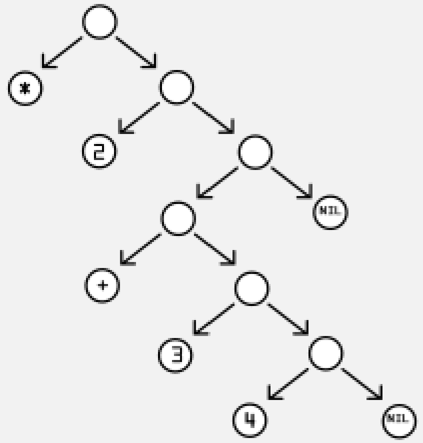
UMass Boston Computer Science

CS450 High Level Languages

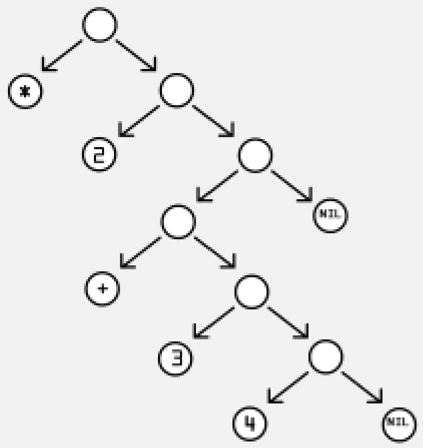
Intertwined Data

Thursday, April 3, 2025



Logistics

- **HW 8 out** (extra credit)
  - <u>due:</u> Tues 4/8 11am EST
  - NOTE: No late days allowed



S-expression (from `wikipedia`)

Last Time

Intertwined Data Definitions

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

Valid Racket Programs

Examples

- 1
- “one”
- (+ 1 2)

Data Def

```
;; 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
```

Written with a single quote, e.g., `'+`

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
;; - List<???>
;; - Tree<???>
```

```
;; An Atom is a:
;; - 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 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 **set of Data Definitions** that **reference each other**
- **Templates** should be **defined together …**

```
;; 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 **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) ...)
```

???

Intertwined Templates

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

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

```
;; 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)) ...]))
```

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

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

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) ... ]
   [(symbol? a) ... ]))
```

S-expressions

- A common real-world data definition!
  - For **representing code**
  - Or **any tree-like data / document**

- Equivalent: **XML**

  Uses:
  - web API queries, e.g., **RSS, Atom, Google, MS**
  - Documents: **MS Office documents, SVG images**
  - Code: **JSX (React)**

- Similar: **JSON**

  Uses:
  - web API queries: **Twitter, Facebook, Github**
  - Documents: **config files (yaml, node.js)**
  - Code: **JS objects!**

- Submit: GradeScope

In-class Coding 4/3: Counting Symbols

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

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

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

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

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

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

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 a)
 (cond
   [(number? a) ... ]
   [(string? a) ... ]
   [(symbol? a) ... ]))
```

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

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 a)
 (cond
   [(number? a) ... ]
   [(string? a) ... ]
   [(symbol? a) ... ]))
```

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

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 a)
 (cond
   [(symbol? a)
    (if (symbol=? sym a) 1 0)]
   [else 0]))
```

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

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 a)
 (cond
   [(symbol? a)
    (if (symbol=? sym a) 1 0)]
   [else 0]))
```

```
;; count-ptree : Symbol ProgTree -> Nat
(define (count-ptree sym pt)
  (cond
   [(empty? pt) 0]
   [else  ... (sexpr-fn (first pt)) ... (ptree-fn (rest pt)) ...]))
```

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 a)
 (cond
   [(symbol? a)
    (if (symbol=? sym a) 1 0)]
   [else 0]))
```

```
;; count-ptree : Symbol ProgTree -> Nat
(define (count-ptree sym pt)
  (cond
   [(empty? pt) 0]
   [else (+ (count sym (first pt))
            (count-ptree sym (rest pt)))]))
```

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 a)
 (cond
   [(symbol? a)
    (if (symbol=? sym a) 1 0)]
   [else 0]))
```

```
;; count-ptree : Symbol ProgTree -> Nat
(define (count-ptree sym pt)
  (cond
   [(empty? pt) 0]
   [else (+ (count sym (first pt))
            (count-ptree sym (rest pt)))]))
```

A “Racket Prog” = S-expression!

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

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

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

```
(define (atom-fn a)

      mber? a) ... ]
      ring? a) ... ]
   [(symbol? a) ... ]))
```

An **S-expression** is the **syntax** of a **Racket program**

