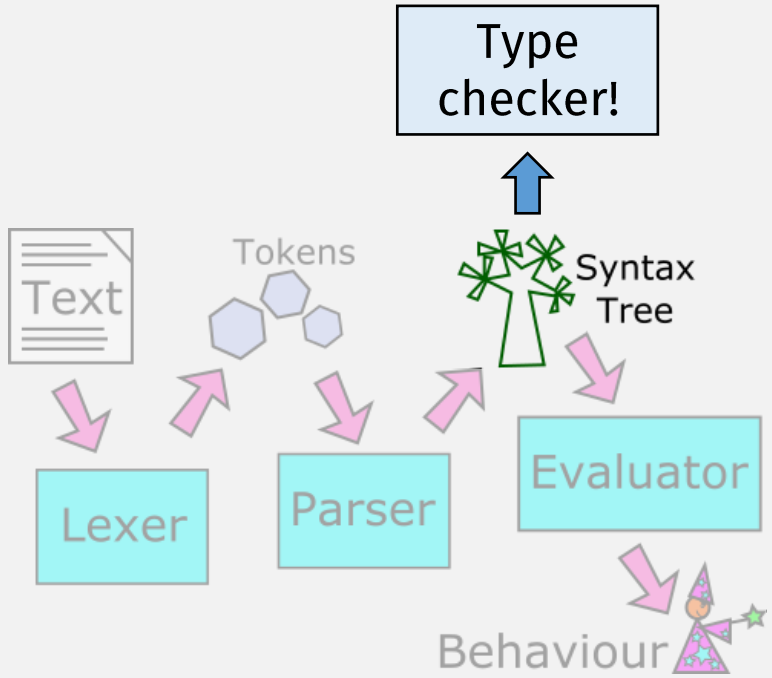
UMass Boston Computer Science

CS450 High Level Languages

Type Checking

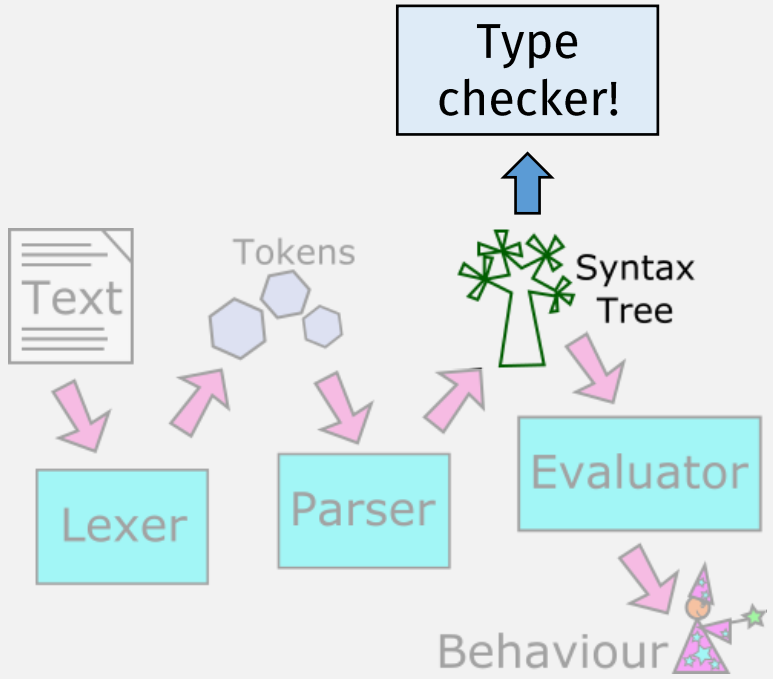
Thursday, April 30, 2026



Logistics

- HW 11 in
  - ~~due: Thurs 4/30, 11am EST~~
  - due: Sat 5/2, 11am EST

- HW 12 out
  - due: Thurs 5/7, 11am EST



Last Time

Interlude: Recursion vs Iteration

- **Recursive** functions have a **self-reference**

```
def factorialUsingRecursion(n):
    if (n == 0):
        return 1;

    # recursion call
    return n * factorialUsingRecursion(n - 1);
```

- **Iterative** code typically uses a **loop**

```
def factorialUsingIteration(n):
    res = 1;

    # using iteration
    for i in range(2, n + 1):
        res *= i;

    return res;
```

Recursion vs Iteration: Which is “Better”?

Recursive vs. Iterative Solutions

Recursive algorithms can be very space inefficient. Each recursive call adds a new layer to the stack, which means that if your algorithm recurses to a depth of n, it uses at least O(n) memory.

For this reason, it's often better to implement a recursive algorithm iteratively. *All* recursive algorithms can be implemented iteratively, although sometimes the code to do so is much more complex. Before diving into recursive code, ask yourself how hard it would be to implement it iteratively, and discuss the tradeoffs with your interviewer.

Cracking the Coding Interview (2008), Ch8

r/learnprogramming • 11 yr. ago

[Best Practices] Recursion. Why is it generally avoided and when is it acceptable?

stackoverflow

Are recursive methods always better than iterative methods in Java?

Recursion vs Iteration: Conventional Wisdom

Iteration

Strengths:

- Iteration can be used to repeatedly execute a set of statements without the overhead of function calls and without using stack memory.
- Iteration is faster and more efficient than recursion.
- It's easier to optimize iterative codes, and they generally have polynomial time complexity.
- They are used to iterate over the elements present in data structures like an array, set, map, etc.
- If the iteration count is known, we can use *for* loops; else, we can use *while* loops, which terminate when the controlling condition becomes false.

Weaknesses:

- In loops, we can go only in one direction, i.e., we can't go or transfer data from the current state to the previous state that has already been executed.
- It's difficult to traverse trees/graphs using loops.
- Only limited information can be passed from one iteration to another, while in recursion, we can pass as many parameters as we need.

Iteration is <u>good</u> with *non-recursive data*

Iteration is <u>bad</u> with *recursive data*!

Recursion better when **accumulators** are needed

https://www.interviewkickstart.com/learn/difference-between-recursion-and-iteration

Recursion vs Iteration: Conventional Wisdom

Recursion

Strengths:

- It's easier to code the solution using recursion when the solution of the current problem is dependent on the solution of smaller similar problems.
  - fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
  - factorial(n) = n * factorial(n-1)
- Recursive codes are smaller and easier to understand. ☑
- We can pass information to the next state in the form of parameters and return information to the previous state in the form of the return value.
- It's a lot easier to perform operations on trees and graphs using recursion.

Weaknesses:

- The simplicity of recursion comes at the cost of time and space efficiency.
- It is much slower than iteration due to the overhead of function calls and control shift from one function to another.
- It requires extra memory on the stack for each recursive call. This memory gets deallocated when function execution is over.
- s difficult to optimize a recursive code, and they generally have higher time complexity an iterative codes due to overlapping subproblems.

Recursion better when **accumulators** are needed

Use recursion with *recursive data*!

Recursion is slow
Recursion is slow
Recursion is slow
Recursion is slow

Investigate:
Is recursion is slower??

https://www.interviewkickstart.com/learn/difference-between-recursion-and-iteration

Recursion vs Iteration: In Racket

Racket Recursion

```
;; sum-to : Nat -> Nat
;; Sums the numbers in the interval [0, x]
(define (sum-to x)
  (if (zero? x)
      x
      (+ x (sum-to (sub1 x)))))
```

Conclusion?
Recursion is **slower?**

WAIT!
Racket does not have **“for” loops**

```
(define BIG-NUMBER 999999)

(time (sum-to BIG-NUMBER))
; cpu time: 202 real time: 201 gc time: 156
```

Racket “Iteration”

```
(time (for/sum ([x (add1 BIG-NUMBER)]) x))
; cpu time: 15 real time: 6 gc time: 0
```

Recursion vs Iteration: In Racket

Conclusion?
Recursion is <u>not</u>
slower than iteration?

equivalent

Racket Recursion

```
;; iterative-sum-to : Nat -> Nat
;; Sums the numbers in the interval [0, x]
(define (iterative-sum-to x result)
  (if (zero? x)
      result
      (iterative-sum-to (sub1 x) (+ x result))))
```

accumulator

```
(time (iterative-sum-to BIG-NUMBER 0))
; cpu time: 15 real time: 13 gc time: 0
```

"for" in **Racket** is just a **macro** (i.e., "**syntactic sugar**") for a **recursive function**

Racket "Iteration"

```
(time (for/sum ([x (add1 BIG-NUMBER)]) x))
; cpu time: 15 real time: 6 gc time: 0
```

Tail Calls

From Wikipedia, the free encyclopedia

In computer science, a **tail call** is a subroutine call performed as the final action of a procedure. If the target of a tail is the same subroutine, the subroutine is said to be **tail recursive**, which is a special case of direct recursion. **Tail recursion** (or **tail-end recursion**) is particularly useful, and is often easy to optimize in implementations.

Tail calls can be implemented without adding a new stack frame to the call stack.

Fixes the problem?

Recursive vs. Iterative Solutions

Recursive algorithms can be very space inefficient. Each recursive call adds a new layer to the stack, which means that if your algorithm recurses to a depth of n, it uses at least $O(n)$ memory.

For this reason, it's often better to implement a recursive algorithm iteratively. *All* recursive algorithms can

Recursion vs Iteration: In Racket

Conclusion?
Recursion is <u>not</u>
slower than iteration?

Racket Recursion

```
;; iterative-sum-to : Nat -> Nat
;; Sums the numbers in the interval [0, x]
(define (iterative-sum-to x result)
  (if (zero? x)
      result
      (iterative-sum-to (sub1 x) (+ x result))))
```

Tail-recursive function

Tail-call (does not add to stack)

(Tail) recursion <u>is</u> iteration!

Recursion vs Iteration: Under the Hood

- **It makes sense that recursion** and **iteration** are **equivalent …**
  - Recursive call compiles to:
    - **`JUMP`** instruction
  - Loop compiles to:
    - **`JUMP`** instruction!

- … *except* in **languages** that make them **not equivalent!**
  - i.e., **languages** that **push extra stack frames** that are **not needed**

Tail-Calls in Other Languages

- Most functional languages (Racket, Haskell, Erlang, F#) implement proper tail calls
  (no extra stack frame)

- Some languages require an explicit annotation
  - Clojure: `recur`
  - Scala: `@tailrec`

- Most imperative languages don't properly implement tail calls
  (they add an unnecessary stack frame)
  - Python, Java, C#, Go

Guido Got It Backwards

Wednesday, April 22, 2009

Tail Recursion Elimination

I recently posted an entry in my Python History blog on the origins of Python's functional features. A side remark about not supporting tail recursion elimination (TRE) immediately sparked several comments about what a pity it is that Python doesn't do this, including links to recent blog entries by others trying to "prove" that TRE can be added to Python easily. So let me defend my position (which is that I don't *want* TRE in the language). If you want a short answer, it's simply unpythonic. Here's the long answer:

First, as one commenter remarked, TRE is incompatible with nice stack traces: when a tail recursion is eliminated, there's no stack frame left to use to print a traceback when something goes wrong later. This will confuse
cently wrote something recursive (the recursion isn't obviou
ce printed), and makes debugging hard. Providing an optio
eems wrong to me: Python's default is and should always b
be maximally helpful for debugging. This also brings me to the next issu

About Me

Guido van Rossum

Python's BDFL

View my complete profile

Blog Archive

► 2022 (2)

► 2019 (1)

Wrong!

