UMass Boston Computer Science

CS450 High Level Languages (section 2)

ASTs and Interpreters

Wednesday, November 8, 2023

Logistics

- HW 6 out
    - <u>due: </u>Sun 11/13 11:59 pm EST

Last Time

Design Recipe For Accumulator Functions

When a function needs to **"remember" extra information:**

1. *Specify* **accumulator:**
    - Name
    - Signature
    - Invariant
2. *Define* internal "helper" fn with extra **accumulator** arg
    - Helper fn does not need to repeat description, statement, or examples, (if they are the same) …
3. *Call* "helper" fn , with initial accumulator value, from original fn

Design Recipe For Accumulators

Function needs to “remember” extra information ...

... range of allowed values for node-data

```
;; valid-bst? : Tree<X> -> Bool
;; Returns true if t is a BST
(define (valid-bst? t)
```

1. *Specify* **accumulator:** name, signature, invariant

```
  ;; accumulator p? : (X -> Bool)
  ;; invariant: if t = (node l data r), p? remembers valid vals
  ;; for node-data such that (p? (node-data t)) is always true
```

2. *Define* internal “helper” fn with **accumulator** arg

```
  (define (valid-bst/p? p? t)
    (or (empty? t)
        (and (p? (node-data t))
             (valid-bst/p? (conjoin p? (curry > (node-data t)))
                           (node-left t))
             (valid-bst/p? (conjoin p? (curry <= (node-data t)))
                           (node-right t)))))

  (valid-bst/p? (lambda (x) true) t))
```

3.*Call* “helper” fn, with initial **accumulator**

Last Time

In-class Coding: Reverse, with accumulator

```
;; rev : List<X> -> List<X>
;; Returns the given list with elements in reverse order
(define (rev lst0)

  ;; accumulator rev-lst-so-far: List<X>
  ;; invariant: reversed elements of “list so far”,
  ;;            i.e., lst0 “minus” remaining-lst

  (define (rev/a rev-lst-so-far remaining-lst)
   (cond
    [(empty? remaining-lst) rev-lst-so-far]
    [else (rev/a (cons (first remaining-lst) rev-lst-so-far)
                 (rest remaining-lst))]))

  (rev/a empty lst0))
```

1. *Specify* **accumulator:** name, signature, invariant

refine accumulator description with param names

2. *Define* internal “helper” fn with **accumulator**

Add current item to front of reversed list

3. *Call* “helper” fn, with initial **accumulator**

In-class Coding: Tree Max

Accumulator used for "remembering" info, but doesn't always "accumulate"

```
;; tree-max : TreeNode<Int> -> Int
;; Returns the maximum value in a given (non-empty) (non-BST) tree
(define (tree-max t0)
```

1. *Specify* **accumulator:** name, signature, invariant

```
  ;; tree-max/a : Tree<Int> -> Int
  ;; accumulator root-val: Int
  ;; invariant: node-data of t0 root node (max of empty tree)
```

(need a "default" max for empty tree)

```
  (define (tree-max/a t root-val)
   (cond
     [(empty? t) root-val]
     [else (max (node-data t)
                (tree-max/a (node-left t) root-val)
                (tree-max/a (node-right t) root-val))]))
```

2. *Define* "helper" fn with **accumulator** (and other args)

This is not the only possible accumulator choice

```
  (tree-max/a t0 (node-data t0)))
```

3. *Call* "helper" fn, with initial **accumulator**

In-class Coding: Tree Max #2

```
;; tree-max : TreeNode<Int> -> Int
;; Returns the maximum value in a given (non-empty) (non-BST) tree
(define (tree-max t0)

  ;; tree-max/a : Tree<Int> -> Int
  ;; accumulator root-val: Int
  ;; invariant: node-data of ~~root~~ parent node (max of empty tree)
  (define (tree-max/a t ~~root-val~~ parent-val)
   (cond
     [(empty? t) ~~root-val~~ parent-val]
     [else (max ~~(node-data t)~~ parent-val
                (tree-max/a (node-left t) ~~root-val~~ (node-data t))
                (tree-max/a (node-right t) ~~root-val~~ (node-data t)))]))


  (tree-max/a t0 (node-data t0)))
```

(need a “default” max for empty tree)

Pass node-data of parent on recursive call

The **accumulator invariant** is key to understanding the program!

Previously

Intertwined Data Definitions

- Come up with a Data Definition for …
- … valid Racket Programs

Basic Valid Racket Programs

- 1
- “one”
- `(+ 1 2)`

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

```
;; - ???
```

Valid Racket Programs

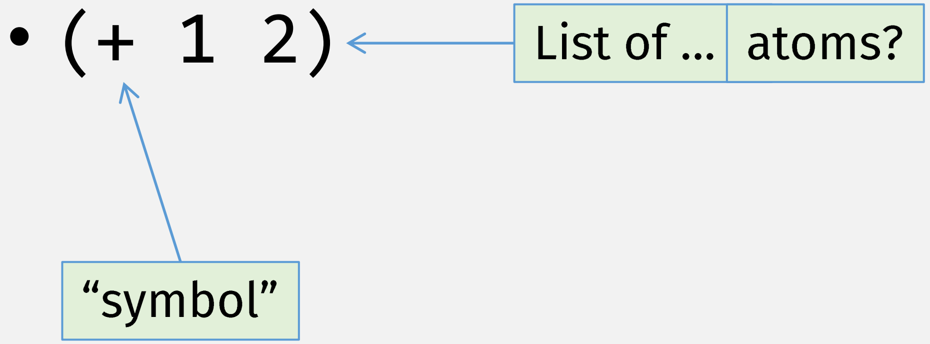
- 1
- “one”
- (+ 1 2)

```
;; A RacketProg is a:
;; - Atom
```

```
;; - ???
```

```
;; An Atom is a:
;; - Number
;; - String
```

Valid Racket Programs



- `(+ 1 2)`

List of … atoms?

