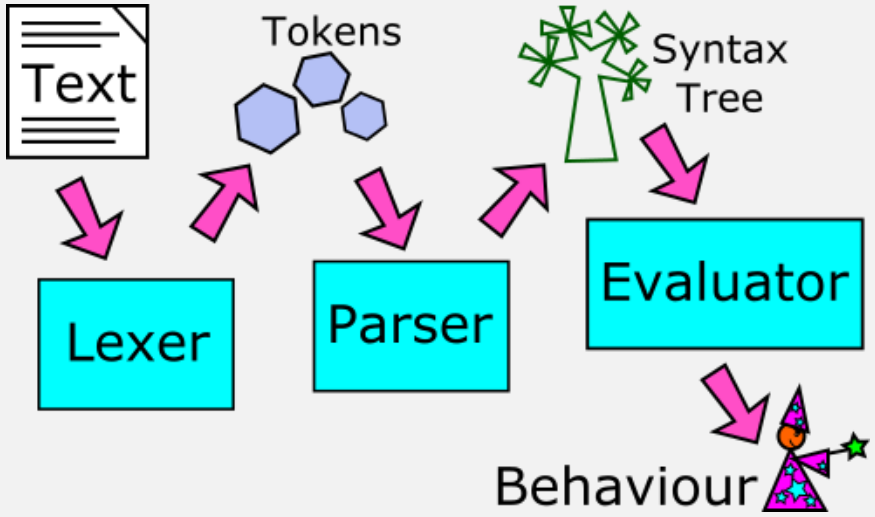
UMass Boston Computer Science

CS450 High Level Languages

Parsing, ASTs

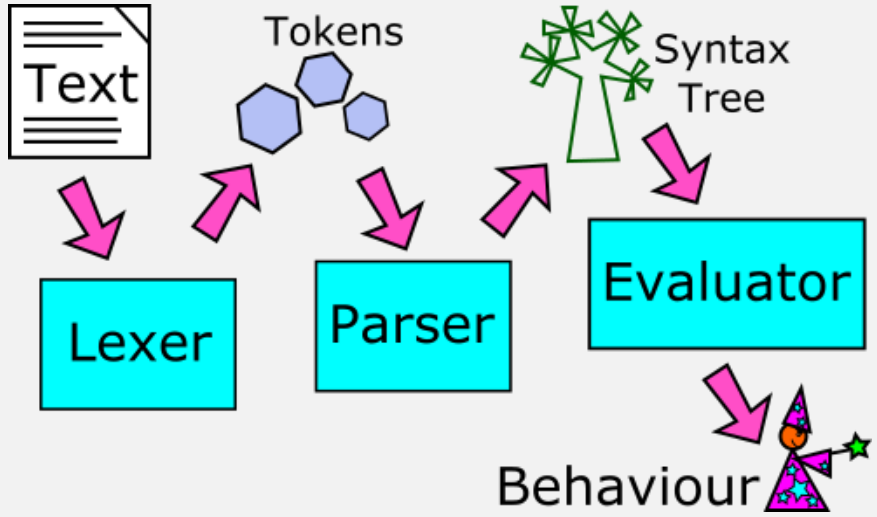
Tuesday, April 8, 2025



Logistics

- HW 8 in
  - due: Tues 4/8, 11am EST

- HW 9 out
  - due: Tues 4/15, 11am EST



"How to Write a Programming Language" https://accu.org/journals/overload/26/146/balaam_2532/

Last Time

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** "sentence": S-expressions
  - Valid **Python** "sentence": follows Python grammar (with whitespace!)

Semantics

- Specifies: "meaning" of language (constructs)

Syntax vs Semantics (Programming Language)

Syntax

- Specifies: valid language constructs
  - E.g., Valid **Racket** "sentence": S-expressions
  - Valid **Python** "sentence": follows Python grammar (with 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**"?

Giving Meaning to, i.e., Running, Programs

```
;; eval : Program -> Result
;; "runs" a given "Program", producing a "Result"
```

An "**eval**" function **turns** a "**program**" into a "**result**"

more generally called an **interpreter**

(Not all programs are directly interpreted)

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

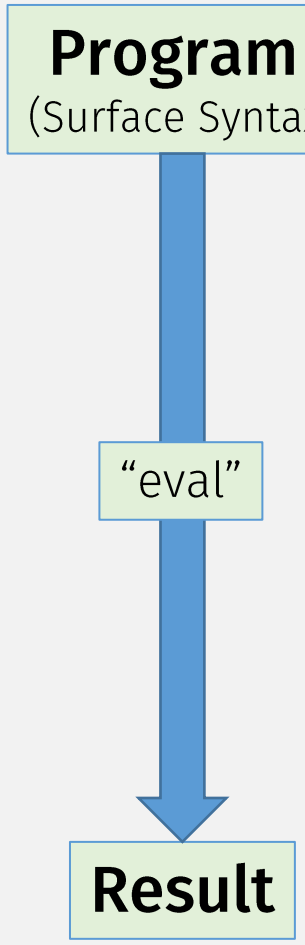
Q: What is the "**meaning**" of a program?

