define (450+ x y)  
  (cond  
    [(and (number? x) (number? y)) (+ x y)]  
    [(and (number? x) (string? y)) ??? ]  
    [(and (string? x) (number? y)) ... ]  
    [(and (string? x) (string? y)) (string-append x y)]
```



```
;; 450+: Result Result -> Result
;; “adds” two CS450Lang Result values together
;; (following js semantics)
```

```
(define (450+ x y)
  (cond
    [(and (number? x) (number? y)) (+ x y)]
    [(and (number? x) (string? y)) (string-append (??? x) y)]
    [(and (string? x) (number? y)) ... ]
    [(and (string? x) (string? y)) (string-append x y)]
```

```
;; 450+: Result Result -> Result
;; “adds” two CS450Lang Result values together
;; (following js semantics)
```

```
(define (450+ x y)
  (cond
    [(and (number? x) (number? y)) (+ x y)]
    [(and (number? x) (string? y)) (string-append (num->str x) y)]
    [(and (string? x) (number? y)) ... ]
    [(and (string? x) (string? y)) (string-append x y)]
```

```
;; 450+: Result Result -> Result
;; “adds” two CS450Lang Result values together
;; (following js semantics)
```

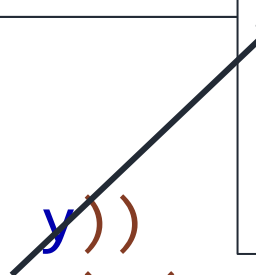
```
(define (450+ x y)
  (cond
    [(and (number? x) (number? y)) (+ x y)]
    [(and (number? x) (string? y)) (string-append (num->str x) y)]
    [(and (string? x) (number? y)) (string-append x (num->str y))]
    [(and (string? x) (string? y)) (string-append x y)]
```

(can any cond clauses be combined?)

```
;; 450+: Result Result -> Result
;; “adds” two CS450Lang Result values together
;; (following js semantics)
```

```
(define (450+ x y)
  (cond
    [(or (string? x) (string? y))
     (string-append (res->str x) (res->str x))]
    [else (+ x y)]))
```

```
;; res->str: Result -> String
(define (res->str x)
  (cond
    [(string? x) x]
    [(number? x) (number->string? x)]))
```



Running: “CS450 LANG” Programs: “minus”

```
;; 450+: Result Result -> Result  
;; “subtracts” 2nd cs450Lang Result from 1st one  
;; (following js semantics)
```

```
(define (450- x y)  
  (cond  
    [(and (number? x) (number? y)) (- x y)]  
    [(and (number? x) (string? y)) ... ]  
    [(and (string? x) (number? y)) ... ]  
    [(and (string? x) (string? y)) ... ])))
```

JavaScript Semantics Exploration: “minus”

“Not a Number”

≡ NaN

From Wikipedia, the free encyclopedia

In [computing](#), **NaN** ([*/næn/*](#)), standing for **Not a Number**, is a particular [value](#) of a numeric [data type](#) (often a [floating-point number](#)) which is undefined or unrepresentable, such as the result of [0/0](#). Systematic use of NaNs was introduced by the [IEEE 754](#) floating-point standard in 1985, along with the representation of other non-finite quantities such as [infinities](#).

 mdn web docs

NaN

The `NaN` global property is a value representing Not-A-Number.

Running: “CS450 LANG” Programs

```
;; run: AST -> Result
;; computes the result of running a CS450Lang program AST
```

```
;; An AST is one of:
;; - (num Number)
;; - (str String)
;; - (add AST AST)
;; - (sub AST AST)
```



```
;; A Result is either:
;; - Number
;; - String
;; - NaN
(struct nan [])
(define NaN (nan)) ; “singleton”
```

Don't forget to update all “Result” functions!

```
;; res->str: Result -> String
(define (res->str x)
  (cond
    [(string? x) x]
    [(number? x) (number->string? x)]
    [(NaN? x) “NaN”]))
```


Running: “CS450 LANG” Programs: “minus”

```
;; 450-: Result Result -> Result  
;; “subtracts” 2nd cs450Lang Result from 1st one  
;; (following js semantics)
```

```
(define (450- x y)  
  (cond  
    [(and (number? x) (number? y)) (- x y)]  
    [else NaN]))
```

```
;; A Result is either:  
;; - Number  
;; - String  
;; - NaN  
(struct nan [])  
(define NaN (nan))
```

In-class Coding 11/6 (hw9): put it all together!

```
;; parse: Expr -> AST
;; Parses "CS450 Lang" Expr to AST
```

```
;; run: AST -> Result
;; Computes result of running CS450Lang AST
```

```
;; An Expr is one of:
;; - Number
;; - String
;; - (list '+ Expr Expr)
;; - (list '- Expr Expr)
```

```
;; A Result is one of:
;; - Number
;; - String
;; - NaN
```

```
;; An AST is one of:
;; - (num Number)
;; - (str String)
;; - (add AST AST)
;; - (sub AST AST)
(struct num [val])
(struct str [val])
(struct add [lft rgt])
(struct sub [lft rgt])
```

BONUS: Program in "CS450 LANG"!

- Change require in cs450lang.rkt to point to your hw9.rkt file
- Write "CS450 LANG" code by putting this at top of any file: #lang s-exp "cs450lang.rkt"
 - See cs450lang-prog.rkt as an example