“symbol”

```
;; A RacketProg is a:
;; - Atom
;; - List<Atom> ???
```

```
;; An Atom is a:
;; - Number
;; - String
;; - Symbol
```

Valid Racket Programs

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

-
  ```
  (* (+ 1 2)
     (- 4 3)
     (/ 10 5))
  ```

Tree?

Each tree "node" is a list, of ... RacketProgs ??

But: how many values does each node have?? Unknown!

```
;; A RacketProg is a:
;; - Atom
;; - Tree<???>
```

```
;; An Atom is a:
;; - Number
;; - String
;; - Symbol
```

Valid Racket Programs

- `(* (+ 1 2)`
  `   (- 4 3))`
- `(* (+ 1 2)`
  `   (- 4 3)`
  `   (/ 10 5))`

Tree?

Each tree “node” is a list, of … RacketProgs ??

But: how many values does each node have??

```
;; A RacketProg is a:
;; - Atom
;; - ProgTree
```

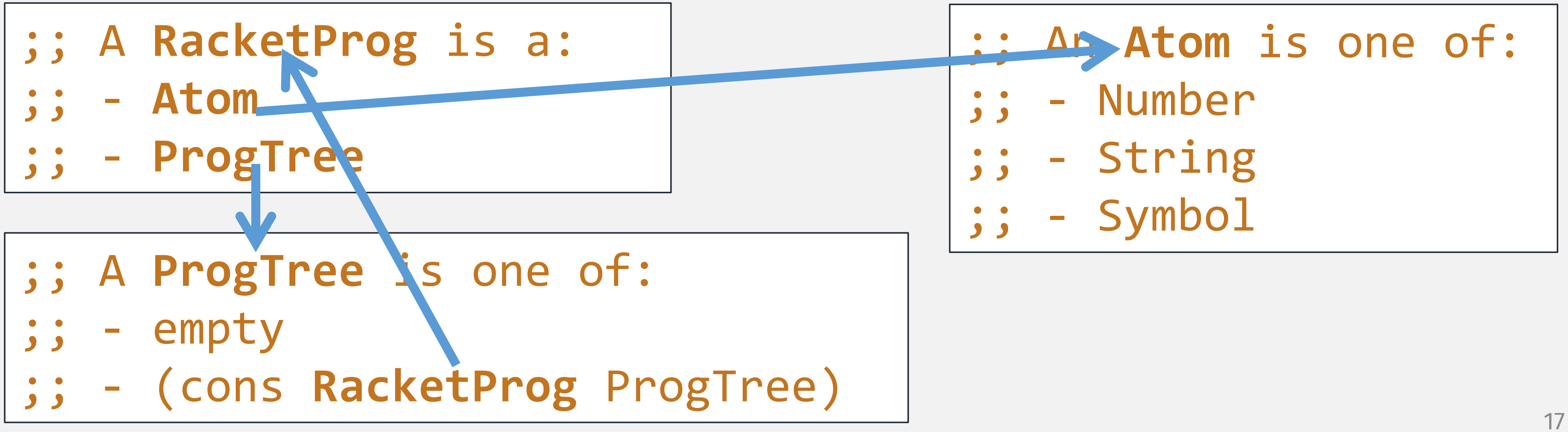
```
;; An Atom is a:
;; - Number
;; - String
;; - Symbol
```

```
;; A ProgTree is one of:
;; - empty
;; - (cons RacketProg ProgTree)
```

Recursive Data Def!

Valid Racket Programs

Also, **Intertwined** Data Defs!



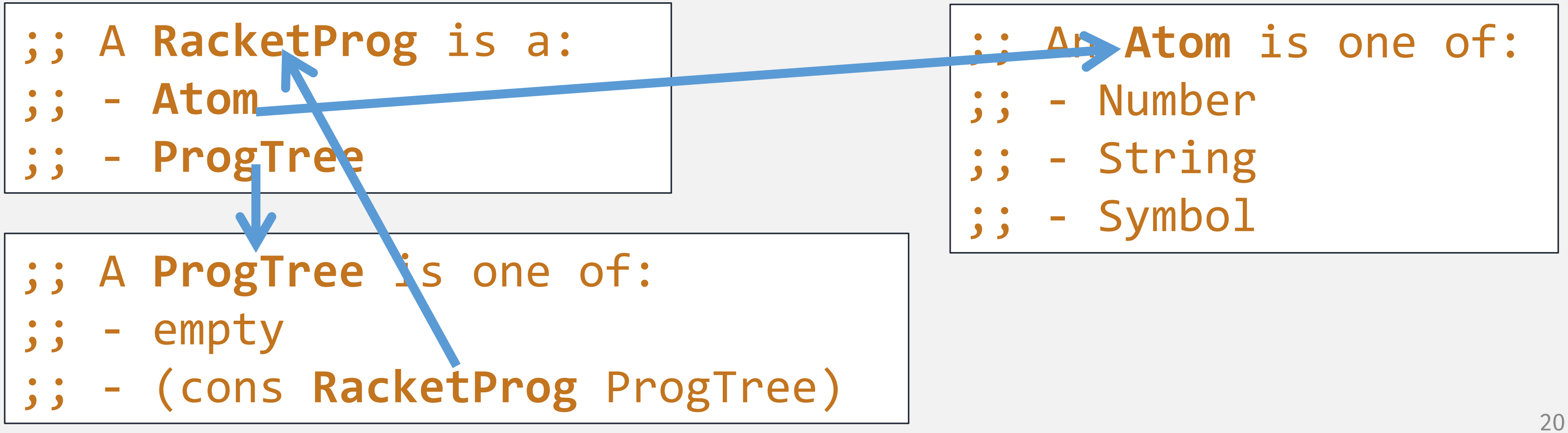
```
;; A RacketProg is a:
;; - Atom
;; - ProgTree
```

```
;; A ProgTree is one of:
;; - empty
;; - (cons RacketProg ProgTree)
```

```
;; An Atom is one of:
;; - Number
;; - String
;; - Symbol
```