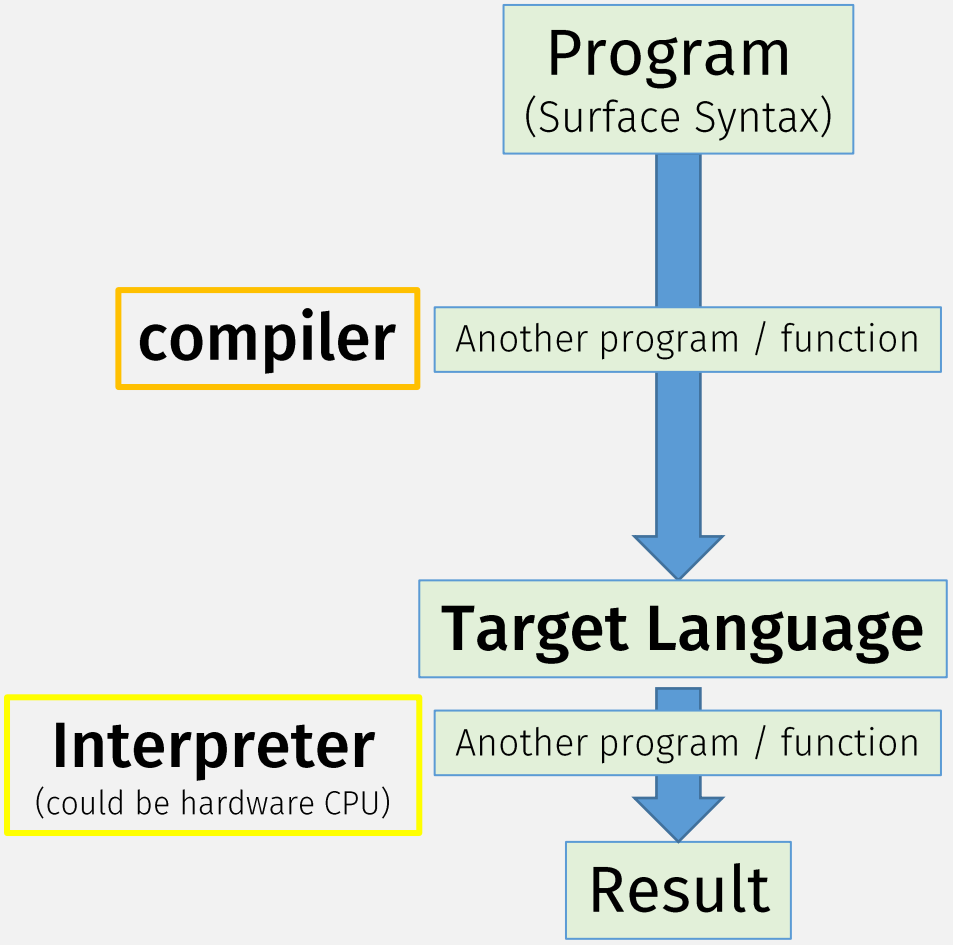
A: The **result** of "**running**" it!

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

Write a function!

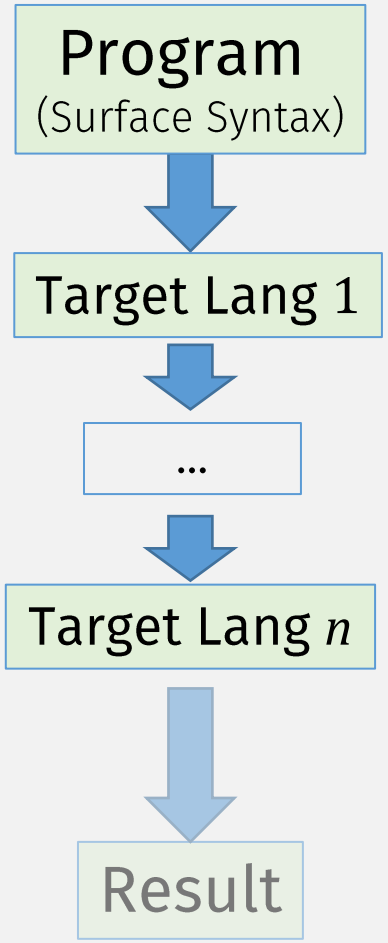
Giving Meaning to, i.e., Running, Programs





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

compiler



Compilers often have multiple steps

Parsing

Program that does this is called
Parser
(compiler's first step)

Program
(Surface Syntax)

Program
(tree)

...

Target Lang n

Result

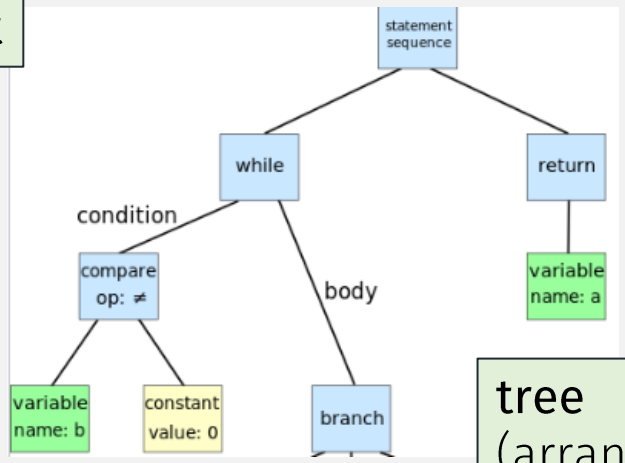
input

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

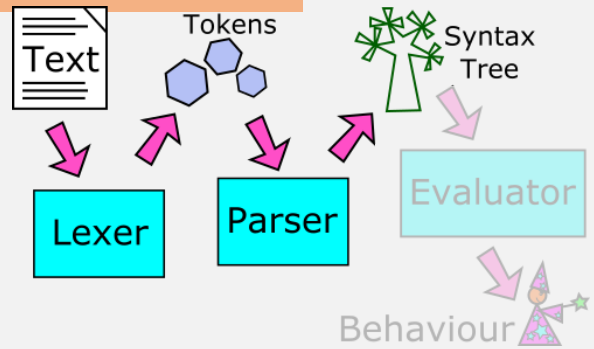
Typically, **string of chars**
(not arranged by features)

Potentially really complicated
(take a compiler's course)

