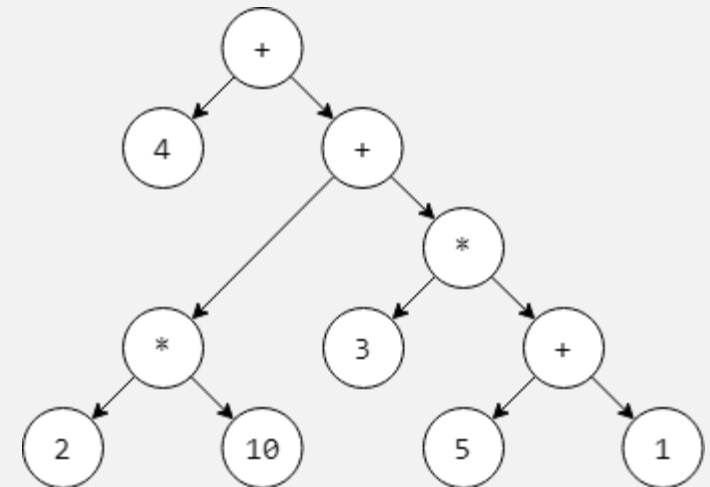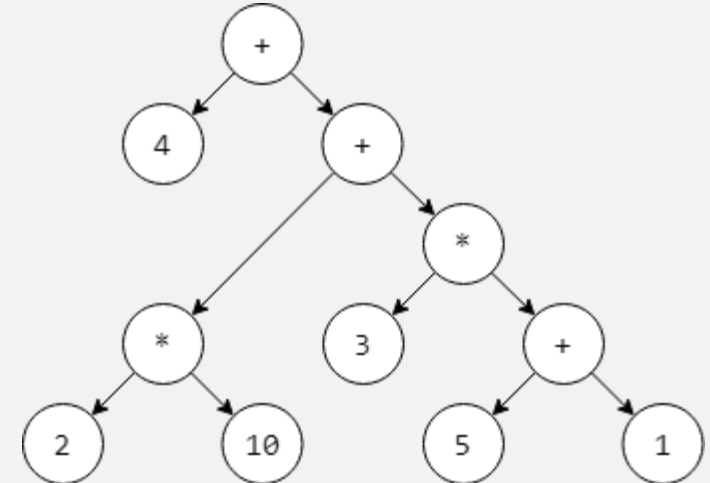UMass Boston Computer Science
CS450 High Level Languages (section 2)
ASTs and Interpreters
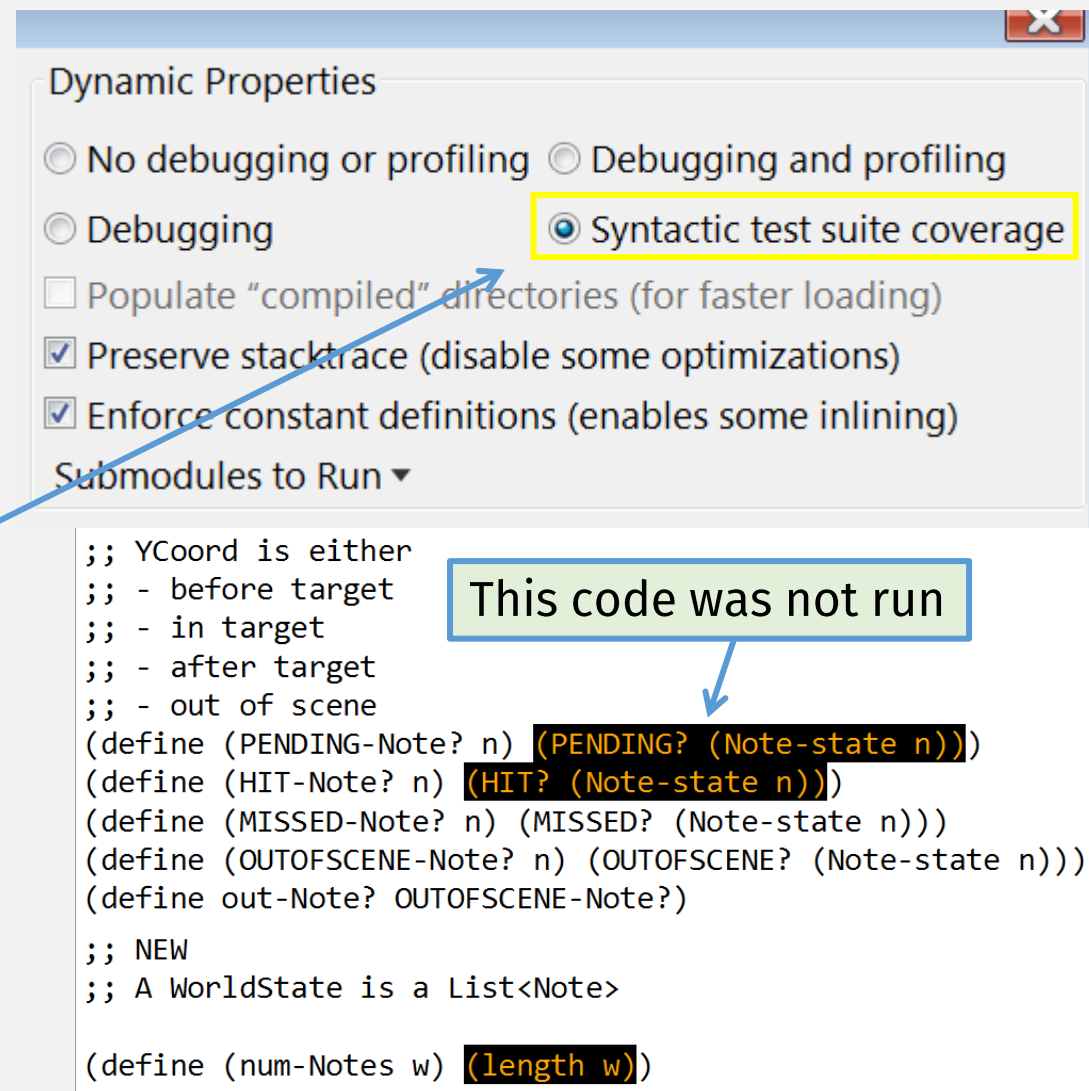
Monday, November 4, 2024