Intertwined Data

- A <u>set</u> of Data Definitions that reference each other
- <u>Templates</u> should be defined together …



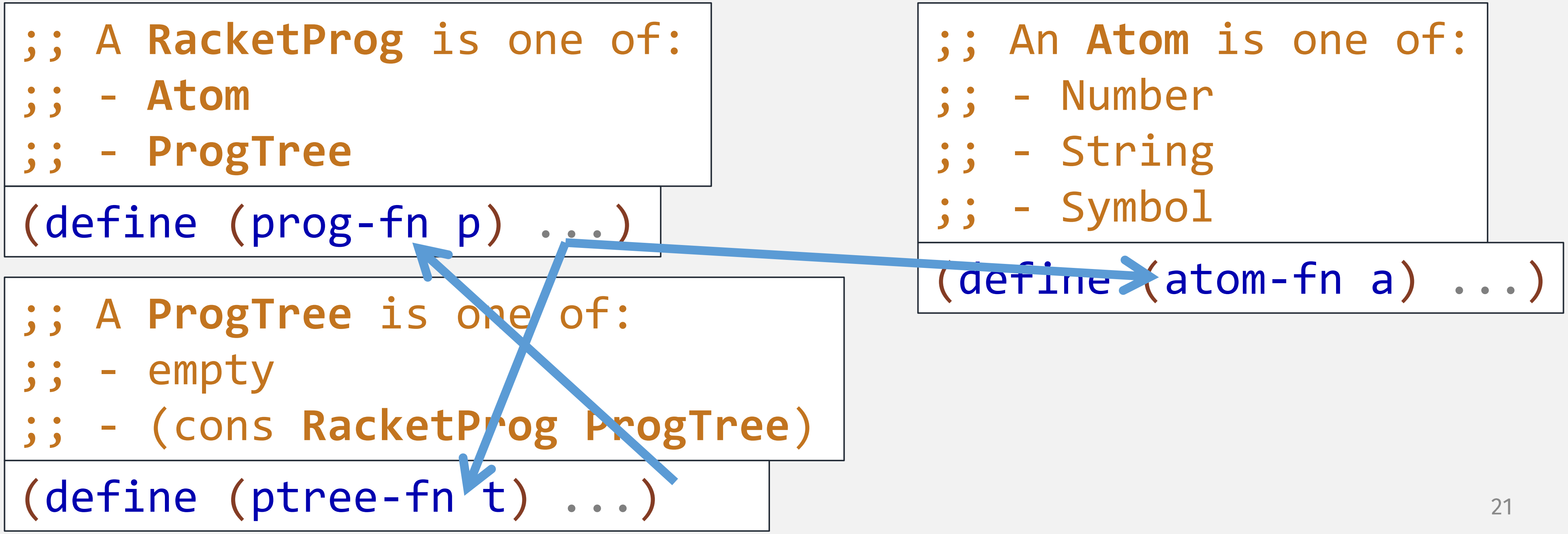
```
;; A RacketProg is a:
;; - Atom
;; - ProgTree
```

```
;; A ProgTree is one of:
;; - empty
;; - (cons RacketProg ProgTree)
```

```
;; An Atom is one of:
;; - Number
;; - String
;; - Symbol
```

Intertwined Data

- A set of Data Definitions that reference each other
- Templates should be defined together …
  - … and should reference each other’s templates (when needed)



```
;; A RacketProg is one of:
;; - Atom
;; - ProgTree
```

```
(define (prog-fn p) ...)
```

```
;; A ProgTree is one of:
;; - empty
;; - (cons RacketProg ProgTree)
```

```
(define (ptree-fn t) ...)
```

```
;; An Atom is one of:
;; - Number
;; - String
;; - Symbol
```

```
(define (atom-fn a) ...)
```

"Racket Prog" = S-expression!

An S-expression is a Racket program's **syntax** ...

What about its **semantics** (meaning)?

```
;; A RacketProg is one of:
;; - Atom
;; - ProgTree
```

```
;; An Atom is one of:
;; - Number
;; - String
;; - Symbol
```

```
;; A ProgTree is one of:
;; - empty
;; - (cons RacketProg ProgTree)
```

Syntax vs Semantics (Spoken Language)

Syntax

- Specifies: **valid language structures**
  - E.g., **sentence** = **noun** (subject) + **verb** + **noun** (object)
- “the ball threw the child”
  - Syntactically: valid!
  - Semantically: ???

Semantics

- Specifies: **meaning of language structures**

Syntax vs Semantics (Programming Language)

Syntax

- Specifies: valid language structures
  - E.g., ???

Semantics

- Specifies: meaning of language structures

Syntax vs Semantics (Programming Language)

Syntax

- Specifies: **valid language structures**
  - E.g., **valid Racket program = s-expressions**
  - E.g., **valid Python program = …**

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

A: The result from “running” it

Semantics

- Specifies: **meaning of language structures**

How does a program “**run**”?

Running Programs: `eval`

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

An “eval” function turns a “program” into a “result”

An “eval” function is more generally called an **interpreter**

(Programs are usually not directly interpreted)

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

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

A: The result from “running” it

How does a program “**run**”?