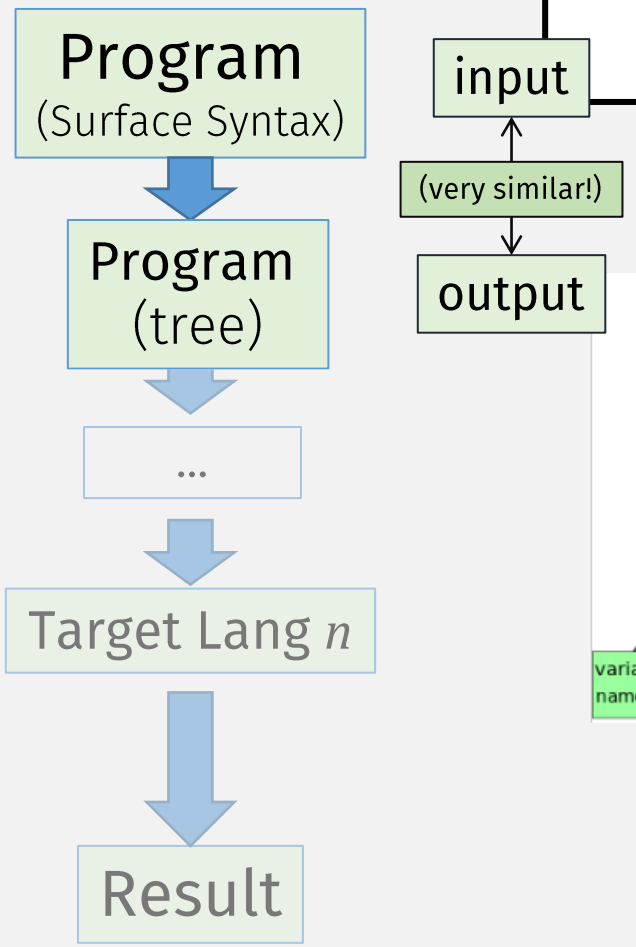
output

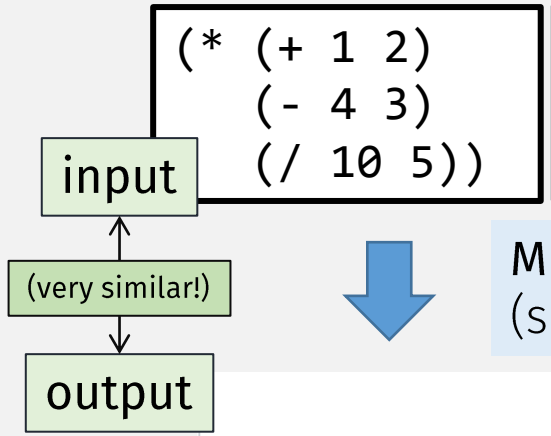


tree
(arranged by features)



Parsing – This Course

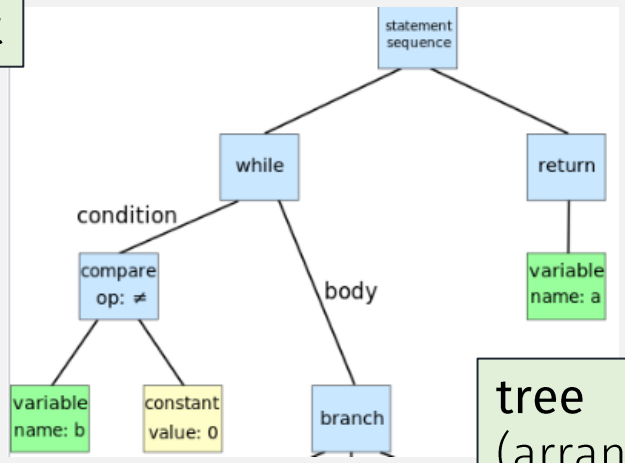


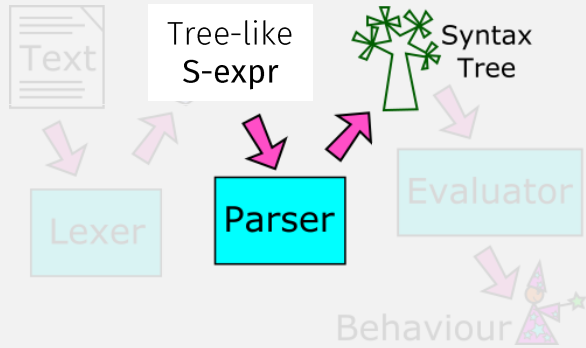


This course: **S-expressions**
(already tree-like and arranged by features)

Already have a data definition!

Much simpler
(save hard stuff for compilers course)





tree
(arranged by features)

This course: **S-expressions** (already tree-like and arranged by features)

Already have a data definition!

```
;; A Program (Simple Sexpr) is one of:
;; - Number
;; - (list '+ Program Program)
;; - (list '× Program Program)
```

NOTE: don't use "checked" constructors here
(this is **surface syntax** of the program, normally "raw strings")

S-Expression Template

A little verbose …

```
;; A Program (SSexpr) is one of:
;; - Number
;; - (list ‘+ Program Program)
;; - (list ‘× Program Program)
```

```
(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 Program is one of:
;; - Number
;; - (list '+ Program Program )
;; - (list '× Program Program )
```

equivalent

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

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" computed s-exprs)

backtick → `` `(+ 1 2) `` ⇨ `(list '+ 1 2)`

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

Comma (only allowed inside quasiquote

S-Expression Template

A little verbose …

```
;; A Program (SSexpr) is one of:
;; - Number
;; - (list '+ Program Program)
;; - (list '× Program Program)
```