Logistics

- HW 9 "out"
  - See: 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

HW Minimum Submission Requirements

- "`main`" runs without errors

- Tests run without errors

- 100% (Test / Example) "Coverage"
  - In "Choose Language" Menu
  - NOTE: only works with single files

Dynamic Properties

- ◯ No debugging or profiling   ◯ Debugging and profiling
- ◯ Debugging   ◉ Syntactic test suite coverage
- ☐ Populate "compiled" directories (for faster loading)
- ☑ Preserve stacktrace (disable some optimizations)
- ☑ Enforce constant definitions (enables some inlining)

Submodules to Run ▾

```
;; YCoord is either
;; - before target
;; - in target
;; - after target
;; - out of scene
(define (PENDING-Note? n) (PENDING? (Note-state n)))
(define (HIT-Note? n) (HIT? (Note-state n)))
(define (MISSED-Note? n) (MISSED? (Note-state n)))
(define (OUTOFSCENE-Note? n) (OUTOFSCENE? (Note-state n)))
(define out-Note? OUTOFSCENE-Note?)

;; NEW
;; A WorldState is a List<Note>

(define (num-Notes w) (length w))
```

This code was not run

HW Minimum Submission Requirements

- "`main`" runs without errors

- Tests run without errors

Code should <u>never</u> get into a state where **this is true!**

Incremental programming!

Function Design Recipe

1. **Name**

2. **Signature** – <u>types</u> of the **function input(s)** and **output**

3. **Description** – <u>explain</u> (in **English prose**) the **function behavior**

4. **Examples** – <u>show</u> (using `rackunit`) the **function behavior**

5. **Template** – <u>sketch out</u> the **function structure** (using input's **Data Definition**)

6. **Code** – <u>implement</u> the **rest of the function** (arithmetic)

7. **Tests** – <u>check</u> (using `rackunit`) the **function behavior**

Incremental Programming

1. **Name**

2. **Signature** – <u>types</u> of the **function input(s)** and **output**

3. **Description** – <u>explain</u> (in **English prose**) the **function behavior**

4. **Examples** – <u>show</u> (using `rackunit`) the **function behavior**

   > Code should <u>never</u> be crashing!