From
Lecture 1

NOTE: This hierarchy is *approximate*

“high” level
(easier for humans
to understand)

“declarative”

perative”

el
pu)

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 intrepreted)

“high” level
(easier for humans
to understand)

surface language

compiler

“decl ative”

target language

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

nperative”

el

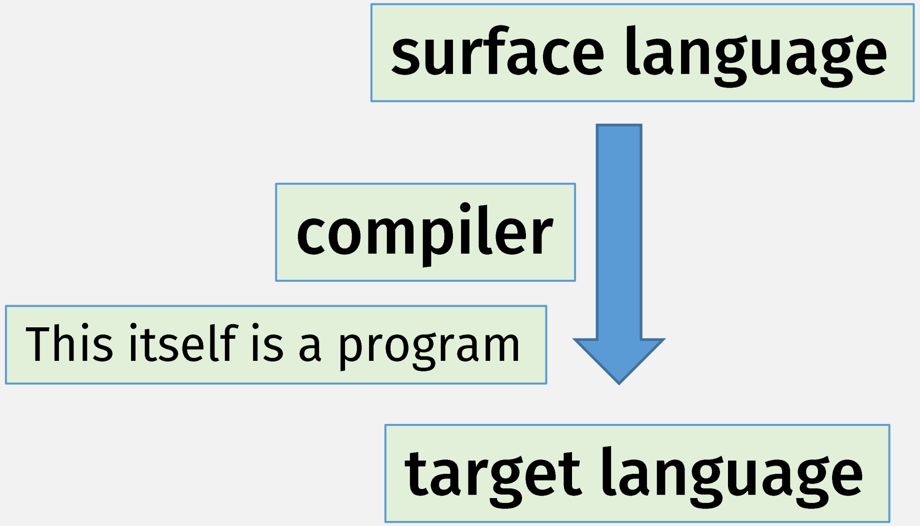
pu)

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

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!**



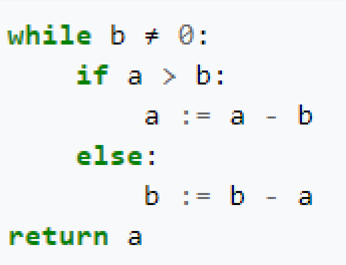
surface language

compiler

This itself is a program

target language

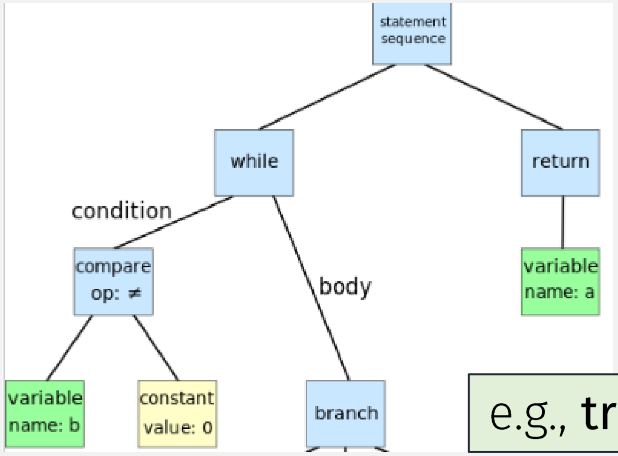
input



```
while b ≠ 0:
    if a > b:
        a := a - b
    else:
        b := b - a
return a
```

e.g., **string of chars** (less structure)

output

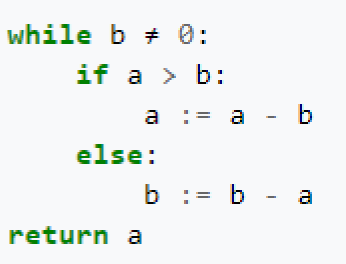


e.g., **tree** (more structure)

Semantics

- Specifies: **meaning of language <u>structures</u>**
- So: **to "run" a program, we need to see the structure first**

input



```
while b ≠ 0:
    if a > b:
        a := a - b
    else:
        b := b - a
return a
```

e.g., **string of chars** (less structure)