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

```
;; A Program (SSexpr) is one of:
;; - Number
;; - `(+ ,Program ,Program)
;; - `(× ,Program ,Program)
```

```
(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

Use (quasi-quoting) to construct lists

"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
| (*id* *id*)	match anything, bind identifier
| _	match anything
| *literal*	match literal
| (quote *datum*)	match `equal?` value
| (list *lvp* ...)	match sequence of *lvp*s
| (list-rest *lvp* ... *pat*)	match *lvp*s consed onto a *pat*
| (list* *lvp* ... *pat*)	match *lvp*s consed onto a *pat*
| (list-no-order *pat* ...)	match *pat*s in any order
| (list-no-order *pat* ... *lvp*)	match *pat*s in any order
| (vector *lvp* ...)	match vector of *pat*s
| (hash-table (*pat* *pat*) ...)	match hash table
| (hash-table (*pat* *pat*) ...+ *ooo*)	match hash table
| (cons *pat* *pat*)	match pair of *pat*s
| (mcons *pat* *pat*)	match mutable pair of *pat*s
| (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 *pat*s match
| (or *pat* ...)	match when any *pat* match
| (not *pat* ...)	match when no *pat* matches
| (app *expr* pats ...)	match (*expr* value) output values to *pat*s
| (? *expr* *pat* ...)	match if (*expr* value) and *pat*s
| (quasiquote *qp*)	match a quasipattern
| *derived-pattern*	match using extension

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

- **Template** =
  - ~~cond to distinguish cases~~
  - `match` = `cond` + accessors

`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)) … ]))
```

In-class Coding 4/8 (HW9): parser

```
;; A Program (Ssexpr) is a:
;; - Number
;; - `(+ ,Program ,Program)
;; - `(× ,Program ,Program)
```

```
;; parse: Program -> ???
;; Converts a Program (simple s-expr) to a ???
```

???

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

Previously

Program that does this is called **Parser** (compiler's first step)

Program (Surface Syntax)

Program (tree)

...

Target Lang n

Result

input

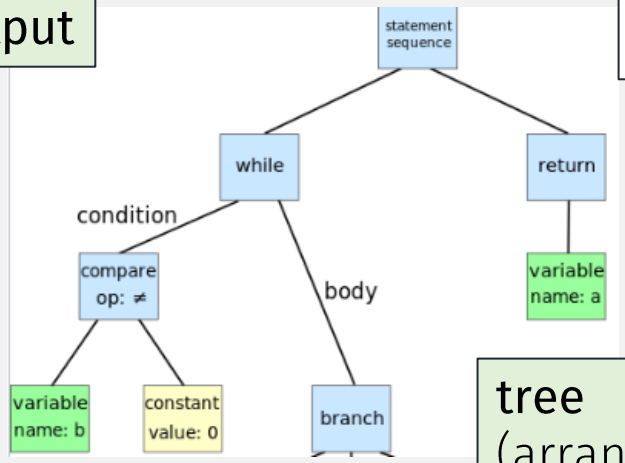
```
(* (+ 1 2)
   (- 4 3)
   (/ 10 5))
```

This course: **S-expressions** (already tree-like and arranged by features)

Already have a data definition!

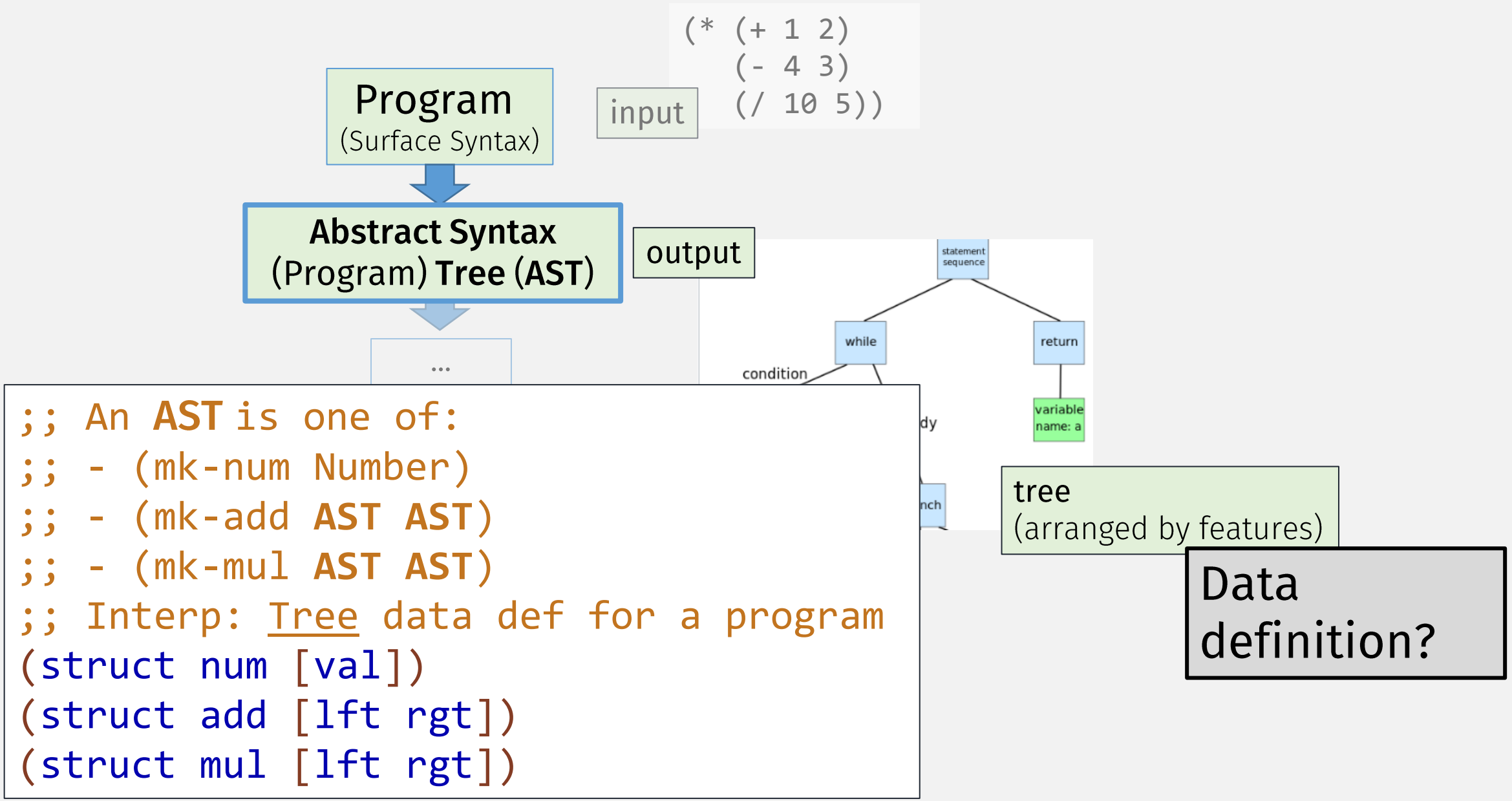
```
;; A Program is one of:
;; - Number
;; - (list '+ Program Program)
;; - (list '× Program Program)
```