Equivalent to saying:
"every **for** loop iteration should push a stack frame!"

Proper tail calls is about eliminating stack frames that shouldn't be there in the first place! (because it's just iteration!)

Non Tail Call

C

```c
// Program to find factorial of a number n modulo prime
int factorial(int n, int prime)
{
    if (n <= 1) {
        // base case
        return 1;
    }

    return (n * factorial(n - 1, prime) % prime) % prime;
}
```

Non-tail-call

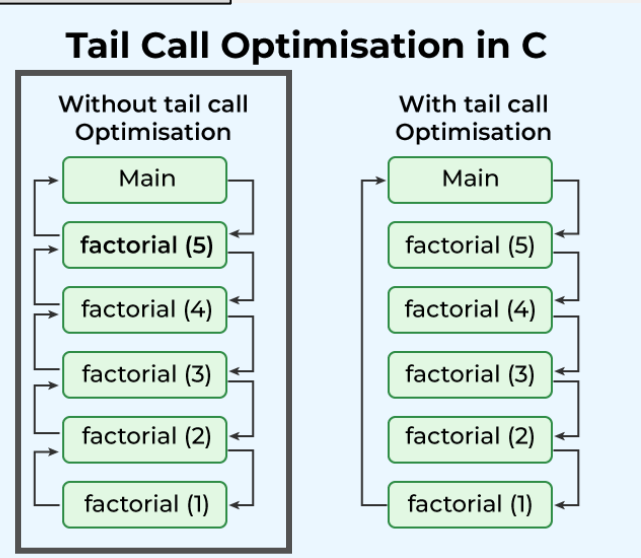
Slower, more memory

Compiler output: **godbolt.org**

x86-64 gcc 14.2 | -O3

```asm
1   factorial:
2     cmp edi, 1
3     jle .L16
4     push r15
5     mov eax, 2
6     push r14
7     push r13
8     push r12
9     lea r12d, [rdi-1]
10    push rbx
11    mov ebx, edi
12    sub rsp, 16
13    cmp edi, 2
14    je .L3
15    lea r13d, [rdi-2]
16    cmp edi, 3
17    je .L4
18    lea r15d, [rdi-3]
19    cmp edi, 4
20    je .L5
21    lea r14d, [rdi-4]
22    cmp edi, 5
23    je .L6
24    lea edi, [rdi-5]
25    mov DWORD PTR [rsp+12], esi
26    call factorial
27    mov esi, DWORD PTR [rsp+12]
28    imul eax, r14d
29    cdq
30    idiv esi
31    mov eax, edx
32    imul eax, r15d
33  .L6:
34    cdq
35    idiv esi
36    mov eax, edx
37    imul eax, r13d
38  .L5:
39    cdq
40    idiv esi
41    mov eax, edx
42    imul eax, r12d
43  .L4:
44    cdq
45    idiv esi
```

Stack push



https://www.geeksforgeeks.org/tail-call-optimisation-in-c/

Tail Calls as Loops

Compiler output: **godbolt.org**

x86-64 gcc 14.2 | -O3

c

```c
// C program to illustrate Tail Call Optimisation
int factorial(int store, int num, int prime) {
  if (num < 1) {
    // Base case
    return store;
  }

  return factorial((store%prime * num%prime)%prime, num - 1, prime);
}
```

Tail call – compiled directly to **for** loop!

Faster, less memory

```asm
1   factorial(int, int, int):
2     mov ecx, edx
3     test esi, esi
4     jle .L5
5   .L2:
6     mov eax, edi
7     cdq
8     idiv ecx
9     mov eax, edx
10    imul eax, esi
11    cdq
12    idiv ecx
13    mov edi, edx
14    sub esi, 1
15    jne .L2
16  .L5:
17    mov eax, edi
18    ret
```

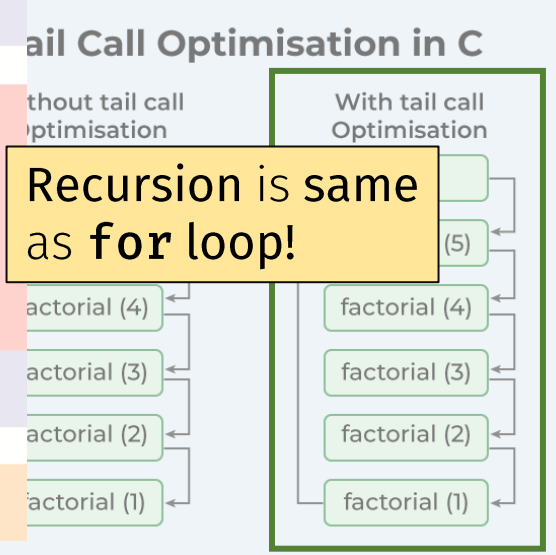
No stack push!

```c
// this function calculates factorial modulo prime
int factorial(int num, int prime) {
    int store = 1;
    for (int i = num; i > 0; i--) {
        store = (store%prime * i%prime)%prime;
    }
  return store;
}
```

for loop

```asm
1   factorial(int, int):
2     mov edx, 1
3     test edi, edi
4     jle .L1
5   .L3:
6     mov eax, edx
7     cdq
8     idiv esi
9     mov eax, edx
10    imul eax, edi
11    cdq
12    idiv esi
13    sub edi, 1
14    jne .L3
15  .L1:
16    mov eax, edx
17    ret
```

Recursion is same as **for** loop!



Some languages (with –O3 optimization) **directly compile recursion to a loop!** (because they are equivalent!)

Tail-Calls in Other Languages

- Most functional languages (Racket, Haskell, Erlang, F#) implement proper tail calls
  (no extra stack frame)

- Some languages require an explicit annotation
  - Clojure: `recur`
  - Scala: `@tailrec`

- Most imperative languages don't properly implement tail calls
  (they add an unnecessary stack frame)
  - Python, Java, C#, Go

- Some languages (JavaScript) have it (ECMAScript 6) … but don't have it

Proper Tail Calls in JavaScript

Proper Tail Calls (PTC) is a new feature in the ECMAScript 6 language. This feature was added to facilitate recursive programming patterns, both for direct and indirect recursion. Various other design patterns can benefit from PTC as well, such as code that wraps some functionality where the wrapping code directly returns the result of what it wraps. Through the use of PTC, the amount of memory needed to run code is reduced. In deeply recursive code, PTC enables code to run that would otherwise throw a stack overflow exception.

https://webkit.org/blog/6240/ecmascript-6-proper-tail-calls-in-webkit/

Feature name

proper tail calls (tail call optimisation)

Not supported in V8 (Chrome) or SpiderMonkey (Firefox) or NodeJS …

https://compat-table.github.io/compat-table/es6/

… but there's hope!

WebAssembly tail calls

Published 06 April 2023 · Tagged with WebAssembly

We are shipping WebAssembly tail calls in V8 v11.2! In this post we give a brief overview of this proposal, demonstrate an interesting use case for C++ coroutines with Emscripten, and show how V8 handles tail

https://v8.dev/blog/wasm-tail-call

Recursion vs Iteration: Conclusion

Recursion

Strengths:

- It's easier to code the solution using recursion when the solution of the current problem is dependent on the solution of smaller similar problems.
  - fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
  - factorial(n) = n * factorial(n-1)
- Recursive codes are smaller and easier to understand.
- We can pass information to the next state in the form of parameters and return information to the previous state in the form of the return value.
- It's a lot easier to perform operations on trees and graphs using recursion.

Weaknesses:

- The simplicity of recursion comes at the cost of time and space efficiency.
- It is much slower than iteration due to the overhead of function calls and control shift from one function to another.
- t requires extra memory on the stack for each recursive call. This memory gets deallocated when function execution is over.
- t is difficult to optimize a recursive code, and they generally have higher time complexity than iterative codes due to overlapping subproblems.

Recursion is (usually) easier to read

Recursion better when **accumulators** are needed

Use recursion with *recursive data*!

Recursion is slower …

… in languages that <u>choose</u> to make it slower!

https://www.interviewkickstart.com/learn/difference-between-recursion-and-iteration

Interlude

Recursion vs Iteration: In Racket

Conclusion?
Recursion is <u>not</u> **slower than iteration?**

equivalent

Racket Recursion

```
;; iterative-sum-to : Nat -> Nat
;; Sums the numbers in the interval [0, x]
(define (iterative-sum-to x result)
  (if (zero? x)
      result
      (iterative-sum-to (sub1 x) (+ x result))))
```

accumulator

```
(time (iterative-sum-to BIG-NUMBER 0))
; cpu time: 15 real time: 13 gc time: 0
```

"for" in **Racket** is just
a macro (i.e., "syntactic sugar")
for a (**tail**) **recursive function**

Racket "Iteration"

```
(time (for/sum ([x (add1 BIG-NUMBER)]) x))
; cpu time: 15 real time: 6 gc time: 0
```

Racket **for** expressions

Generic "sequence"
(i.e., a number, most data structures …)

```
(for/list ([x lst]) (add1 x))
```

⇩

```
(map add1 lst)
```

```
(for/list ([x n]) (add1 x))
```

⇩

```
(build-list n add1)
```

```
(for/list ([x lst] #:when (odd? x)) (add1 x))
```

⇩

```
(filter odd? (map add1 lst))
```

```
(for/sum ([x lst] #:when (odd? x)) (add1 x))
```

⇩

```
(foldl + 0 (filter odd? (map add1 lst)))
```

Note:
These are still expressions!
(not imperative statements)

Lots of variations!
(see docs)

Racket **for*** expressions

"nested" `for` loops

```
> (for* ([i '(1 2)]
         [j "ab"])
    (display (list i j)))
(1 a)(1 b)(2 a)(2 b)
```

```
> (for*/list ([i '(1 2)]
              [j "ab"])
    (list i j))
'((1 #\a) (1 #\b) (2 #\a) (2 #\b))
```

```
(for*/list (for
(for*/lists (id
  body-or-break
(for*/vector ma
(for*/hash (for
(for*/hasheq (f
(for*/hasheqv (
(for*/hashalw (
(for*/and (for-
(for*/or (for-c
(for*/sum (for-
(for*/product (
(for*/first (fo
(for*/last (for
(for*/fold ([ac
  body-or-break
(for*/foldr ([a
            (for
```

Lots of variations! (see docs)

Previously

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

Program
(Surface Syntax)

"eval"

Result

Q_1: Can we **catch errors** before **running a program?**

A_1: Some of them!