surface language

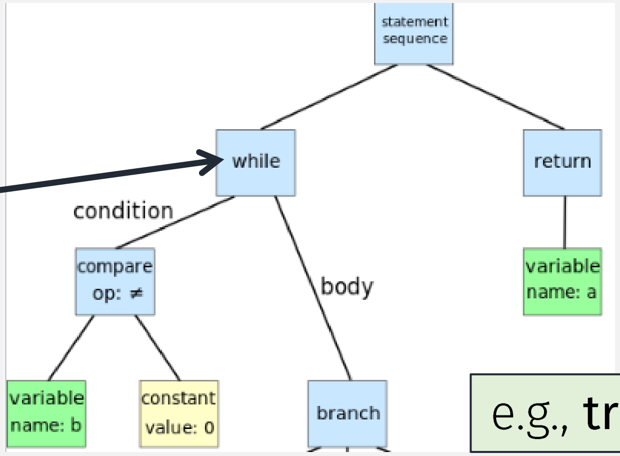
Compiler, step 1

= **parser**

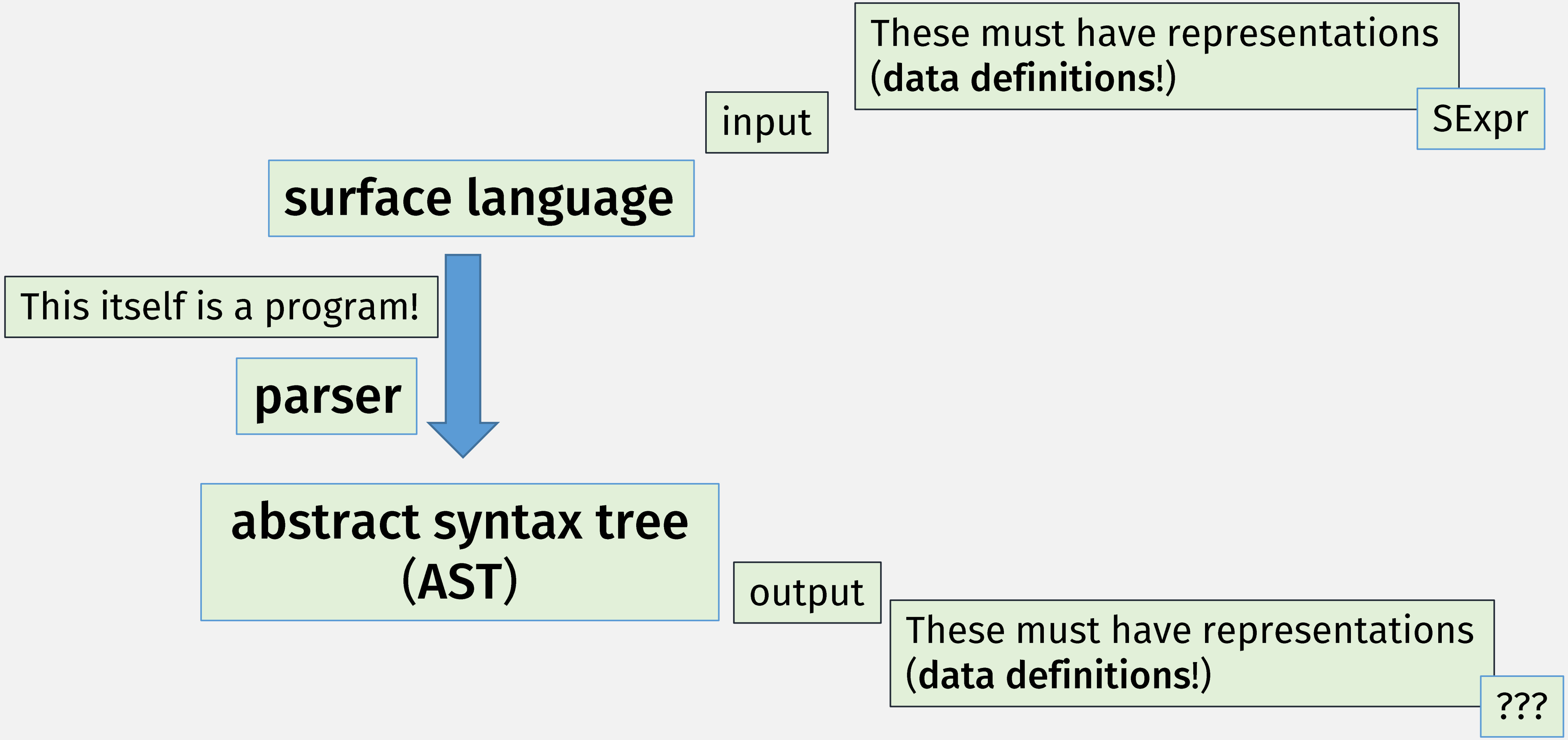
(a **compiler** actually has <u>many steps</u> ... take a compilers course!)

abstract syntax tree (AST)

output



e.g., **tree** (more structure)



input

These must have representations
(**data definitions!**)

SExpr

surface language

This itself is a program!

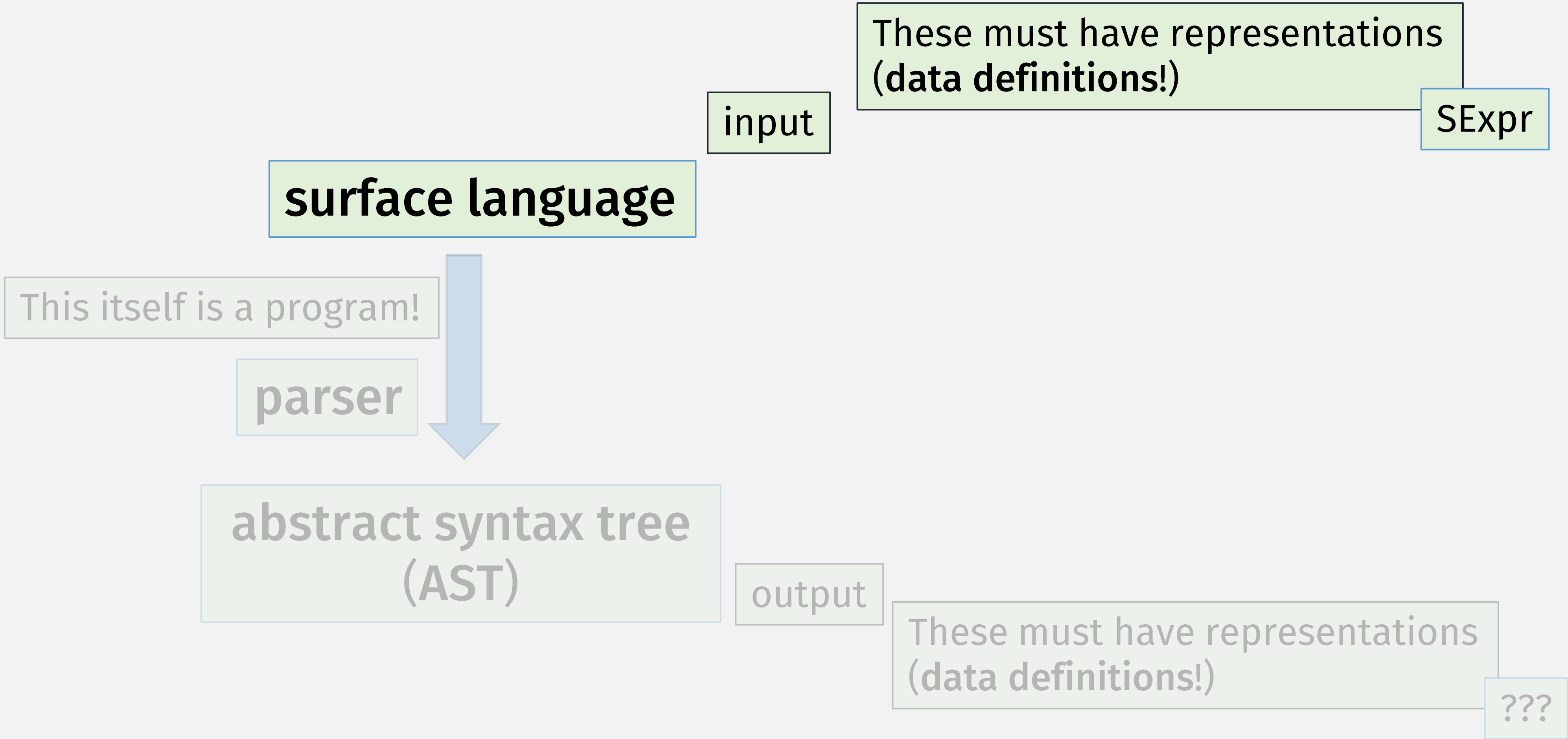
parser

abstract syntax tree
(AST)

output

These must have representations
(**data definitions!**)

???



input

These must have representations
(**data definitions!**)

SExpr

surface language

This itself is a program!

parser

abstract syntax tree (AST)

output

These must have representations
(**data definitions!**)

???