5. **Template** – <u>sketch out</u> the **function structure** (using input's **Data Definition**)

6. **Code** – <u>implement</u> the **rest of the function** (arithmetic)

   > Start: by filling in with "placeholders"

7. **Tests** – <u>check</u> (using `rackunit`) the **function behavior**

Incremental Programming

1. **Name**

2. **Signature** – <u>types</u> of the **function input(s)** and **output**

3. **Description** – <u>explain</u> (in **English prose**) the **function behavior**

4. **Examples** – <u>show</u> (using `rackunit`) the **function behavior**

5. **Template** – <u>sketch out</u> the **function structure** (using input's **Data Definition**)

6. **Code** – <u>implement</u> the **rest of the function** (arithmetic)

7. **Tests** – <u>check</u> (using `rackunit`) the **function behavior**

> Code should <u>never</u> be crashing!

> This way: always know there the "bug" is

> Then: make <u>small</u> code changes and test <u>immediately</u>

> Tests (and example tests) should <u>always</u> be passing!

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 one of:
;; - Number
;; - String
```

Valid Racket Programs

- (+ 1 2) ← | List of … | atoms? |

"symbol" → (points to `+`)

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

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

Valid Racket Programs

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

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

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

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

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

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

Valid Racket Programs

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

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

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 one of:
;; - 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
```

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

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

Intertwined Data

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

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

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

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

Intertwined Data

- A <u>set</u> of Data Definitions that <u>reference each other</u>
- <u>Templates</u> should **be** <u>defined together</u> …
  - … and should <u>reference each other's templates</u> (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) ...)
```

???

18

Intertwined Templates

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

(define (prog-fn s)
  (cond
    [(atom? s) ... (atom-fn s) ...]
    [else ... (ptree-fn s) ...]))
```

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

(define (atom-fn a)
  (cond
    [(number? a) ... ]
    [(string? a) ... ]
    [else ... ]))
```

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

(define (ptree-fn t)
  (cond
    [(empty? t) ...]
    [else ... (prog-fn (first t)) ... (ptree-fn (rest t)) ...]))
```

Intertwined data have intertwined templates!

A "Racket Prog" = S-expression!

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

```
(define (sexpr-fn s)
  (cond
    [(atom? s) ... (atom-fn s) ...]
    [else  ... (ptree-fn s) ...]))
```

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

```
(define (ptree-fn t)
  (cond
    [(empty? t) ...]
    [else  ... (sexpr-fn (first t)) ... (ptree-fn (rest t)) ...]))
```

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

```
(define (atom-fn a)
  (cond
    [(number? a) ... ]
    [(string? a) ... ]
    [else ... ]))
```

Counting Symbols

```
;; count : Symbol Sexpr -> Nat
;; Computes the number of times the given
;; symbol appears in the given s-expression
(define (count sym se)
  (cond
    [(atom? s) ... (atom-fn s) ...]
    [else  ... (ptree-fn s) ...]))
```

```
;; count-atom : Symbol Atom -> Nat
(define (count-atom sym se)
  (cond
    [(number? a) ... ]
    [(string? a) ... ]
    [else ... ]))
```

```
;; count-ptree : Symbol ProgTree -> Nat
(define (count-ptree sym se)
  (cond
    [(empty? t) ...]
    [else  ... (sexpr-fn (first t)) ... (ptree-fn (rest t)) ...]))
```

Counting Symbols

```
;; count : Symbol Sexpr -> Nat
;; Computes the number of times the given
;; symbol appears in the given s-expression

(define (count sym se)
  (cond
    [(atom? s) (count-atom sym se)]
    [else (count-ptree sym se)]))
```

```
;; count-atom : Symbol Atom -> Nat

(define (count-atom sym se)
  (cond
    [(number? a) ... ]
    [(string? a) ... ]
    [else ... ]))
```

```
;; count-ptree : Symbol ProgTree -> Nat

(define (count-ptree sym se)
  (cond
    [(empty? t) ...]
    [else  ... (sexpr-fn (first t)) ... (ptree-fn (rest t)) ...]))
```

Counting Symbols

```
;; count : Symbol Sexpr -> Nat
;; Computes the number of times the given
;; symbol appears in the given s-expression
```

```
(define (count sym se)
  (cond
    [(atom? s) (count-atom sym se)]
    [else (count-ptree sym se)]))
```

```
;; count-atom : Symbol Atom -> Nat
```

```
(define (count-atom sym se)
  (cond
    [(symbol? se)
     (if (symbol=? sym se) 1 0)]
    [else 0]))
```

```
;; count-ptree : Symbol ProgTree -> Nat
```

```
(define (count-ptree sym se)
  (cond
    [(empty? t) ...]
    [else  ... (sexpr-fn (first t)) ... (ptree-fn (rest t)) ...]))
```