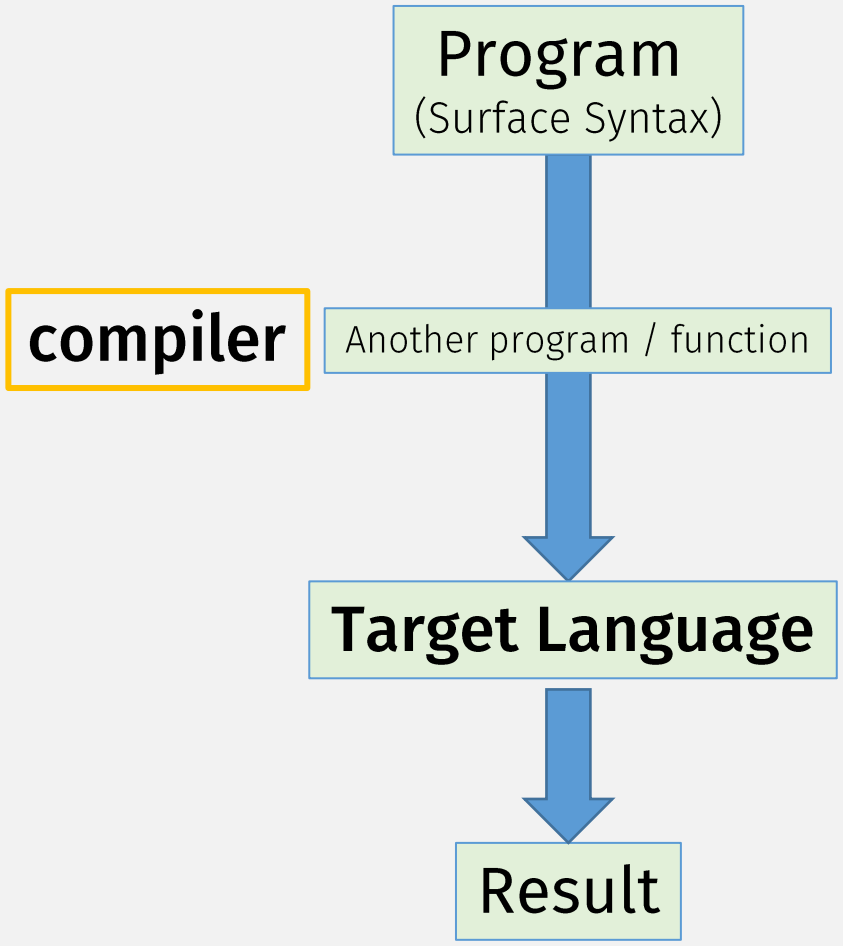
(just by "analyzing" code)

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

```
;; An ErrorResult is a:
;; - undefined-var-err
;; - not-fn-err
;; - arity-err
;; - circular-err
```

Sometimes, the **"result" of running a program** is an **Error**

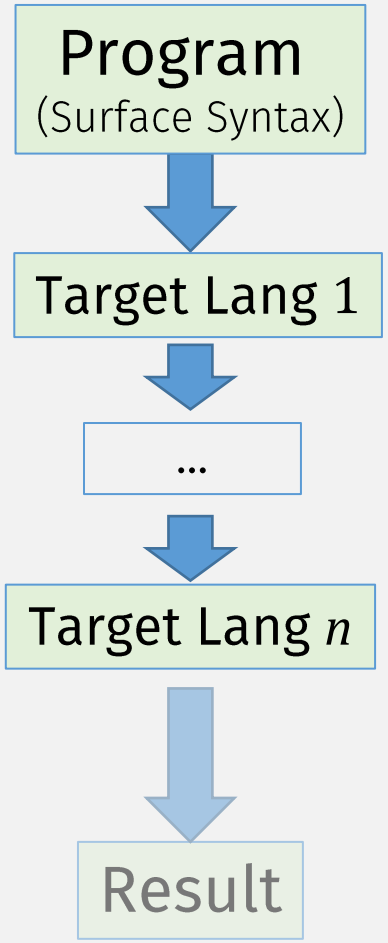
Previously



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

Previously

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

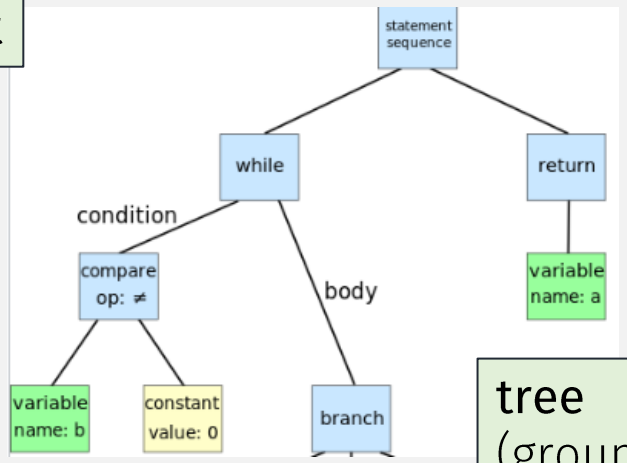
input

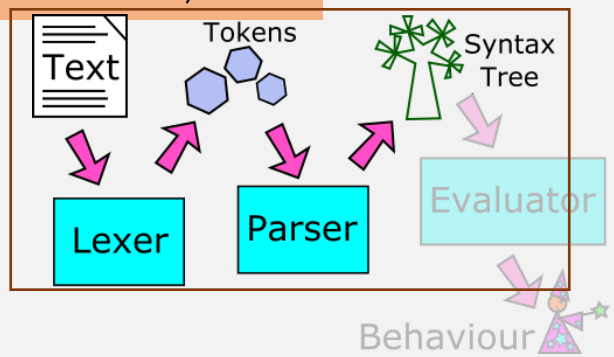
```
while b ≠ 0:
    if a > b:
        a := a - b
    else:
        b := b - a
return a
```

Typically, **string of chars**
(not a tree, not grouped by features)

Potentially really complicated
(take a compiler's course)

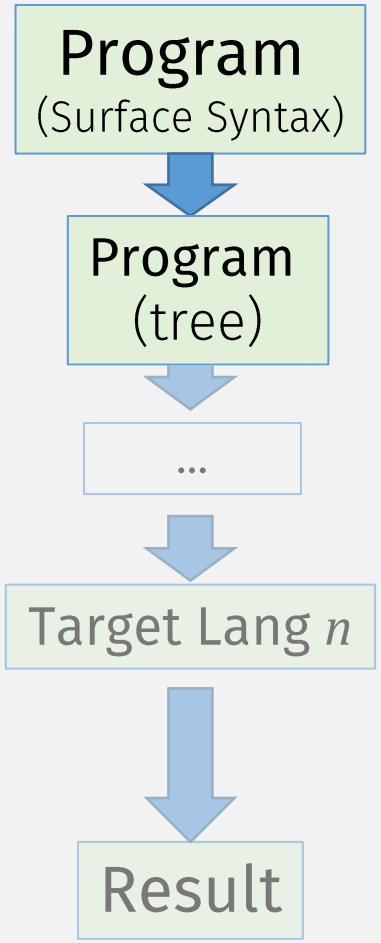
output

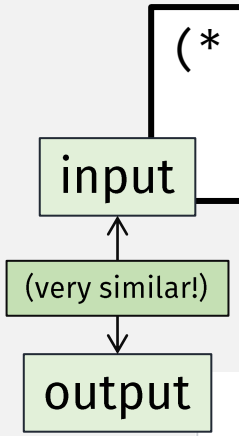




tree
(grouped by features)

Parsing – This Course



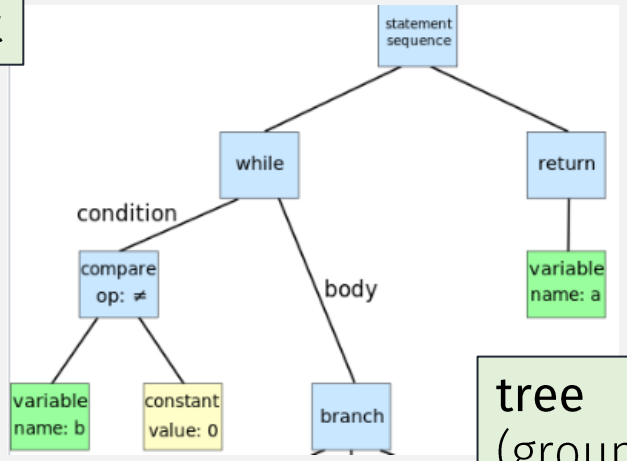



```
(* (+ 1 2)
   (- 4 3)
   (/ 10 5))
```

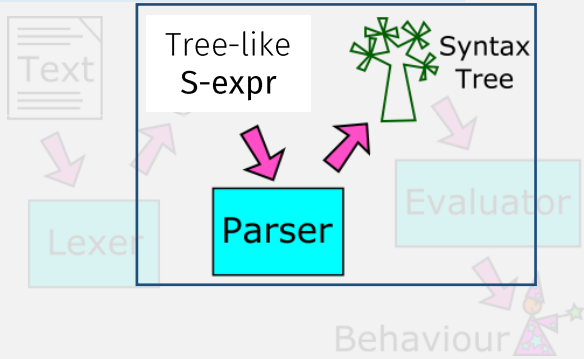
This course: **S-expressions**
(already tree-like and grouped by features)

Data definition?

Much simpler
(save hard stuff for compilers course)



tree
(grouped by features)



```
;; A Program (Sexpr) is a:
;; - Number
;; - `(+ ,Program ,Program)
;; - `(* ,Program ,Program)
```

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

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

Data definition?

```
(define (prog-fn p)
  (match p
    [(? number?) … ]
    [`(+ ,x ,y)
      … (prog-fn x) … (prog-fn y) … ]
    [`(* ,x ,y)
      … (prog-fn x) … (prog-fn y) … ]))
```

TEMPLATE?

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

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

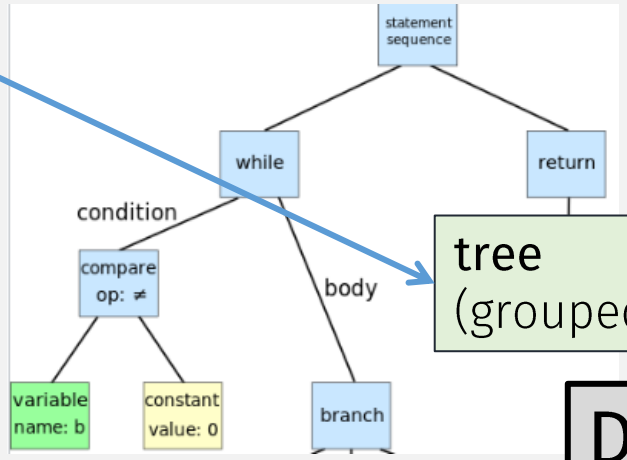
Program
(Surface Syntax)

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

...

Target Lang n

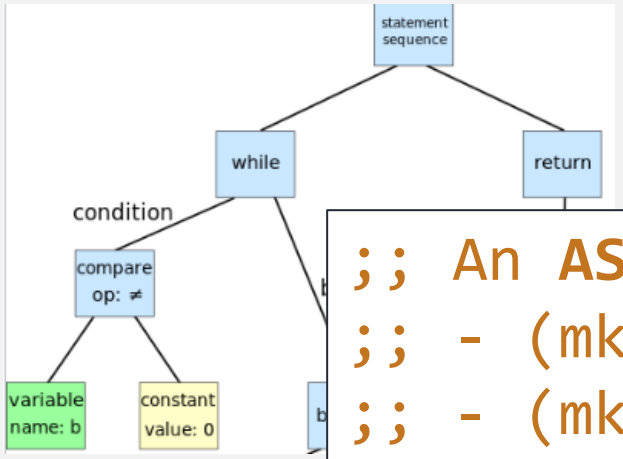
Result



tree
(grouped by features)

Data definition?