These must have representations (**data definitions!**)

input

SExpr

surface language

```
;; A SimpleSexpr (Ssexpr) is one of:
;; - Number
;; - (list ‘+ Ssexpr Ssexpr)
;; - (list ‘- Ssexpr Ssexpr)
```

Data Definition Template

When a **Data Definition** is an **itemization** of **compound data** …

- **Template =**
  - cond to distinguish cases
  - Getters to extract pieces
  - recursive calls

```
;; A SimpleSexpr (Ssexpr) is one of:
;; - Number
;; - (list ‘+ Ssexpr Ssexpr)
;; - (list ‘- Ssexpr Ssexpr)
```

```
(define (ss-fn s)
  (cond
    [(number? s) … ]
    [(and (list? s) (equal? ‘+ (first s)))
     … (ss-fn (second s)) … (ss-fn (third s)) … ]
    [(and (list? s) (equal? ‘- (first s)))
     … (ss-fn (second s)) … (ss-fn (third s)) … ]))
```

Cond guards must distinguish the different cases

Cond clause has getters and recursive calls

Interlude: pattern **match**ing (again)

When a **Data Definition** is an **itemization** of **compound data** …

- **Template** =
  - ~~cond to distinguish cases~~
  - `match` = cond + getters
  - recursive calls

```
;; A SimpleSexpr (Ssexpr) is one of:
;; - Number
;; - (list ‘+ Ssexpr Ssexpr)
;; - (list ‘- Ssexpr Ssexpr)
```

```
(define (ss-fn s)
  (match s
    [(? number?) … ]
    [`(+ ,x ,y)
      … (ss-fn x) … (ss-fn y) … ]
    [`(- ,x ,y)
      … (ss-fn x) … (ss-fn y) … ]))
```

Predicate pattern

“Quasiquote” pattern

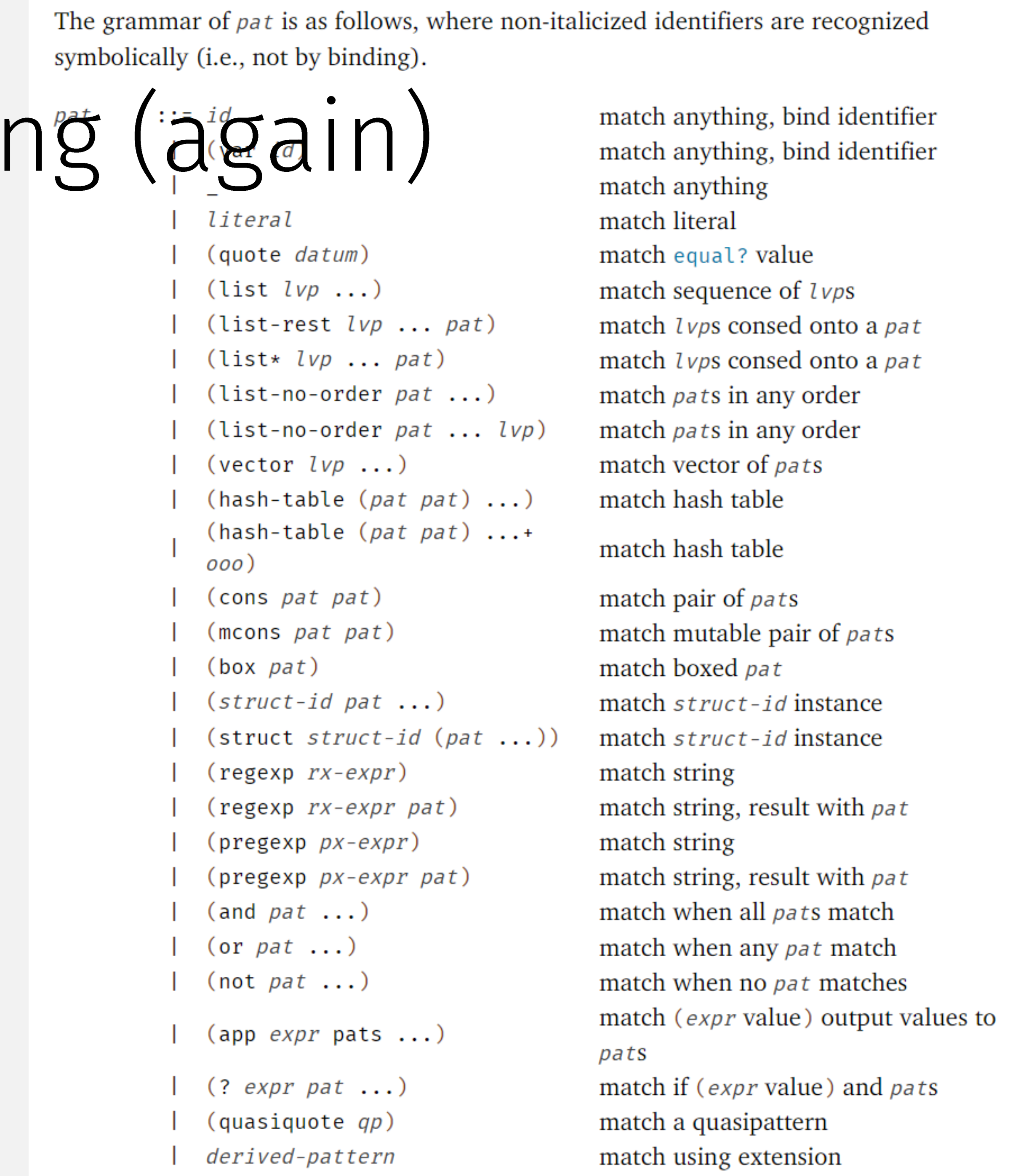
Symbols match exactly

Match patterns

“Unquote” defines new variable name (for value at that position)