output



tree (arranged by features)

Data definition?


(* (+ 1 2)
(- 4 3)
(/ 10 5))
input
Program
(Surface Syntax)
Abstract Syntax
(Program) Tree (AST)
output
...
statement sequence
while
return
condition
variable name: a
dy
nch
tree
(arranged by features)
;; An AST is one of:
;; - (mk-num Number)
;; - (mk-add AST AST)
;; - (mk-mul AST AST)
;; Interp: Tree data def for a program
(struct num [val])
(struct add [lft rgt])
(struct mul [lft rgt])
Data definition?

use “checked” constructors as usual

```
;; An AST is one of:
;; - (mk-num Number)
;; - (mk-add AST AST)
;; - (mk-mul AST AST)
;; Interp: Tree data def for a program
(struct num [val])
(struct add [lft rgt])
(struct mul [lft rgt])
```

```
(define/contract (mk-num n)
  (-> number? AST?)
  (num n))
```

contract

Unchecked constructor

```
(define/contract (mk-add x y)
  (-> AST? AST? AST?)
  (add x y))
```

???

Interlude: Inheritance and "Super" Structs

```
;; A Shape is one of:
;; - Rectangle
;; - Circle
(struct rect [w h c])
(struct circ [r c])
```

```
;; A Shape is one of:
;; - Rectangle
;; - Circle
(struct Shape [])
(struct rect Shape [w h c])
(struct circ Shape [r c])
```

"abstract" struct (implicitly defines **Shape?** predicate)

Alternatively …

"super" struct declaration

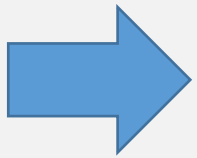
```
(define (Shape? s)
  (or (rect? s) (circ? s)))
```

e.g., if **r** = `(rect 1 2 'red)`
then both `(rect? r)` = true
and `(Shape? r)` = true