Abstract Syntax
(Program) **Tree** (**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])
```

Program that does this is called

Parser

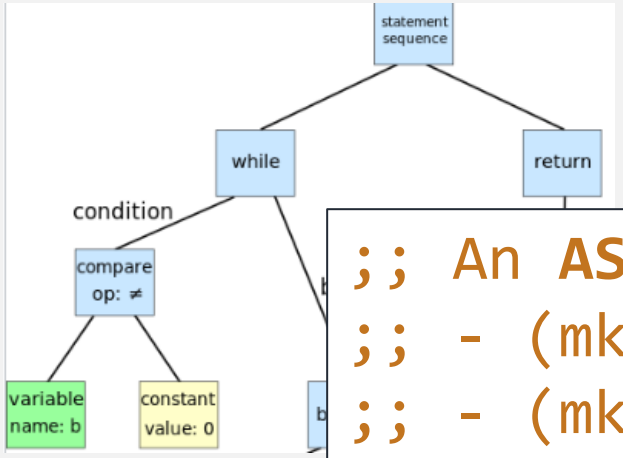
(compiler's first step)

Program

(Surface Syntax)

Abstract Syntax

(Program) **Tree (AST)**



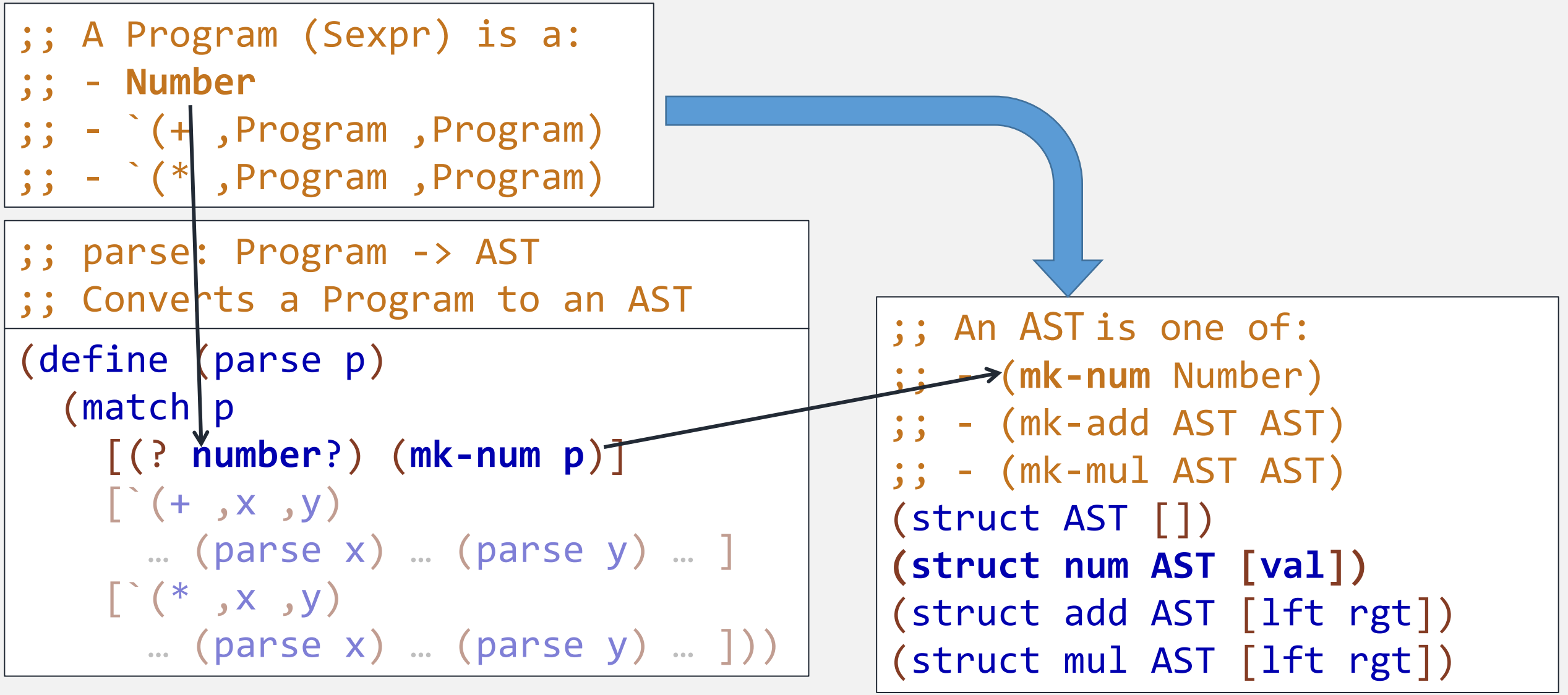

```
;; An AST is one of:
;; - (mk-num Number)
;; - (mk-add AST AST)
;; - (mk-mul AST AST)
(struct AST [])
(struct num AST [val])
(struct add AST [lft rgt])
(struct mul AST [lft rgt])
```

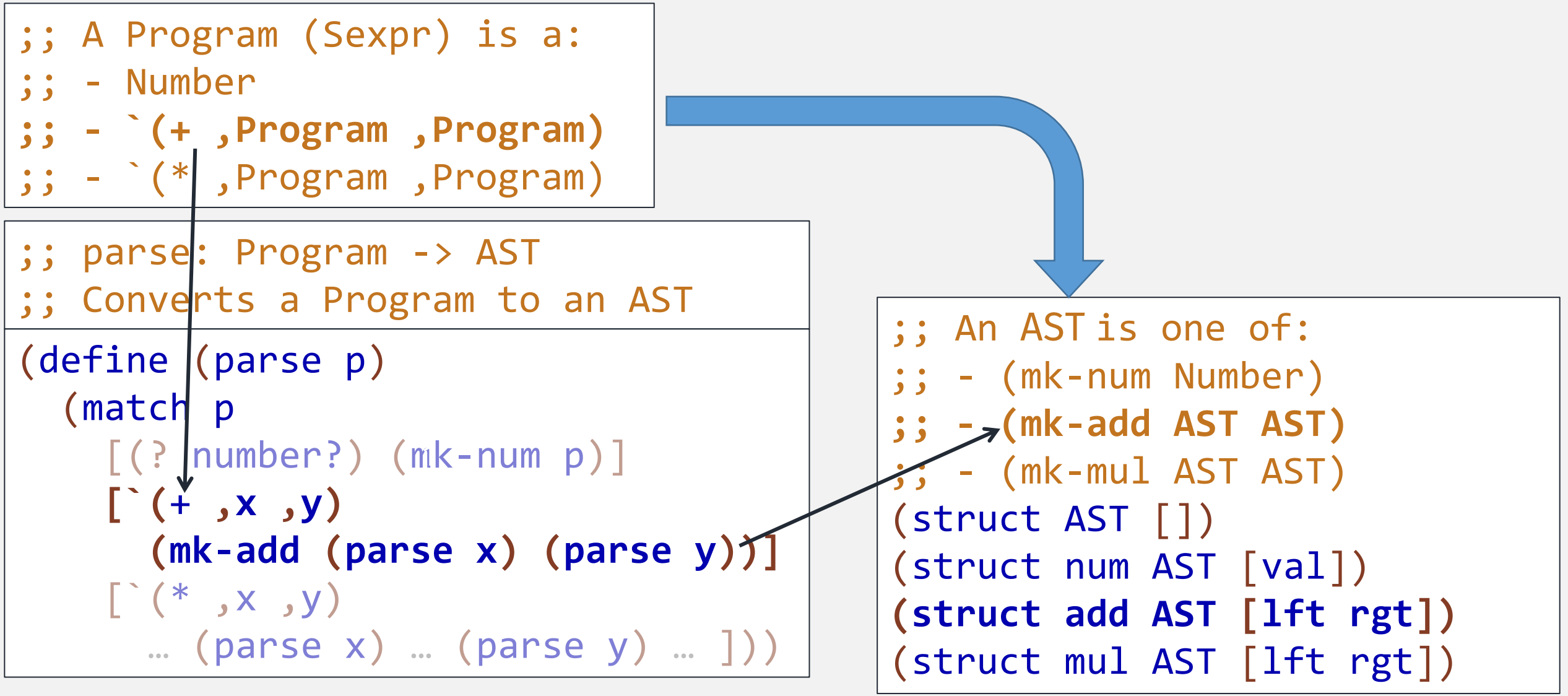
```
;; A Program (Sexpr) is a:
;; - Number
;; - `(+ ,Program ,Program)
;; - `(* ,Program ,Program)
```

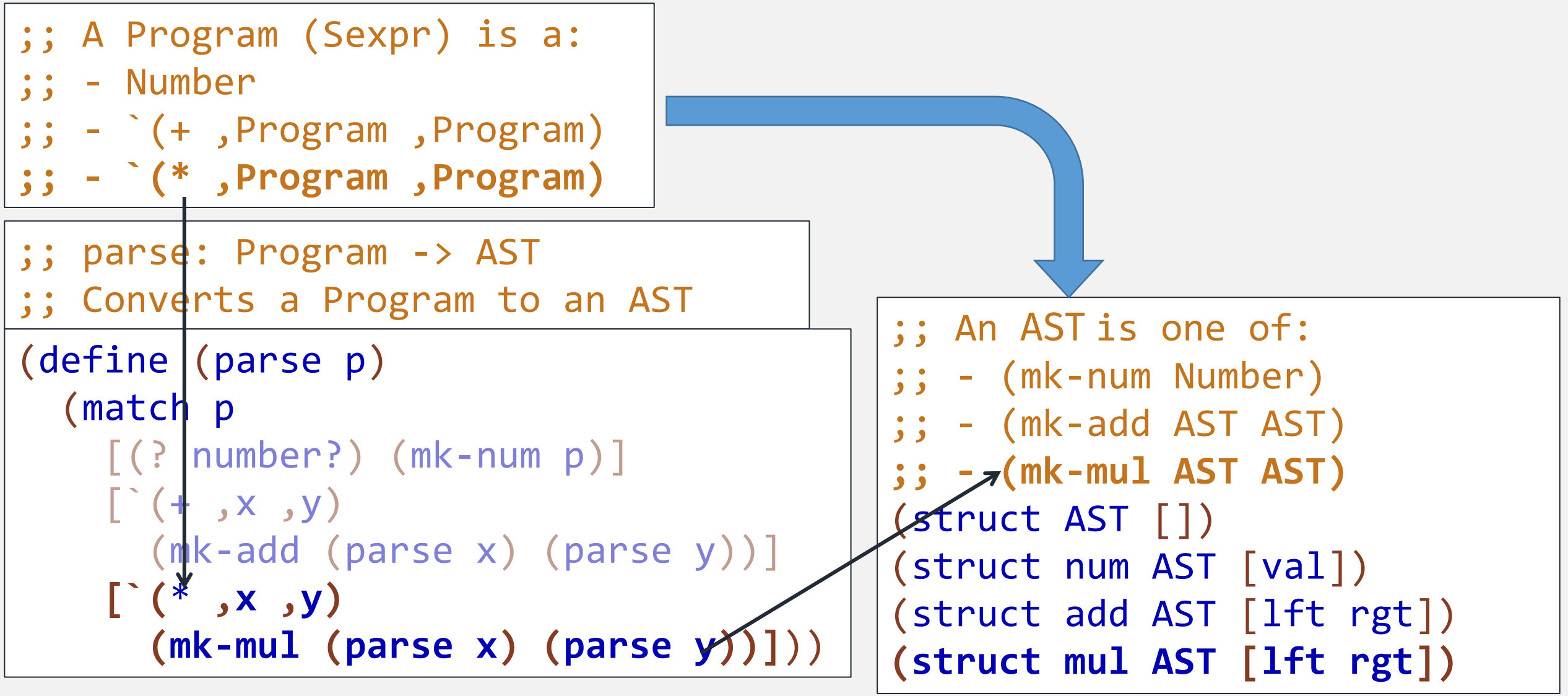
```
;; An AST is one of:
;; - (mk-num Number)
;; - (mk-add AST AST)
;; - (mk-mul AST AST)
(struct AST [])
(struct num AST [val])
(struct add AST [lft rgt])
(struct mul AST [lft rgt])
```