Interlude: pattern **`matching`** (again)

- See Racket docs for the full pattern language



The grammar of *pat* is as follows, where non-italicized identifiers are recognized symbolically (i.e., not by binding).

pat ::=	*id*	match anything, bind identifier
|	(var *id*)	match anything, bind identifier
|	_	match anything
|	*literal*	match literal
|	(quote *datum*)	match equal? value
|	(list *lvp* ...)	match sequence of *lvps*
|	(list-rest *lvp* ... *pat*)	match *lvps* consed onto a *pat*
|	(list* *lvp* ... *pat*)	match *lvps* consed onto a *pat*
|	(list-no-order *pat* ...)	match *pats* in any order
|	(list-no-order *pat* ... *lvp*)	match *pats* in any order
|	(vector *lvp* ...)	match vector of *pats*
|	(hash-table (*pat* *pat*) ...)	match hash table
|	(hash-table (*pat* *pat*) ...+ *ooo*)	match hash table
|	(cons *pat* *pat*)	match pair of *pats*
|	(mcons *pat* *pat*)	match mutable pair of *pats*
|	(box *pat*)	match boxed *pat*
|	(*struct-id* *pat* ...)	match *struct-id* instance
|	(struct *struct-id* (*pat* ...))	match *struct-id* instance
|	(regexp *rx-expr*)	match string
|	(regexp *rx-expr* *pat*)	match string, result with *pat*
|	(pregexp *px-expr*)	match string
|	(pregexp *px-expr* *pat*)	match string, result with *pat*
|	(and *pat* ...)	match when all *pats* match
|	(or *pat* ...)	match when any *pat* match
|	(not *pat* ...)	match when no *pat* matches
|	(app *expr* pats ...)	match (*expr* value) output values to *pats*
|	(? *expr* *pat* ...)	match if (*expr* value) and *pats*
|	(quasiquote *qp*)	match a quasipattern
|	*derived-pattern*	match using extension

Interlude: pattern `matching` (again)

When a **Data Definition** is an **itemization** of **compound data** …

- **Template** =
  - ~~cond to distinguish cases~~
  - `match` = cond + getters
  - recursive calls

`match` can be more concise and readable

```
(define (ss-fn s)
  (match s
    [(? number?) … ]
    [`(+ ,x ,y)
      … (ss-fn x) … (ss-fn y) … ]
    [`(- ,x ,y)
      … (ss-fn x) … (ss-fn y) … ]))
```

```
(define (ss-fn s)
  (cond
    [(number? s) … ]
    [(and (list? s) (equal? ‘+ (first s)))
     … (ss-fn (second s)) …
     … (ss-fn (third s)) … ]
    [(and (list? s) (equal? ‘- (first s)))
     … (ss-fn (second s)) …
     … (ss-fn (third s)) … ]))
```

input

These must have representations
(**data definitions!**)

SExpr

surface language

This itself is a program!

parser

abstract syntax tree (AST)

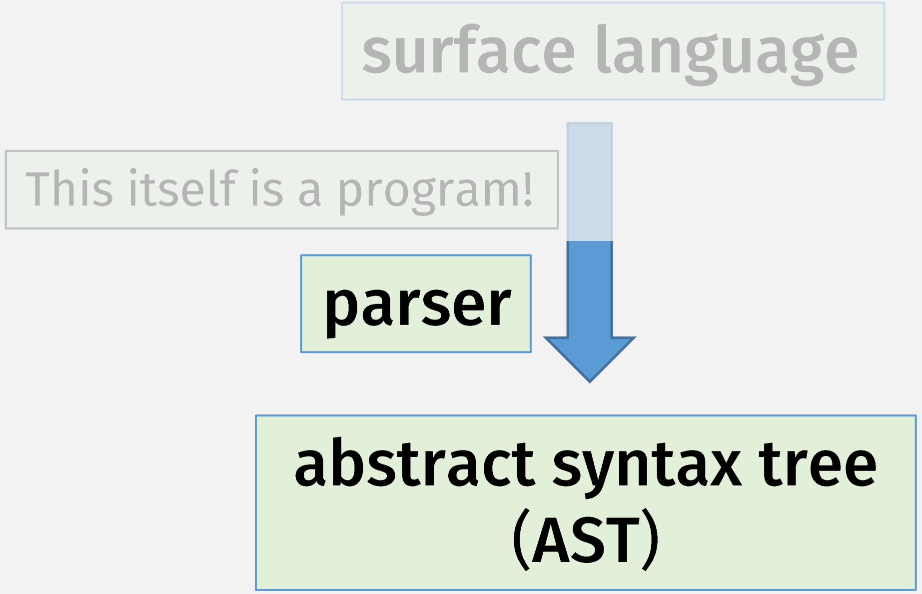
```
;; A SimpleSexpr (Ssexpr) is one of:
;; - Number
;; - (list ‘+ Ssexpr Ssexpr)
;; - (list ‘- Ssexpr Ssexpr)
```

output

These must have representations
(**data definitions!**)

???