```
;; A ProgTree
;; - empty
;; - (cons RacketProg Sexpr ProgTree)
```

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

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., **sentence** = A valid program!

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!)

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

A: The **result of "running" it!**

... but **how does a program "run"?**

Semantics

- Specifies: **"meaning" of language (constructs)**

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**

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

(Not all programs are directly interpreted)

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

A: The **result of** **“running” it!**

… but **how does** a **program** **“run”?**

From Lecture 1

NOTE: This hierarchy is *approximate*

"high" level
(easier for humans to understand)

"declarative"

"imperative"

"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

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

“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!**

input

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

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

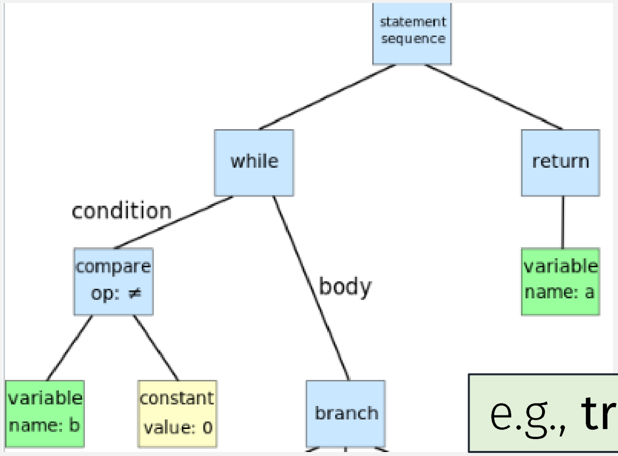
surface language

compiler

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, 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

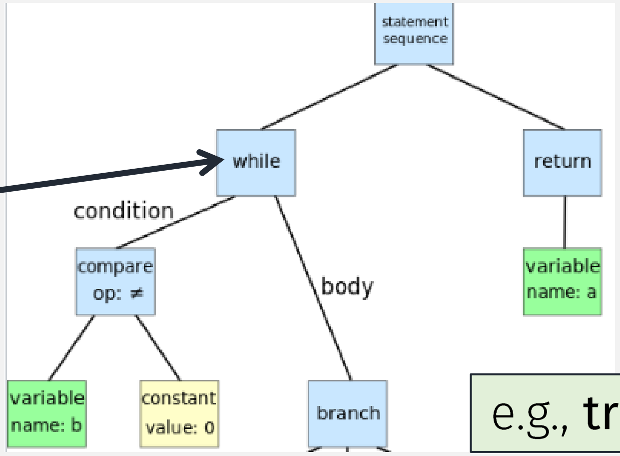
Compiler, step 1

= **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!**)

input

SExpr

surface language

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

Data Definition Template

Getting complicated ...

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

"getters"

Recursive call(s)

Interlude: quoting and quasi-quoting

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

equivalent

```
;; A Ssexpr is one of:
;; - Number
;; - `(+ ,Ssexpr ,Ssexpr)
;; - `(- ,Ssexpr ,Ssexpr)
```

Uses (quasi-quoting) to construct lists

QUOTING

Shorthand for **constructing S-exprs**

(nested lists of atoms)

single quote

`'(+ 1 2)` ⇨ `(list '+ 1 2)`

`'(+ 1 (+ 2 3))` ⇨ `(list '+ 1 (list '+ 2 3))`

QUASI-QUOTING

Like quoting but **allows "escapes"**

(to "splice in" <u>computed</u> s-exprs)

backtick

`` `(+ 1 2) `` ⇨ `(list '+ 1 2)`

`` `(+ 1 ,(+ 2 3)) `` ⇨ `(list '+ 1 5)`

Comma (only allowed inside quasiquote

Getting complicated …

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 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

“getters”

Recursive call(s)

Getting complicated …

Data Definition Template

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

- **Template =**
  - ~~cond to distinguish cases~~
  - ???
  - recursive calls

```
;; A Ssexpr is one of:
;; - Number
;; - `(+ ,Ssexpr ,Ssexpr)
;; - `(- ,Ssexpr ,Ssexpr)
```

Use (quasi-quoting) to construct lists

```
(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)) … ]))
```

???

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 Ssexpr is one of:
;; - Number
;; - `(+ ,Ssexpr ,Ssexpr)
;; - `(- ,Ssexpr ,Ssexpr)
```

Use (quasi-quoting) to construct lists

```
(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

???

Match patterns

Symbols match exactly

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

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
|	(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 **`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

With **`match`**

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

vs

With **accessors** and **predicates**

```
(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

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

```
(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))   … ])
```

With **accessors** and **predicates**

vs

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

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

With `match`

`match` can be more concise and readable