```
;; 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


;; A Program (Sexpr) 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 (Sexpr) 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 (Sexpr) 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])

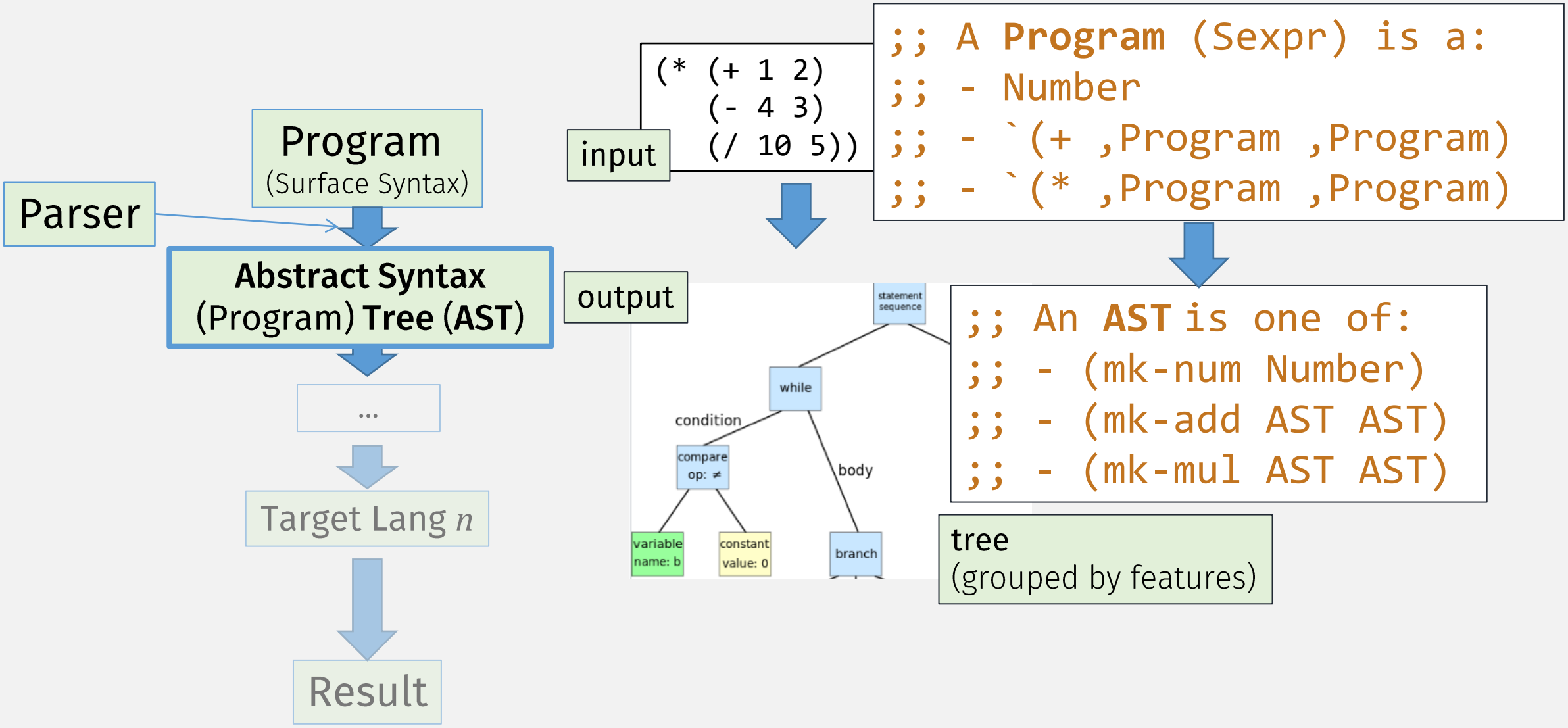
```
;; A Program (Sexpr) is a:
;; - Number
;; - `(+ ,Program ,Program)
;; - `(* ,Program ,Program)
```

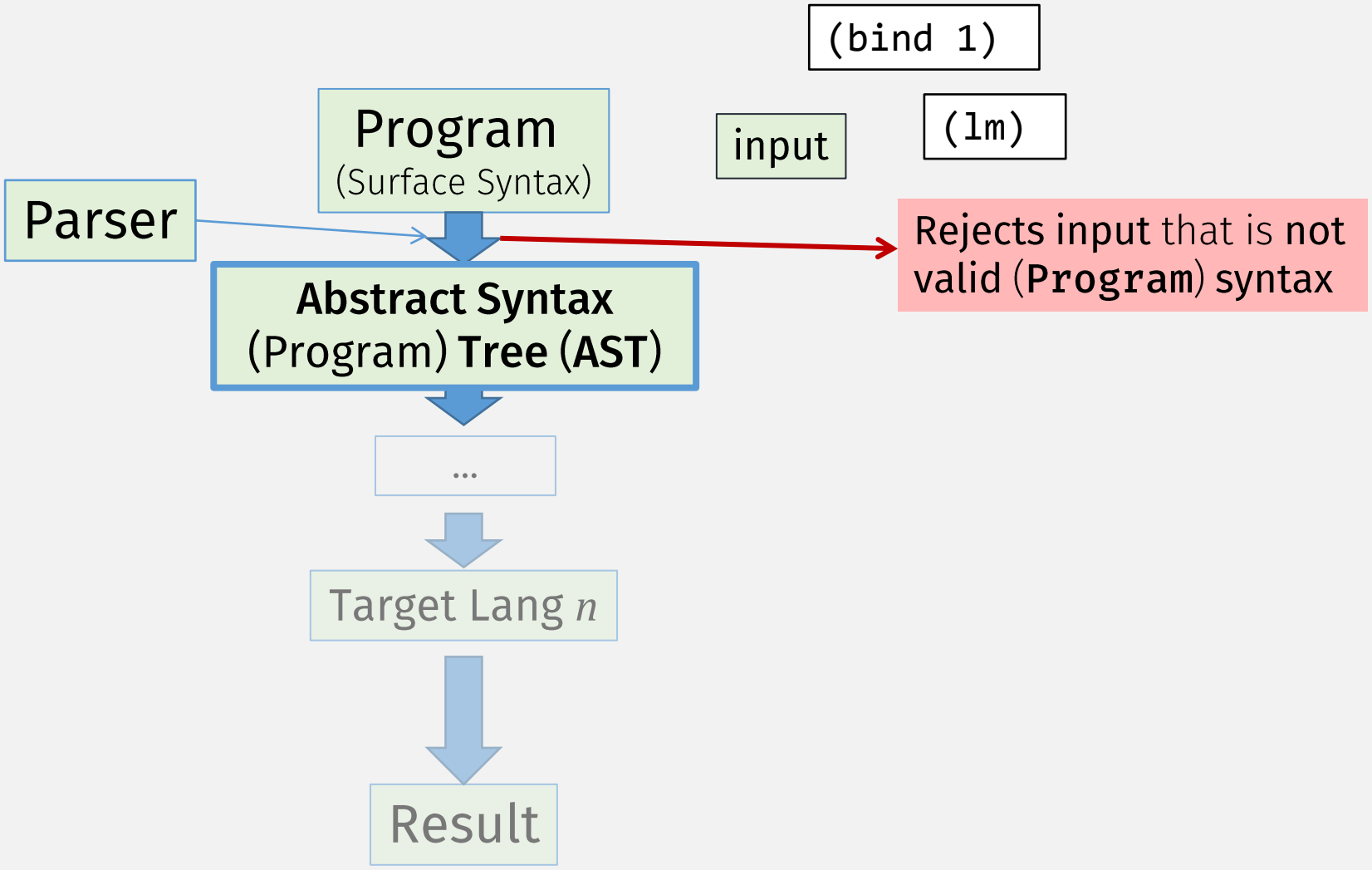
```
;; parse: Program -> AST
;; Converts a Program to an AST
```

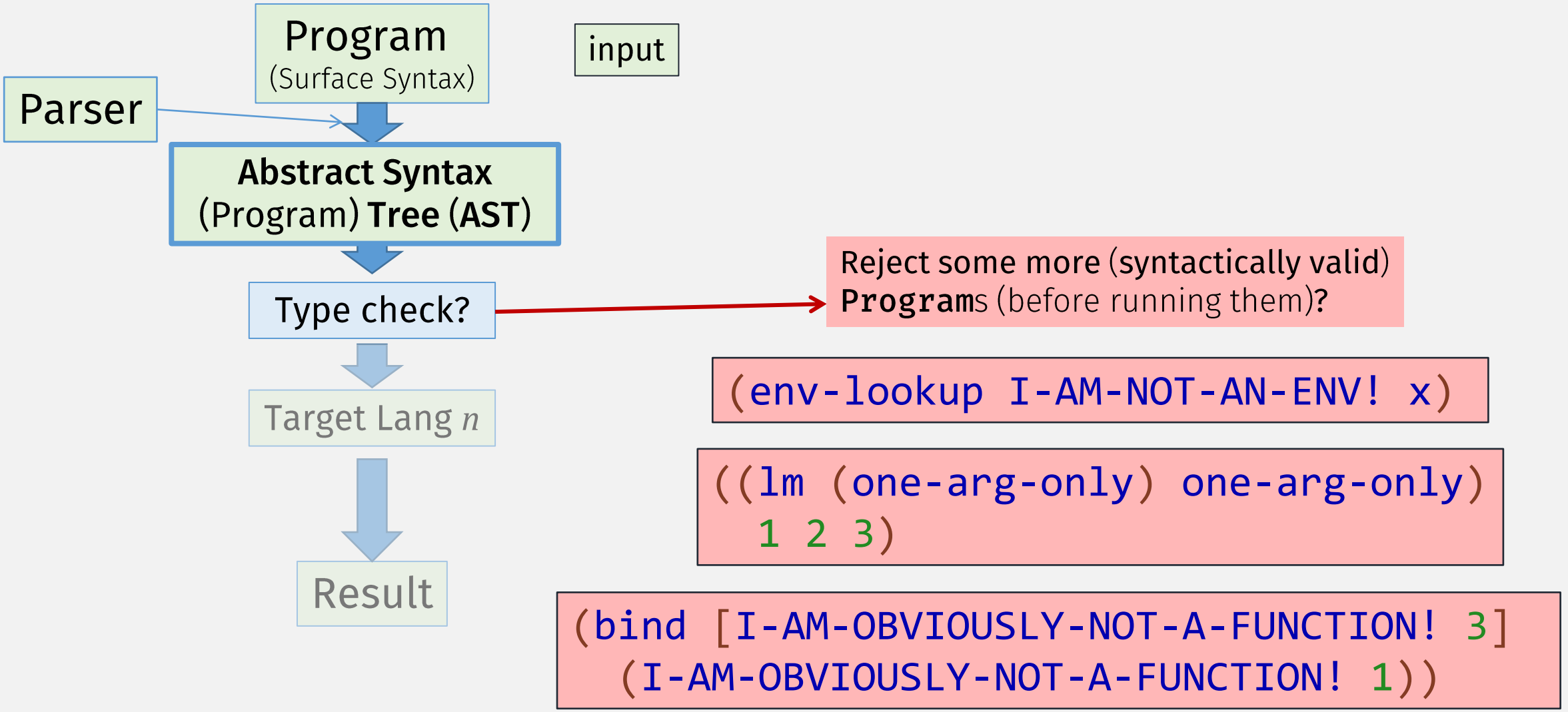
```
(define (parse p)
  (match p
    [(? number?) (mk-num p)]
    [`(+ ,x ,y)
      (mk-add (parse x) (parse y))]
    [`(* ,x ,y)
      (mk-mul (parse x) (parse y))]))
```