Counting Symbols

```
;; count : Symbol Sexpr -> Nat
;; Computes the number of times the given
;; symbol appears in the given s-expression
```

```
(define (count sym se)
  (cond
    [(atom? s) (count-atom sym se)]
    [else (count-ptree sym se)]))
```

```
;; count-atom : Symbol Atom -> Nat
```

```
(define (count-atom sym se)
  (cond
    [(symbol? se)
      (if (symbol=? sym se) 1 0)]
    [else 0]))
```

```
;; count-ptree : Symbol ProgTree -> Nat
```

```
(define (count-ptree sym se)
  (cond
    [(empty? se) 0]
    [else  ... (sexpr-fn (first t)) ... (ptree-fn (rest t)) ...]))
```

Counting Symbols

```
;; count : Symbol Sexpr -> Nat
;; Computes the number of times the given
;; symbol appears in the given s-expression

(define (count sym se)
  (cond
    [(atom? s) (count-atom sym se)]
    [else (count-ptree sym se)]))
```

```
;; count-atom : Symbol Atom -> Nat

(define (count-atom sym se)
  (cond
    [(symbol? se)
     (if (symbol=? sym se) 1 0)]
    [else 0]))
```

```
;; count-ptree : Symbol ProgTree -> Nat

(define (count-ptree sym se)
  (cond
    [(empty? se) 0]
    [else (+ (count sym (first se))
             (count-ptree sym (rest se)))]))
```

Syntax vs Semantics (Spoken Language)

Syntax

- Specifies: <u>validity</u> of language structures
  - E.g., **sentence** = **noun** (subject) + **verb** + **noun** (object)
- "the ball threw the child"
  - Syntactically: valid!
  - Semantically: ???

Semantics

- Specifies: <u>meaning</u> of language structures

Syntax vs Semantics (<u>Programming</u> Language)

Syntax

- Specifies: <u>validity</u> of language structures
  - E.g.,   ???

Semantics

- Specifies: <u>meaning</u> of language structures

Syntax vs Semantics (<u>Programming</u> Language)

Syntax

- Specifies: <u>validity</u> of ~~language structures~~ programs!
    - E.g., valid Racket program = s-expressions
    - E.g., valid Python program = …

Semantics

- Specifies: <u>meaning</u> of ~~language structures~~ programs!

Q: What is the "meaning" of a program?

A: The result from "running" it

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**"?

"high" level
(easier for humans to understand)

NOTE: This hierarchy is *approximate*

"**declarative**"

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

mperative"

"low" level
(runs on cpu)

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

"high" level
(easier for humans
to understand)

surface language

compiler

"declarative"