Without **superstruct**

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

(struct num [val])
(struct add [lft rgt])
(struct mul [lft rgt])
```



With **superstruct**

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

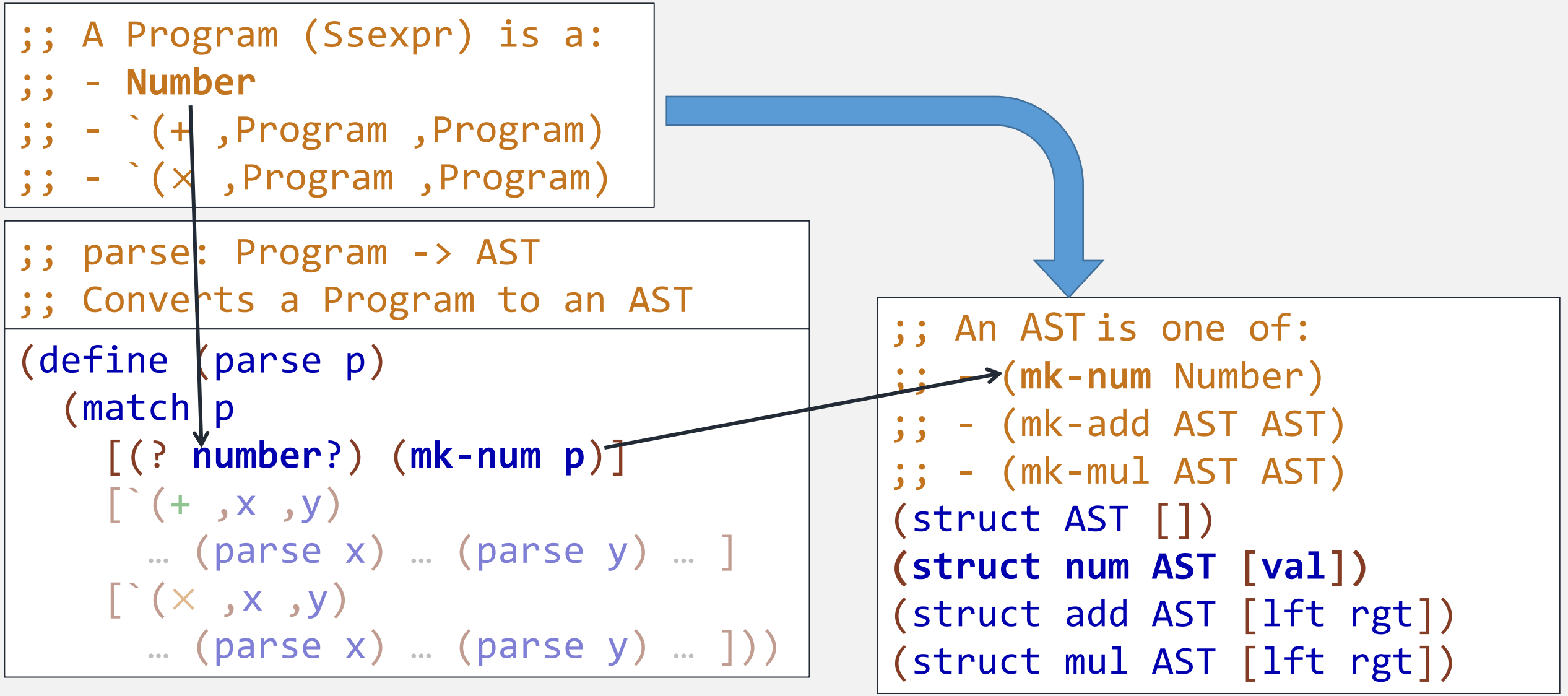
In-class Coding 4/8 (HW9): parser

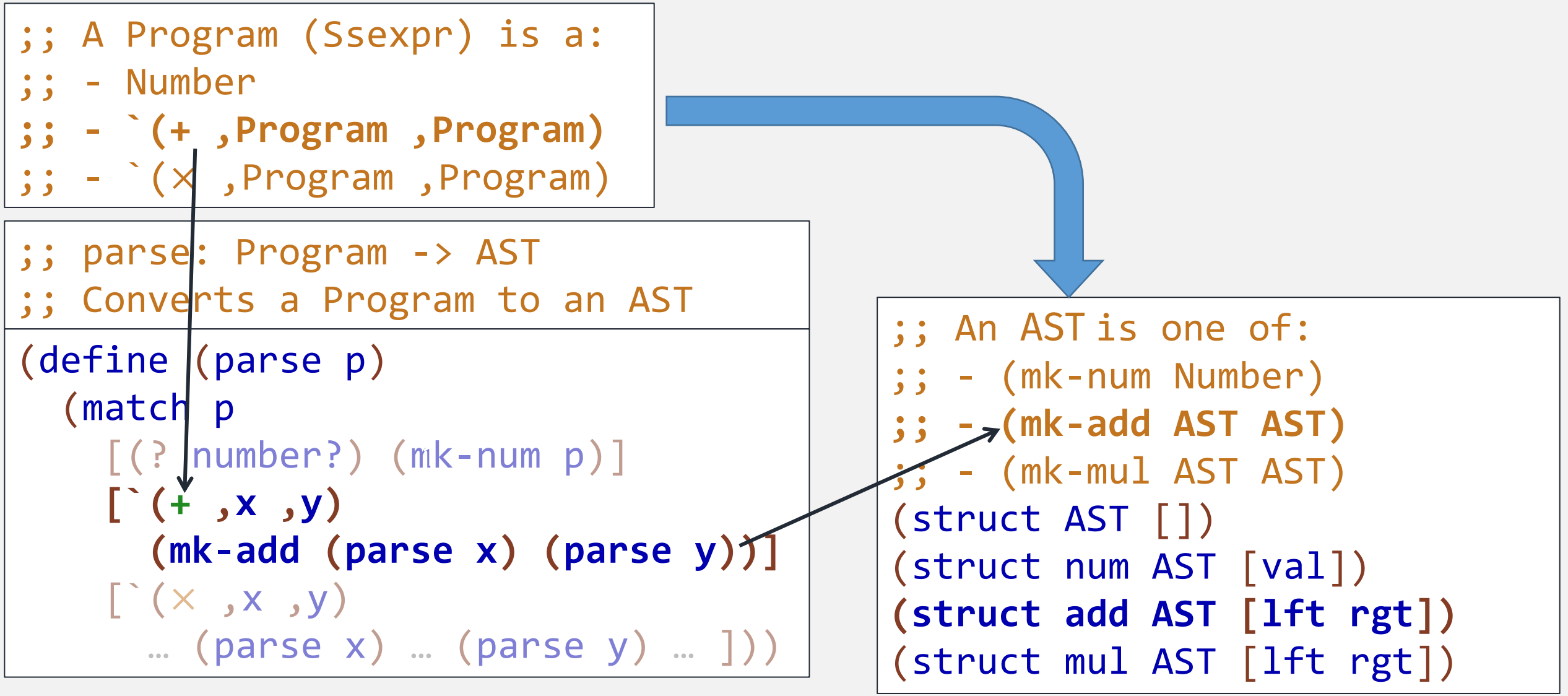
```
;; A Program (Ssexpr) is a:
;; - Number
;; - `(+ ,Program ,Program)
;; - `(× ,Program ,Program)
```

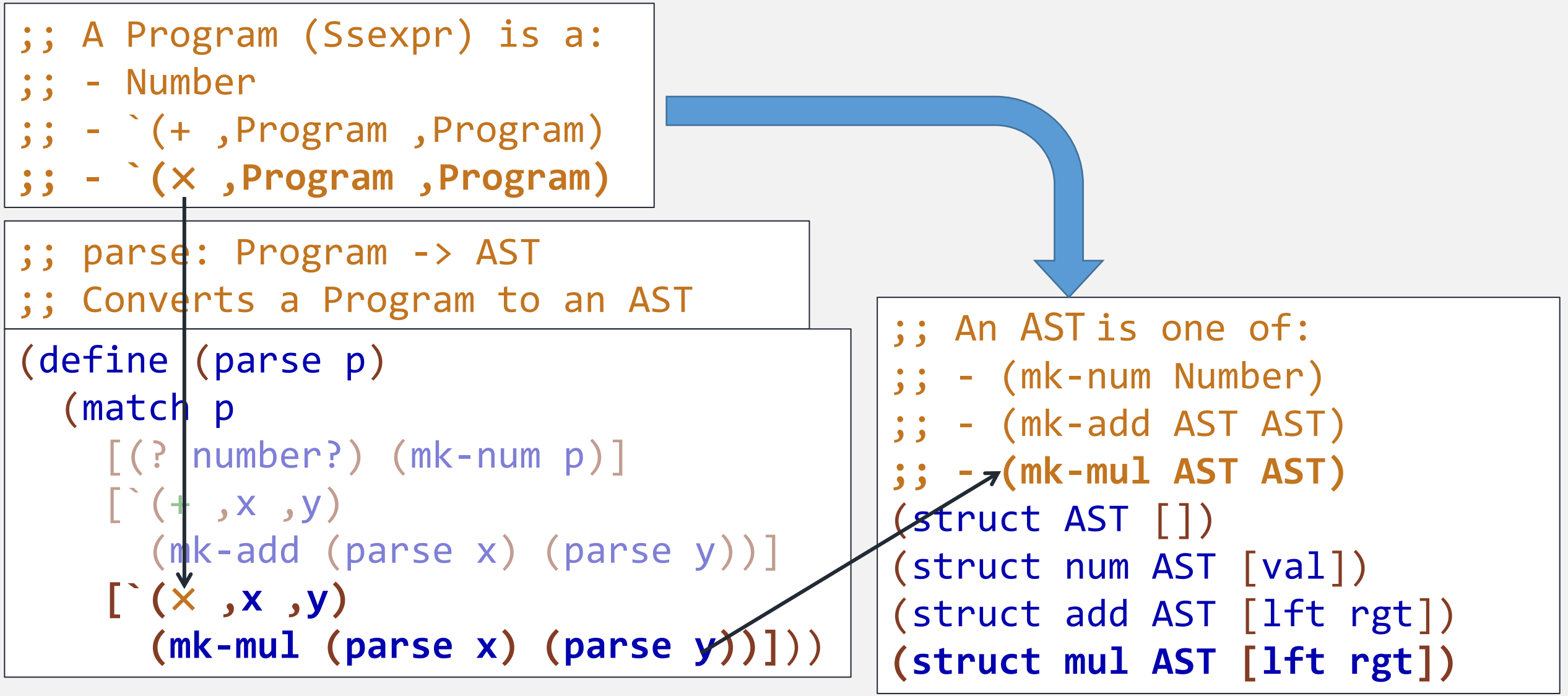
```
;; parse: Program -> AST
;; Converts a Program to an AST
(define (parse p)
  (match p
    [(? number?) … ]
    [`(+ ,x ,y)
      … (parse x) … (parse y) … ]
    [`(× ,x ,y)
      … (parse x) … (parse y) … ]))
```

TEMPLATE

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


;; A Program (Ssexpr) is a:
;; - Number
;; - `(+ ,Program ,Program)
;; - `(× ,Program ,Program)
;; parse: Program -> AST
;; Converts a Program to an AST
(define (parse p)
(match p
[(? number?) (mk-num p)]
[`(+ ,x ,y)
… (parse x) … (parse y) …]
[`(× ,x ,y)
… (parse x) … (parse y) …]))
;; An AST is one of:
;; - (mk-num Number)
;; - (mk-add AST AST)
;; - (mk-mul AST AST)
(struct AST [])
(struct num AST [val])
(struct add AST [lft rgt])
(struct mul AST [lft rgt])


;; A Program (Ssexpr) is a:
;; - Number
;; - `(+ ,Program ,Program)
;; - `(× ,Program ,Program)
;; parse: Program -> AST
;; Converts a Program to an AST
(define (parse p)
(match p
[(? number?) (mk-num p)]
[`(+ ,x ,y)
(mk-add (parse x) (parse y))]
[`(× ,x ,y)
… (parse x) … (parse y) …]))
;; An AST is one of:
;; - (mk-num Number)
;; - (mk-add AST AST)
;; - (mk-mul AST AST)
(struct AST [])
(struct num AST [val])
(struct add AST [lft rgt])
(struct mul AST [lft rgt])


;; A Program (Ssexpr) is a:
;; - Number
;; - `(+ ,Program ,Program)
;; - `(× ,Program ,Program)
;; parse: Program -> AST
;; Converts a Program to an AST
(define (parse p)
(match p
[(? number?) (mk-num p)]
[`(+ ,x ,y)
(mk-add (parse x) (parse y))]
[`(× ,x ,y)
(mk-mul (parse x) (parse y))]))
;; An AST is one of:
;; - (mk-num Number)
;; - (mk-add AST AST)
;; - (mk-mul AST AST)
(struct AST [])
(struct num AST [val])
(struct add AST [lft rgt])
(struct mul AST [lft rgt])