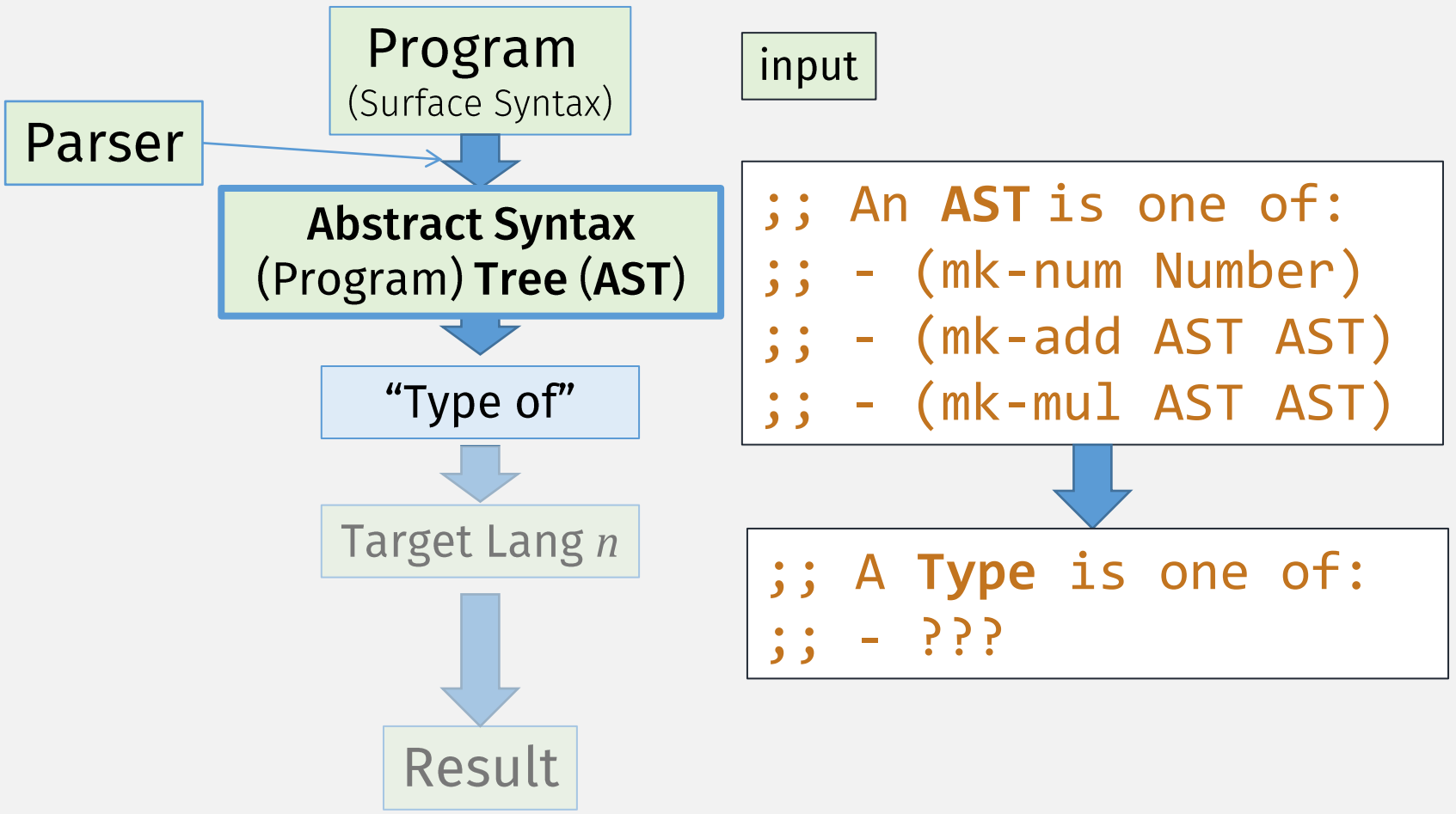
TEMPLATE MAKES THIS EASY!

```
;; 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])
```


Program
(Surface Syntax)
Parser
Abstract Syntax
(Program) Tree (AST)
...
Target Lang n
Result
input
(* (+ 1 2)
(- 4 3)
(/ 10 5))
;; A Program (Sexpr) is a:
;; - Number
;; - `(+ ,Program ,Program)
;; - `(* ,Program ,Program)
output
statement sequence
while
condition
compare op: ≠
body
variable name: b
constant value: 0
branch
;; An AST is one of:
;; - (mk-num Number)
;; - (mk-add AST AST)
;; - (mk-mul AST AST)
tree
(grouped by features)


(bind 1)
input
(lm)
Program
(Surface Syntax)
Parser
Rejects input that is not valid (Program) syntax
Abstract Syntax
(Program) Tree (AST)
...
Target Lang n
Result

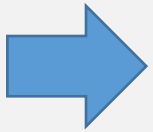

Program
(Surface Syntax)
input
Parser
Abstract Syntax
(Program) Tree (AST)
Type check?
Reject some more (syntactically valid)
Programs (before running them)?
Target Lang n
Result
(env-lookup I-AM-NOT-AN-ENV! x)
((lm (one-arg-only) one-arg-only)
1 2 3)
(bind [I-AM-OBVIOUSLY-NOT-A-FUNCTION! 3]
(I-AM-OBVIOUSLY-NOT-A-FUNCTION! 1))



Data definition?

Previously

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



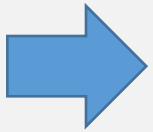
```
;; A Result is one of:
;; - ???
```

```
;; run : AST -> Result
;; Computes result of running 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) … ]))
```

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



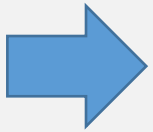
```
;; A Type is one of:
;; - ???
```

```
;; typeof : AST -> Type
;; Computes the Type of the given program
```

TEMPLATE

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

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

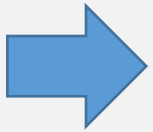


```
;; A Type is one of:
(struct Type [])
(struct Int Type [])
```

```
;; typeof : AST -> Type
;; Computes the Type of the given program
```

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

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



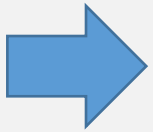
```
;; A Type is one of:
(struct Type [])
(struct Int Type [])
```

```
;; typeof : AST -> Type
;; Computes the Type of the given program
(define (typeof p)
  (match p
    [(num n) (Int)]
    [(add x y)  … (typeof x) …
                … (typeof y) …
                ?    (Int)   ?    ]
    [(mul x y)  … (typeof x) …
                …
```

What about Template Pieces?

Result should be **Int Type**

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



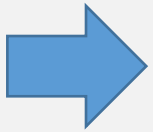
```
;; A Type is one of:
(struct Type [])
(struct Int Type [])
```

```
;; typeof : AST -> Type
;; Computes the Type of the given program
```

```
(define (typeof p)
  (match p
    [(num n) (Int)]
    [(add x y) (if (and (type=? (typeof x) (Int))
                        (type=? (typeof y) (Int)))
                   ?    (Int)    ?
                        TYPE-ERROR])
```

(hw12)

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

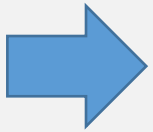


```
;; A Type is one of:
(struct Type [])
(struct Int Type [])
;; or, TYPE-ERROR (???)
```

```
;; typeof : AST -> Type
;; Computes the Type of the given program
```

```
(define (typeof p)
  (match p
    [(num n) (Int)]
    [(add x y) (if (and (type=? (typeof x) (Int))
                        (type=? (typeof y) (Int)))
                   (Int)
                   TYPE-ERROR])
```

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



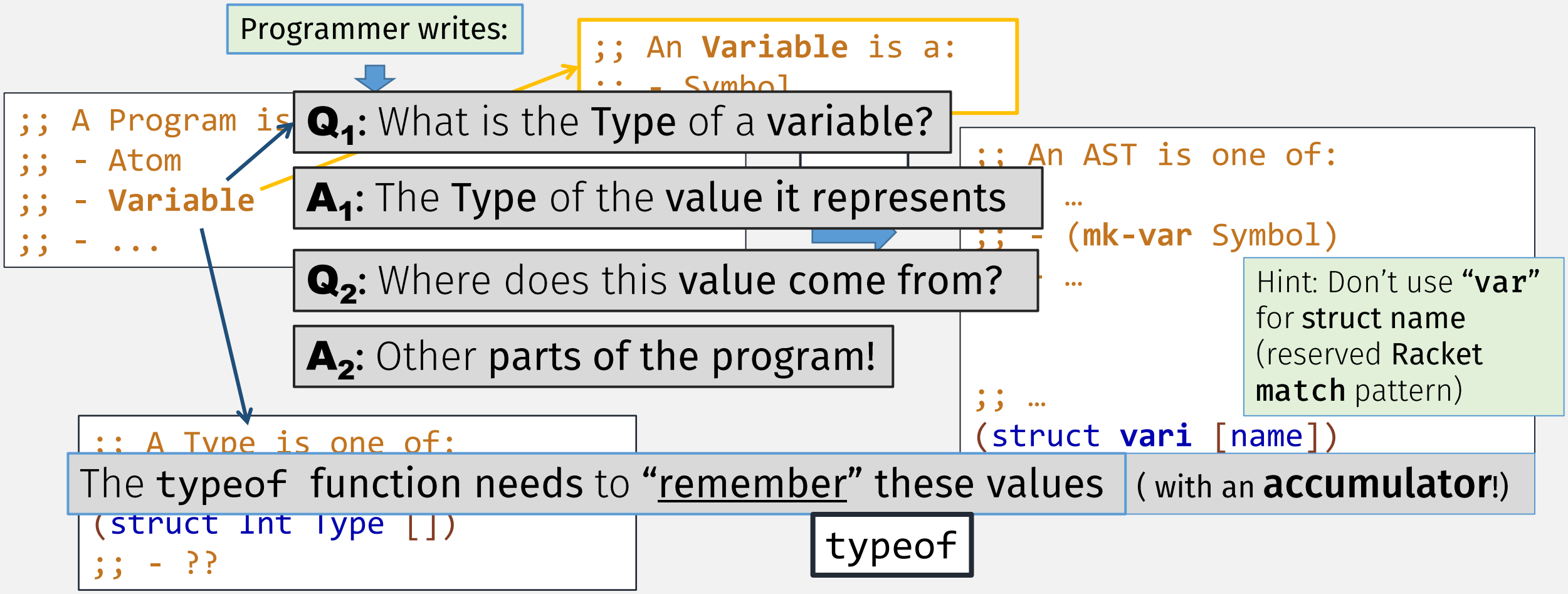
```
;; A Type is one of:
(struct Type [])
(struct Int Type [])
;; or, TYPE-ERROR (?)
```

```
;; typeof : AST -> Type
;; Computes the Type of the given program
```

```
(define (typeof p)
  (match p
    [(num n) (Int)]
    [(add x y) (if .... (type=? (typeof x) (Int)) ....
                   ;; else  ....
                   (raise-syntax-error 'typeof
                    "type mismatch, expected Int, got ~a" (typeof x)
                    #:exn exn:fail:typecheck:cs450)])
```

Still sort of a "**syntax time**", i.e., **static**, **error** (because program not run yet!)