target language

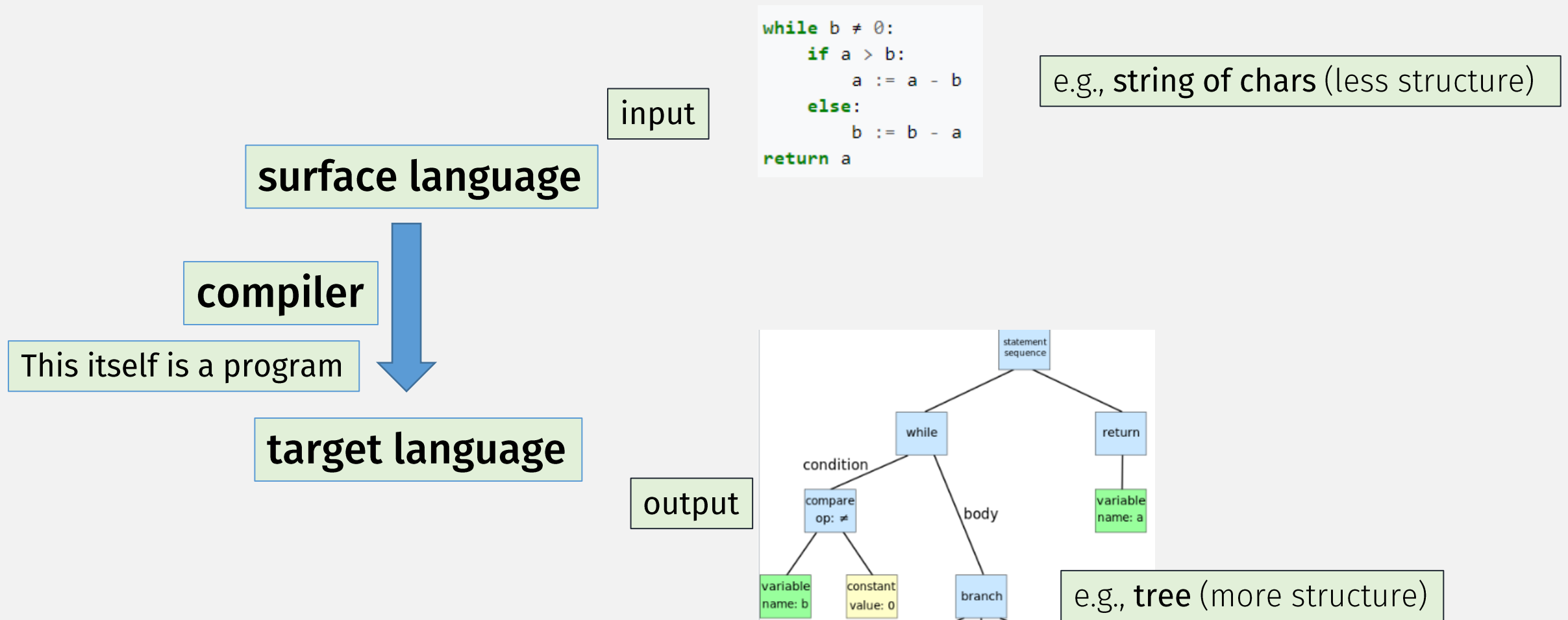"imperative"

"low" level
(runs on cpu)

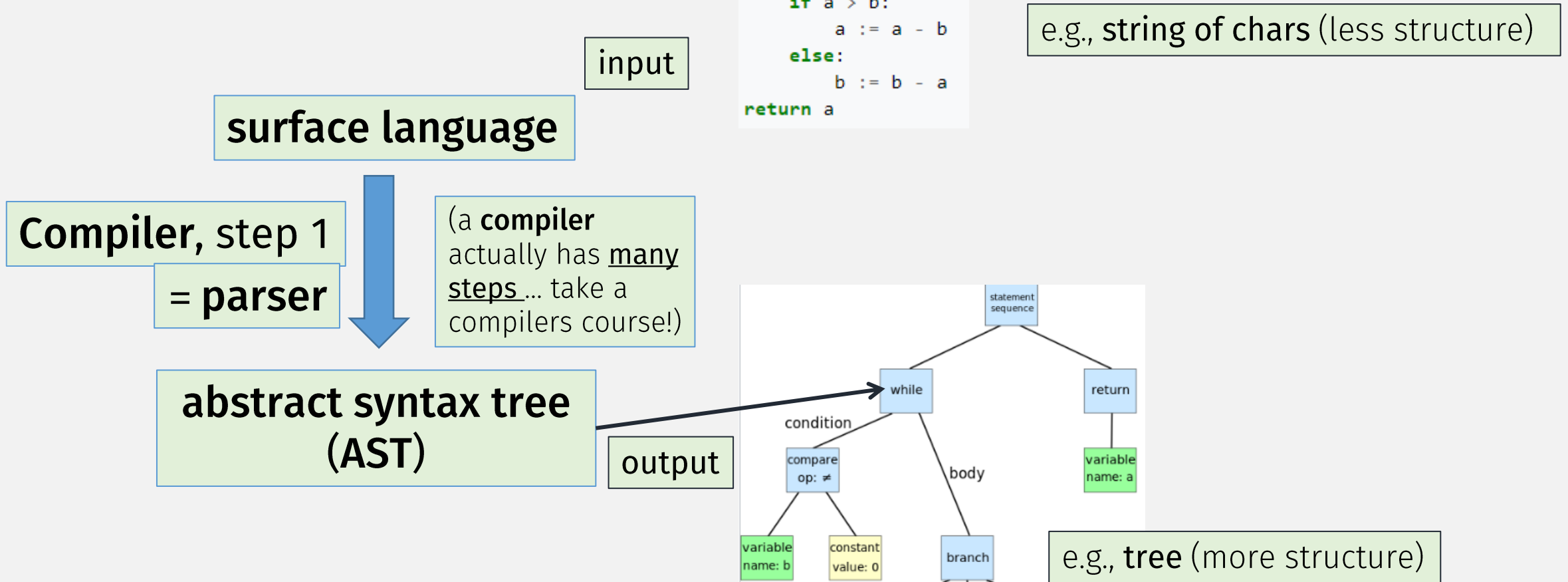| Specification langs |
| Markup (html, markdown) |
| Database (SQL) |
| Logic Program (Prolog) |
| Lazy lang (Haskell, R) |
| Functional lang (Racket) |
| JavaScript, Python |
| C# / Java |
| C++ |
| C |
| Assembly Language |
| Machine code |