These must have representations



surface language

This itself is a program!

parser

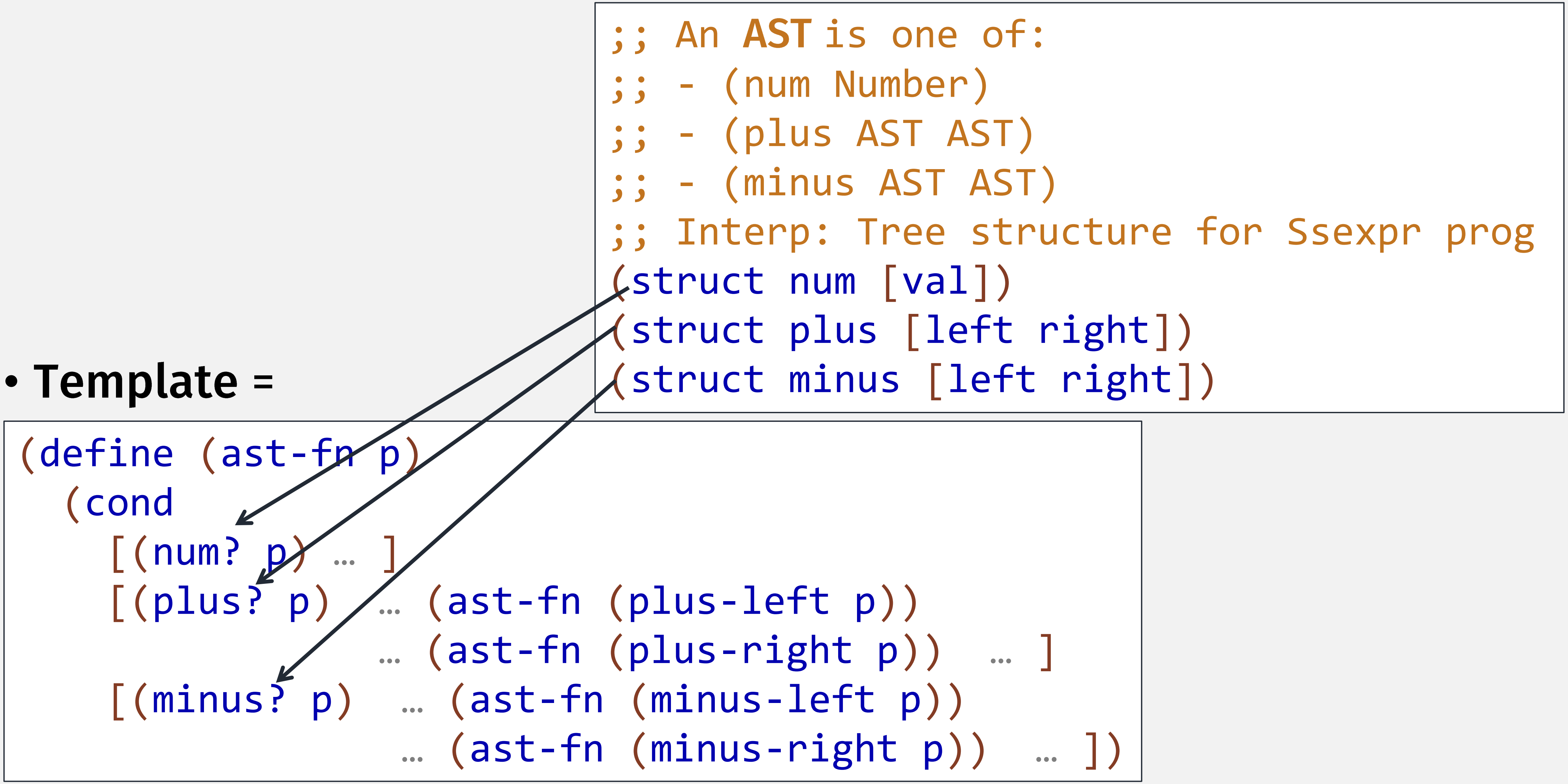
abstract syntax tree (AST)

```
;; An AST is one of:
;; - (num Number)
;; - (plus AST AST)
;; - (minus AST AST)
;; Interp: Tree structure for Ssexpr prog
(struct num [val])
(struct plus [left right])
(struct minus [left right])
```

output

These must have representations
(**data definitions!**)

???



```
;; An AST is one of:
;; - (num Number)
;; - (plus AST AST)
;; - (minus AST AST)
;; Interp: Tree structure for Ssexpr prog
(struct num [val])
(struct plus [left right])
(struct minus [left right])
```

- **Template** =

```
(define (ast-fn p)
  (cond
    [(num? p) … ]
    [(plus? p)  … (ast-fn (plus-left p))
                … (ast-fn (plus-right p))  … ]
    [(minus? p)  … (ast-fn (minus-left p))
                 … (ast-fn (minus-right p))  … ])
```

```
;; An AST is one of:
;; - (num Number)
;; - (plus AST AST)
;; - (minus AST AST)
;; Interp: Tree structure for Ssexpr prog
(struct num [val])
(struct plus [left right])
(struct minus [left right])
```

- **Template** (with `match`) =

```
(define (ast-fn p)
  (~~cond~~ match p
    [(num n) … ]
    [(plus x y) … (ast-fn x) …
                … (ast-fn y) … ]
    [(minus x y) … (ast-fn x) …
                   (ast-fn y) … ])
```

Struct name

Struct patterns

Extracts and names fields

- Repo: **cs450f23/lecture18-inclass**
- File: **parse**-*<your last name>*.rkt

In-class Coding 11/8 #1: parser

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

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

```
;; An AST is one of:
;; - (num Number)
;; - (plus AST AST)
;; - (minus AST AST)
;; Interp: Tree structure for Ssexpr
(struct num [val])
(struct plus [left right])
(struct minus [left right])
```

- Repo: **cs450f23/lecture18-inclass**
- File: **eval-***<your last name>***.rkt**

In-class Coding 11/8 #2: eval

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

```
;; An AST is one of:
;; - (num Number)
;; - (plus AST AST)
;; - (minus AST AST)
;; Interp: Tree structure for Ssexpr
(struct num [val])
(struct plus [left right])
(struct minus [left right])
```

No More Quizzes!

but push your in-class work to:
Repo: **`cs450f23/lecture18-inclass`**