Adding Variables



`typeof`, with an accumulator

```
;; typeof: AST -> Type
;; Computes type of a CS450 Lang Program
(define (typeof p)
  ;; accumulator acc : TypeEnvironment
  ;; invariant: Contains types of variables … currently in-scope
  (define (typeof/acc p acc)
    (match p
      [(num n) ....]
      [(vari x) ....]))
 (typeof/acc p  ???  ))
```

Type Environments

- A **data structure** that **“associates”** (`Var`, `Type`) **together**
  - E.g., **maps, hashes,** etc
  - For simplicity, let’s use **list-of-pairs**

```
;; An TypeEnvironment is one of:
;; - empty
;; - (cons (list Var Type) TypeEnvironment)
```

```
;; Represents: a type environment for
;; cs450-lang variables
```

```
;; if there are duplicates,
;; vars at front of list shadow those in back
```

Type Environments

- A **data structure** that **"associates"** (`Var`, `Type`) **together**
  - E.g., **`maps`, `hashes`,** etc
  - For simplicity, let's use **list-of-pairs**

```
;; An TypeEnvironment (TyEnv) is one of:
;; - empty
;; - (cons (list Var Type) TypeEnvironment)
```

- Needed operations:
  - `tyenv-add    : TyEnv Var Type -> TyEnv`
  - `tyenv-lookup : TyEnv Var -> Type`

Should throw "typecheck" exn, if var not found

(don't want to "run" the program)

`typeof`, with a Type Environment accumulator

```
;; typeof : AST -> Type
(define (typeof p)
  ;; accumulator tyenv : TypeEnvironment
  ;; invariant: contains in-scope Vars + their Types
  (define (typeof/env p tyenv)
    (match p
      …
      [(vari x) (tyenv-lookup tyenv x)]

      …   ))
 (typeof/env p  ???  ))
```

`typeof`, with a Type Environment accumulator

```
;; typeof : AST -> Type
(define (typeof p)
  ;; accumulator tyenv : TypeEnvironment
  ;; invariant: contains in-scope Vars + their Types
  (define (typeof/env p tyenv)
    (match p
      …
      [(vari x) (tyenv-lookup tyenv x)]
      [(bind x e body) … (tyenv-add tyenv x (run/env e tyenv)) …]
      …   ))
 (typeof/env p  ???  ))
```

Don't name this `var`

`typeof`, with a Type Environment accumulator

TODO:
- When are variables "added" to environment???
- What is initial environment?

```
;; typeof : AST -> Type
(define (typeof p)
  ;; accumulator tyenv : TypeEnvironment
  ;; invariant: contains in-scope Vars + their Types
  (define (typeof/env p tyenv)
    (match p
      …
      [(vari x) (tyenv-lookup tyenv x)]
      [(bind ??? body) … (tyenv-add tyenv x (typeof/env e tyenv)) …]
      …  ))
 (typeof/env p ???  ))
```

Programs that Add Variables to Environment

```
;; A Program is one of:
;; - Atom
;; - Variable (Var)
;; - ?????
```

Programs that Add Variables to Environment

```
;; A Program is one of:
;; - Atom
;; - Variable (Var)
;; - `(bind [,Var ,Program] ,Program)
;; - ...
```

(like "**let**" in other langs)

`typeof`, with **`bind`**

```
;; typeof: AST -> Type
(define (typeof p)
  ;; accumulator tyenv : TypeEnvironment
  ;; invariant: contains in-scope Vars + their Types
                                    p tyenv)
; An AST is one of:
; - …
; - (mk-bind Symbol AST AST)
      [(vari x) (tyenv-lookup tyenv x)]
      [(bind x e body) … (tyenv-add tyenv x (typeof/env e tyenv)) …]
       …   ))
 (typeof/env p  ???  ))
```

```
;; An TypeEnvironment (TyEnv) is one
of:
;; - empty
;; - (cons (list Var Type) Env)
```

TypeEnvironment has **Type**s (not **AST**)

How to convert **AST** to **Type**?

(From template!)

Add to environment

Be careful to get correct "**scoping**"
(x not visible in expression e,
so use unmodified input **tyenv**)

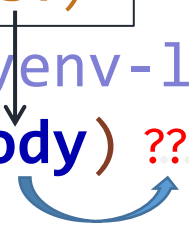
`typeof`, with **`bind`**

`typeof` must produce `Type`

```
;; typeof: AST -> Type
(define (typeof p)
  ;; accumulator tyenv : TypeEnvironment
  ;; invariant: contains in-scope Vars + their Types
                                    p tyenv)

; An AST is one of:
; - …
; - (mk-bind Symbol AST AST)

      [(vari x) (tyenv-lookup tyenv x)]
      [(bind x e body) ??? (tyenv-add tyenv x (typeof/env e tyenv)) …]
       …   ))
 (typeof/env p  ???  ))
```



`typeof`, with **`bind`**

```
;; typeof: AST -> Type
(define (typeof p)
  ;; accumulator tyenv : TypeEnvironment
  ;; invariant: contains in-scope Vars + their Types
  (define (typeof/env p tyenv)
    (match p
      …
     [(vari x) (tyenv-lookup tyenv x)]
     [(bind x e body) (typeof/env body (tyenv-add tyenv x (typeof/env
      …   ))
 (typeof/env p  ???  ))
```

(From template!

Typeof body with new env containing **x**

Initial Environment?

TODO:
- ~~When are variables "added" to environment~~
- What is **initial environment?** `empty` (for now)

```
;; typeof: AST -> Type
(define (typeof p)
  ;; accumulator tyenv : TypeEnvironment
  ;; invariant: contains in-scope Vars + their Types
  (define (typeof/env p tyenv)
    (match p
      …
      [(vari x) (tyenv-lookup tyenv x)]
      [(bind x e body) (typeof/env body (tyenv-add tyenv x (typeof/env
      …   ))
 (typeof/env p ???  ))
```

`empty ???` (for now)

Initial Environment

```
;; A Program is one of:
;; - Atom
;; - Variable
;; - `(bind [,Var ,Program] ,Program)
;; - `(+ ,Program ,Program)
;; - `(* ,Program ,Program)
```

These **don't** need to be **separate constructs!**

Put these into **"initial" environment**

Initial Environment

```
;; A Program is one of:
;; - Atom
;; - Variable
;; - `(bind [,Var ,Program] ,Program)
;; - `(+ ,Program ,Program)
;; - `(* ,Program ,Program)
```

Put these into "initial" environment

```
;; An TypeEnv is one of:
;; - empty
;; - (cons (list Var Type) TypeEnv)
```

Need new kind of Type

```
(define INIT-TYENV
`((+ ,(mk-FnTy (Int) (Int) (Int)))
  (* ,(mk-FnTy (Int) (Int) (Int)))))
```

+ variable

```
;; A Type is one of:
(struct Type [])
(struct Int Type [])
;; - (mk-FnTy Type … Type)
(struct Fn Type [tins tout])
```

Initial Environment

But … how to type check **function applications??**

```
(define INIT-TYENV `((+ ,(mk-FnTy (Int) (Int) (Int)))
                     (* ,(mk-FnTy (Int) (Int) (Int)))))
```

```
(define (typeof p)

  ;; accumulator env : TypeEnvironment
  (define (typeof/e p env)
    (match p
      …
      [(vari x) (tyenv-lookup env x)]
      [(bind x e body) (typeof/e body (tyenv-add env x (typeof/e e env)
      … ))
 (typeof/e p INIT-TYENV ))
```

Type Checking Function Application

```
;; A Program is one of:
;; - Atom
;; - Variable
;; - `(bind [,Var ,Program] ,Program)
;; - (cons Program List<Program>)
```

parse

```
;; An AST is one of:
;; - …
;; - (mk-var Symbol)
;; - (mk-bind Symbol AST AST)
;; - (mk-appl AST List<AST>)

(struct vari [name])
(struct bind [var expr body])
(struct appl [fn args])
```

Type Checking Function Application

TEMPLATE: extract pieces of compound data

```
;; An AST is one of:
;; - …
;; - (mk-var Symbol)
;; - (mk-bind Symbol AST AST)
;; - (mk-appl AST List<AST>)

(struct vari [name])
(struct bind [var expr body])
(struct appl [fn args])
```

```
(define (typeof p)


  (define (typ/e p env)
    (match p
           …
      [(appl fn args) (typecheck-apply
                       (typ/e fn env)
                       (map (curryr typ/e env) args))]

           …
      ))
  (typ/e p INIT-TYENV))
```

Type Checking Function Application

```
;; An AST is one of:
;; - …
;; - (mk-var Symbol)
;; - (mk-bind Symbol AST AST)
;; - (mk-appl AST List<AST>)
```

```
(define (typeof p)


  (define (typ/e p env)
    (match p
           …
      [(appl fn args) (typecheck-apply
                       (typ/e fn env)
                       (map (curry typ/e env) args))]

           …
      ))
  (typ/e p INIT-TYENV))
```

TEMPLATE: recursive calls

???

Compute type of args

Type Checking Function Application

How to actually type check the application?

```
;; A Type is one of:
(struct Type [])
(struct Int Type [])
(struct Fn Type [tins tout])
```

```
(define (typeof p)


  (define (typ/e p env)
    (match p
           …
      [(appl fn args) (typeof-apply ???
                                    (typ/e fn env)
                                    (map (curryr typ/e env) args))]

           …
      ))
  (typ/e p INIT-TYENV))
```

Should only work for Fn types?