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!**

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

input

surface language

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

compiler

This itself is a program

target language

output

e.g., **tree** (more structure)
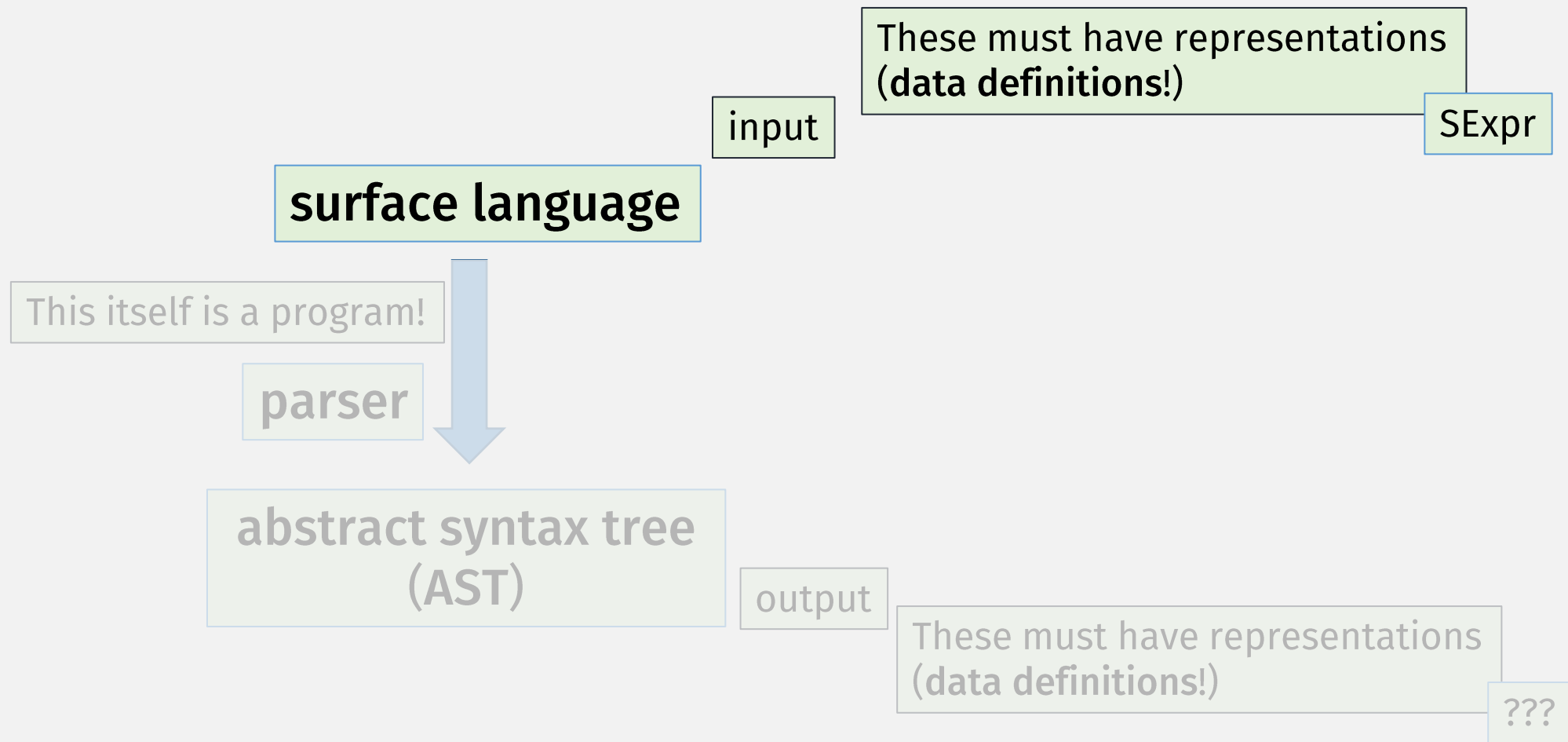
Semantics
- Specifies: meaning of language <u>structures</u>
- So: to "run" a program, we need to see the structure first

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)

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

input

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

This itself is a program!

parser

abstract syntax tree
(AST)

output

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

???