```
;; A Program (Ssexpr) is a:
;; - Number
;; - `(+ ,Program ,Program)
;; - `(× ,Program ,Program)
```

```
;; parse: Program -> AST
;; Converts a Program to an AST
```

```
(define (parse p)
  (match p
    [(? number?) (mk-num p)]
    [`(+ ,x ,y)
      (mk-add (parse x) (parse y))]
    [`(× ,x ,y)
      (mk-mul (parse x) (parse y))]))
```

TEMPLATE MAKES THIS EASY!

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

Previously

Program
(Surface Syntax)

Parser

Abstract Syntax
(Program) **Tree (AST)**

...

Target Lang n

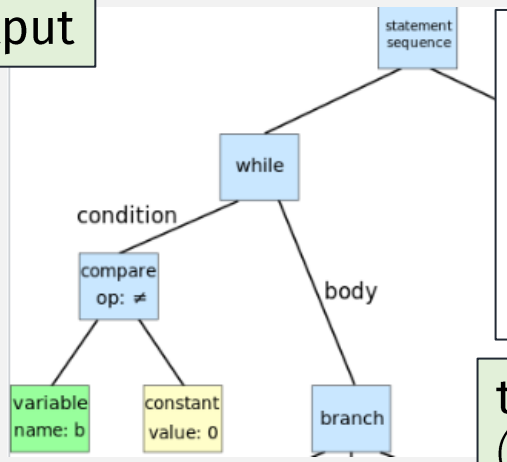
Result

input

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