Type Checking Function Application

```
;; A Type is one of:
(struct Type [])
(struct Int Type [])
(struct Fn Type [tins tout])
```

```
(define (typeof p)

  (define (typ/e p env)
    (match p
           …
      [(appl fn args) (typeof-apply
                       (typ/e fn env)
                       (map (curryr typ/e env) args))]
           …
      ))
  (typ/e p INIT-TYENV))
```

Bad Example:

```
(typeof '(10 10)) ; apply non-fn
```

Type Checking Function Application

```
;; A Type is one of:
(struct Type [])
(struct Int Type [])
(struct Fn Type [tins tout])
```

```
(define (typeof p)


  (define (typ/e p env)
    (match p
           …
      [(appl fn args) (typeof-apply
                        (typ/e fn env)
                        (map (curryr typ/e env) args))]
           …
      ))
  (typ/e p INIT-TYENV))
```

Type Checking Function Application

```
;; A Type is one of:
(struct Type [])
(struct Int Type [])
(struct Fn Type [tins tout])
```

```
;; typeof-apply: Type Listof<Type> -> Type
```

```
(define (typeof-apply fntype argtypes)
  (match fntype
   [(Int)                    ... ]
   [(Fn expected-argtypes outtype)
                                  ... ]

))
```

TEMPLATE?

Type Check fail

Type Checking Function Application

```
;; A Type is one of:
(struct Type [])
(struct Int Type [])
(struct Fn Type [tins tout])
```

```
;; typeof-apply: Type Listof<Type> -> Type
```

```
(define (typeof-apply fntype argtypes)
  (match fntype
   [(Fn expected-argtypes outtype)
    (if
          (andmap type=? expected-argtypes argtypes)
        outtype
        (raise-type-exn ...) )]))
```

Check correct number of arguments???

If input arg types match the expected …

Type Checking Function Application

```
;; A Type is one of:
(struct Type [])
(struct Int Type [])
(struct Fn Type [tins tout])
```

```
;; typeof-apply: Type Listof<Type> -> Type
```

```
(define (typeof-apply fntype argtypes)
  (match fntype
   [(Fn expected-argtypes outtype)
    (if (and (= (length expected-argtypes) (length argtypes))
             (andmap type=? expected-argtypes argtypes))
        outtype
        (raise-type-exn ....) )]))
```

Check correct number of arguments???

Type Checking CS450 Lang

```
;; A Program is one of:
;; - Atom
;; - Variable
;; - `(bind [,Var ,Program] ,Program)
;; - (cons Program List<Program>)
```

What if a user wants to define their own function!

Next Feature: **Lambdas?**

“Lambdas” in CS450 Lang

```
;; A Program is one of:
;; - Atom
;; - Variable (Var)
;; - `(bind [,Var ,Program] ,Program)
;; - `(lm ,List<Var> ,Program)
;; - (cons Program List<Program>)
```

CS450 Lang “Lambda” examples

```
;; A Program is one of:
;; - Atom
;; - Variable (Var)
;; - `(bind [,Var ,Program] ,Program)
;; - `(lm ,List<Var> ,Program)
;; - (cons Program List<Program>)
```

CS450LANG

```
(lm (x) (+ x 1))
```

Type of x? `Int`

```
(lm (x) x)
```

Type of x? **???**

Programmer needs to give it!

Typed CS450 Lang “Typed Lambda” examples

CS450LANG

```
(lm [x : Int] (+ x 1))
```

Type of x? `Int`

```
;; A Program is one of:
;; - Atom
;; - Variable (Var)
;; - `(bind [,Var ,Program] ,Program)
;; - `(lm [,Var : ,Type] ,Program)
;; - (cons Program List<Program>)
```

(single-arity for now)

```
(lm [x : Int] x)
```

Type of x? `Int`

Programmer needs to give it!

Typed CS450 Lang – parsing “Typed Lambda”

```
;; A Program is one of:
;; - Atom
;; - Variable (Var)
;; - `(bind [,Var ,Program] ,Program)
;; - `(lm [,Var : ,Type] ,Program)
;; - (cons Program List<Program>)
```

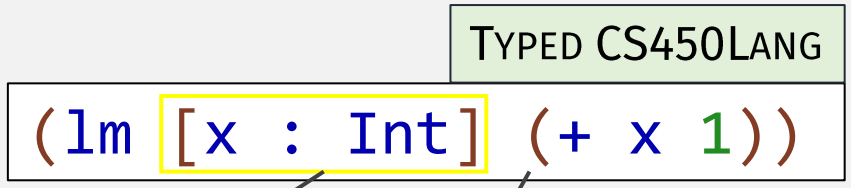
parse

```
;; An AST is one of:
;; …
;; - (mk-tylm-ast Var Type AST)
;; …



(struct tylm-ast [x ty body])
;; …
```

Parsing “Typed Lambda”




```
(define/contract (parse p)
  (-> Program? AST?)
  (match p
        ...
   [`(lm [,(? symbol? x) : ,ty] ,bod) ... ]

   [`(,fn . ,args) ... ]
   [_ (raise-syntax-error
        'parse "not a valid CS450 Lang program" p
       #:exn exn:fail:syntax:cs450)]))
```

Pattern matching
correct typed `lm` syntax

Parsing “Typed Lambda”

Typed CS450Lang

```
(lm [x : Int] (+ x 1))
```

```
(define/contract (parse p)
  (-> Program? AST?)
  (match p
        ...
   [`(lm [,(? symbol? x) : ,ty] ,bod)
    (mk-tylm-AST x (parse ??type ty) (parse bod))   ]


   [`(,fn . ,args) ... ]
   [_ (raise-syntax-error
        'parse "not a valid CS450 Lang program" p
       #:exn exn:fail:syntax:cs450)]))
```

```
;; An AST is one of:
;; …
;; - (mk-lm-ast Var Type AST)
;; …
```

not an AST!

Recursive call for ASTs

Parsing “Type Syntax”? (what the programmer writes)

```
;; TypeSyntax is:
;; - 'Int
```

```
;; A Type is one of:
;; (mk-Int)
;; (mk-FnTy Type … Type)
```

```
(define/contract (parse-type ty)
  (-> TypeSyntax? Type?)
  (match ty
        ...
   ['Int ... ]
```

```
))
```

Parsing “Type Syntax”? (what the programmer writes)

```
;; TypeSyntax is:
;; - 'Int
```

```
;; A Type is one of:
;; (mk-Int)
;; (mk-FnTy Type … Type)
```

```
(define/contract (parse-type ty)
  (-> TypeSyntax? Type?)
  (match ty
        ...
   ['Int (mk-Int)]
                                        ))
```

Parsing “Type Syntax”? (what the programmer writes)

```
;; TypeSyntax is:
;; - 'Int
;; - '(-> TypeSyntax …)
```

```
;; A Type is one of:
;; (mk-Int)
;; (mk-FnTy Type … Type)
```

```
(define/contract (parse-type ty)
  (-> TypeSyntax? Type?)
  (match ty
        ...
   ['Int (mk-Int)]
```

```
))
```

Parsing “Type Syntax”? (what the programmer writes)

```
;; TypeSyntax is:
;; - 'Int
;; - '(-> TypeSyntax …)
```

```
;; A Type is one of:
;; (mk-Int)
;; (mk-FnTy Type … Type)
```

```
(define/contract (parse-type ty)
  (-> TypeSyntax? Type?)
  (match ty
        ...
   ['Int (mk-Int)]
   [`(-> ,tyins … ,tyout) ... ]
                                  ))
```

Parsing “Type Syntax”? (what the programmer writes)

```
;; TypeSyntax is:
;; - 'Int
;; - '(-> TypeSyntax …)
```

```
;; A Type is one of:
;; (mk-Int)
;; (mk-FnTy Type … Type)
```

```
(define/contract (parse-type ty)
  (-> TypeSyntax? Type?)
  (match ty
        ...
   ['Int (mk-Int)]
   [`(-> ,tyins … ,tyout) (mk-FnTy tyins tyout)]
                                              ))
```

Typed CS450 Lang – type of Typed Lambda

```
;; A Program is one of:
;; - Atom
;; - Variable (Var)
;; - `(bind [,Var ,Program] ,Program)
;; - `(lm [,Var : ,Type] ,Program)
;; - (cons Program List<Program>)
```

parse

```
;; An AST is one of:
;; …
;; - (mk-tylm-ast Var Type AST)
;; …


(struct tylm-ast [x ty body])
;; …
```

typeof

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

Typed CS450 Lang – type of Typed Lambda

```
;; A Program is one of:
;; - Atom
;; - Variable (Var)
;; - `(bind [,Var ,Program] ,Program)
;; - `(lm [,Var : ,Type] ,Program)
;; - (cons Program List<Program>)
```

parse

```
;; An AST is one of:
;; …
;; - (mk-tylm-ast Var Type AST)
;; …


(struct tylm-ast [x ty body])
;; …
```

typeof

```
;; A Type is one of:
;; (mk-Int)
;; (mk-FnTy Type … Type)
```

Typeof Lambdas

```
;; typeof: AST -> Type
(define (typeof p)


  (define (typ/e p env)
    (match p
          …
      [(tylm-ast x ty body) ??          ??                    ??]
          …
      ))
  (typ/e p INIT-TYENV))
```

TEMPLATE?

```
;; An AST is one of:
;; …
;; - (mk-tylm-ast Var Type AST)
;; …
(struct tylm-ast [x ty body])
```

Typeof Lambdas

```
;; typeof: AST -> Type
(define (typeof p)


  (define (typ/e p env)
    (match p
          …
      [(tylm-ast x ty body) ??  x ty ?? (typ/e body env) ??]
          …
      ))
  (typ/e p INIT-TYENV))
```

TEMPLATE

```
;; An AST is one of:
;; …
;; - (mk-tylm-ast Var Type AST)
;; …
(struct tylm-ast [x ty body])
```

Typeof Lambdas

```
;; typeof: AST -> Type
(define (typeof p)


  (define (typ/e p env)
    (match p
          …
      [(tylm-ast x ty body) ??  x ty  ?? (typ/e body env) ??]
      ))
  (typ/e p INIT-TYENV))
```

What is the **Type** of a **tylm** function?

```
;; An AST is one of:
;; …
;; - (mk-tylm-ast Var Type AST)
;; …
(struct tylm-ast [x ty body])
```

```
;; A Type is one of:
;; (mk-Int)
;; (mk-FnTy Type … Type)
```

Typeof Lambdas

```
;; typeof: AST -> Type
(define (typeof p)


  (define (typ/e p env)
    (match p
           …
      [(tylm-ast x ty body) (mk-FnTy ty (typ/e body (env-add env x ty)))]
           …
      ))
  (typ/e p INIT-TYENV))
```

Type of **body** is function output type

Can't compute unless **x** is in type env!

Type of x is function input type

```
;; A Type is one of:
;; (mk-Int)
;; (mk-FnTy Type … Type)
```

Type Checking Function Application

Still works for typed `lm`!

```
;; A Type is one of:
(struct Type [])
(struct Int Type [])
(struct Fn Type [tins tout])
```

```
;; typeof-apply: Type Listof<Type> -> Type
```

```
(define (typeof-apply fntype argtypes)
  (match fntype
   [(Fn expected-argtypes outtype)
    (if (and (= (length expected-argtypes) (length argtypes))
             (andmap type=? expected-argtypes argtypes))
        outtype
        (raise-type-exn ....) )]))
```

Next: Recursion with Types?