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

SExpr

input

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 <u>guards</u> must distinguish the different cases

"getters"

Recursive call(s)

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)

43

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

When a **Data Definition** is an **itemization** of **compound data** ...
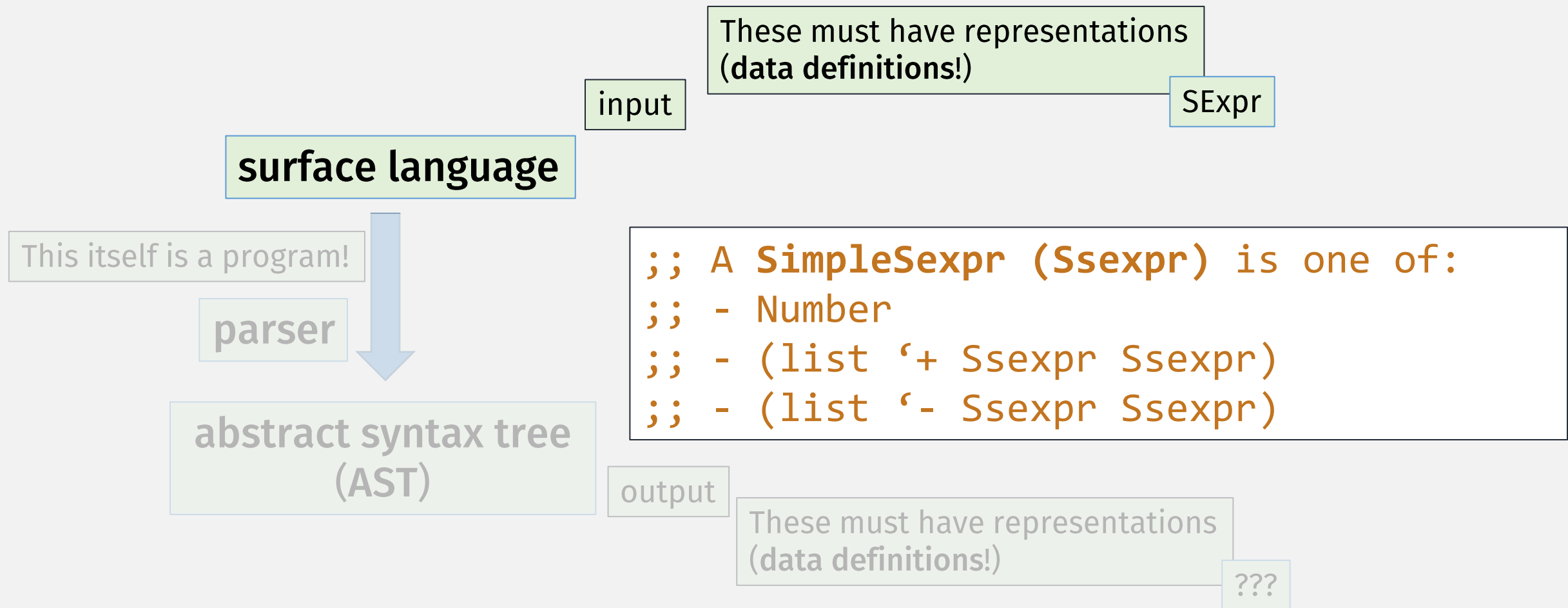
- **Template** =
  - ~~cond to distinguish cases~~
  - match = cond + getters
  - recursive calls

> 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) … ]))
```
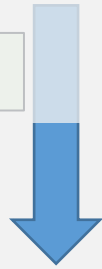
VS

```
(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)) … ]))
```

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

input

SExpr

surface language

This itself is a program!

parser

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

abstract syntax tree (AST)

output

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

???

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

In-class Coding 11/4 #1 (HW9): 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])
```

In-class Coding 11/4 #2 (HW9): eval

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

```
;; A Result is a … ???
```

```
;; 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])
```

```
;; eval-ssexpr : Ssexpr -> Result
(define eval-ssexpr
  (compose eval-ast parse)
```