```
;; A Program (Ssexpr) is a:
;; - Number
;; - `(+ ,Program ,Program)
;; - `(× ,Program ,Program)
```

output




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

tree
(arranged by features)

Program
(Surface Syntax)

Abstract Syntax
(Program) **Tree (AST)**

Interpreter
(could be hardware CPU)

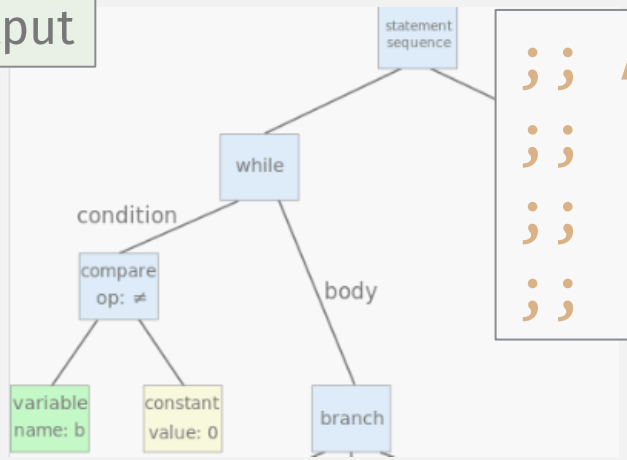
Another program / function

Result

input

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

output



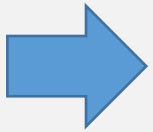

```
;; An AST is one of:
;; - (mk-num Number)
;; - (mk-add AST AST)
;; - (mk-mul AST AST)
```

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

Data definition?

In-class Coding 4/8 #2: **run**

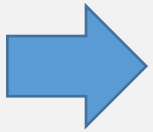
```
;; An AST is one of:
;; - (mk-num Number)
;; - (mk-add AST AST)
;; - (mk-mul AST AST)
```



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

```
;; run: AST -> Result
;; Computes the Result of running the given program AST
```

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



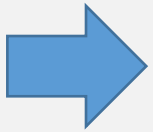
```
;; A Result is a:
;; - Number
```

```
;;
;;                                    ing the given program AST
```

TEMPLATE?

```
(define (ast-fn p)
  (cond
    [(num? p) … ]
    [(add? p)  … (ast-fn (add-lft p))
               … (ast-fn (add-rgt p))  … ]
    [(mul? p)  … (ast-fn (mul-lft p))
               … (ast-fn (mul-rgt p))  … ]))
```

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



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

```
;;
;;                                    ing the given program AST
```

TEMPLATE --- WITH **match**

```
(define (ast-fn p)
  (cond match p
    [(num n) … ]
    [(add x y)  … (ast-fn x) …
                … (ast-fn y) … ]
    [(mul x y)  … (ast-fn x) …
                  (ast-fn y) … ])
```

Struct name

Struct patterns

Extracts and names fields

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

```
(define (ast-fn p)
  (cond
    [(num? p) … ]
    [(add? p)  … (ast-fn (add-lft p))
               … (ast-fn (add-rgt p))  … ]
    [(mul? p)  … (ast-fn (mul-lft p))
               … (ast-fn (mul-rgt p))  … ])
```

vs

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

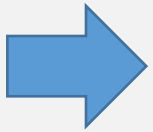
With **`match`**

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

`match` can be more concise and readable

In-class Coding 4/8 #2: **run** (HW9)

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



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

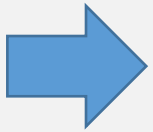
```
;; run: AST -> Result
;; Computes the Result of running the given program AST
```

TEMPLATE

```
(define (run p)
  (match p
    [(num n) … ]
    [(add x y)  … (run x) …
                … (run y) … ]
    [(mul x y)  … (run x) …
                … (run y) … ])
```

In-class Coding 4/8 #2: **run**

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



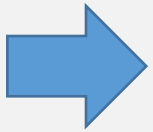
```
;; A Result is a:
;; - Number
```

```
;; run: AST -> Result
;; Computes the Result of running the given program AST
(define (run p)
  (match p
    [(num n) n]
    [(add x y)  … (run x) …
                … (run y) … ]
    [(mul x y)  … (run x) …
                … (run y) … ])
```

How to combine Results?

In-class Coding 4/8 #2: **run**

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



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

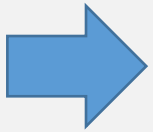
```
;; run: AST -> Result
;; Computes the Result of running the given program AST
(define (run p)
  (match p
    [(num n) n]
    [(add x y) (+ (run x)
                  (run y))]
    [(mul x y) … (run x) …
               … (run y) … ]))
```

Racket **+** gives **semantics** to our **new** language "+" operator

How to combine?

In-class Coding 4/8 #2: **run**

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



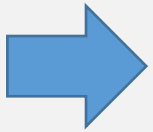
```
;; A Result is a:
;; - Number
```

```
;; run: AST -> Result
;; Computes the Result of running the given program AST
(define (run p)
  (match p
    [(num n) n]
    [(add x y) (+ (run x)
                  (run y))]
    [(mul x y) (* (run x)
                  (run y))]))
```

Racket ***** gives **semantics** to our **new** language “×” **operator**

In-class Coding 4/8 #2: **run**

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



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

```
;; run: AST -> Result
;; Computes the Result of running the given program AST
(define (run p)
  (match p
    [(num n) n]
    [(add x y) (+ (run x)
                  (run y))]
    [(mul x y) (* (run x)
                  (run y))]))
```

TEMPLATE MAKES THIS EASY!