

## Summary

- ASTs are abstract representation of a program
- There is way to theoretically define the abstract syntax by looking at the concrete syntax
- Now, we are defining a language with *only addition*. So, we also show how to define this AST in racket (using `define-datatype`), and then write an unparser to go from AST to concrete syntax and then a parser to go from concrete syntax to AST

**Agenda:** Introducing ASTs by defining what they are, and how to show them in racket.

- We will be defining our first (very trivial) language
- Will come around to (eventually) Abstract Syntax Trees
- First, a few questions about programs:
  1. What is a program? How is it represented? `<--` will tackle this today
  2. What does it mean? How does it run?

## What is a program

Say you have a python program:

```
def f(x):
    return x+2
```

- It's a sequence of characters, a sequence of tokens
- However, we are more concerned with abstract representations

## A more abstract representation of a program

- **Trees** are an abstract representation. We will be using that.
- We start with a simple language: addition

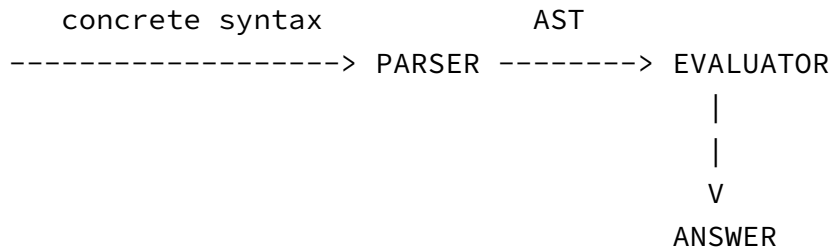
- It's representation of something like `2 + 3` is

```

      +
     / \
    2   3
  
```

Abstract syntax tree : this is what it is represented internally

Equivalent concrete structures: `2 + 3`  
`+ 2 3`



### From concrete to abstract

We'll work with a very simple language, *addition*.

Say we have this concrete syntax:

```

<exp> ::= <num> |          \_ prefix
        '(+ ' <exp> ' ' <exp> ' )'  /
  
```

```

<exp> ::= <num> |          \_ infix
        '(' <exp> '+' <exp> ')'  /
  
```

The abstract syntax for that language is: <sup>1</sup>

```

<ast> ::= <num> |          \_ grammar of trees
        + <ast> <ast>  /
  
```

We pass judgement on the astness of an ast using this:

```

|-----|
| if n ∈ N      | num rule
| ⇒ n AST      |
|           OR  |
| if e1 AST & e2 AST | plus rule
| ⇒ + e1 e2 AST  |
|-----|
  
```

<sup>1</sup>at around 14:00, sir talks about the AST structure as a language. This caught me off guard causing me to lose track. It was mentioned once before, but was unable to pick it up at that moment.

-----  
 | Now, the valid expressions  
 V in this language are  
 usable for rating? judgements

Judgement:  $\vdash e \text{ AST} \vdash$   
 -----

An example of the judgement:

Judgement	rating	justification
3 AST	sound	num, 3 $\in$ N
2 AST	sound	num, 2 $\in$ N
2+3 AST	sound	plus, 2 AST & 3 AST
2+ AST	unsound	not derivable

## Implementing a parser and unparser

Rest of class: we will implement regularisation of ASTs, and write two functions: parse, unparse

## Implementing ASTs in Racket

Other way to define

$\langle \text{ast} \rangle ::= \langle \text{num} \rangle \mid + \langle \text{ast} \rangle \langle \text{ast} \rangle$   
 ----- $\vee$ -----  
 |  
 $\vee$   
 Variant (sum) type  
 base case | Inductive case

Implementing it in racket:

```
> (define-datatype ast ast? ;; the second is the type predicate
  [num (n number?)]
  [plus (left ast?) (right ast?)])
```

```

> (number? 5)
#t
> symbol?
> procedure?
;; so
> (check-true (ast? (num 5)))
#t

```

num and plus get autodefined as *constructor functions* with the following signatures:

- num ::= number? → ast?
- plus ::= [ast?, ast?] → ast?

Eg:

```

> (num 5); --> an AST          num 5

;; example:
> (ast? (plus (num 5)          ;      plus
              (num 6)))        ;  /   \
                                ; num5  num6
#t

; example of more complex????
> (let ([e1 (plus (num 5) (num 6))] ;      +
      [e2 (plus (num 3) (num 3))] ;      / \
      (plus e1 e2))               ;  +   +
      ;      / \ / \
      ;      n5 n6 n2 n3

```

Now something something looking at abstract to concrete syntax

Abstract syntax —unparser→ concrete syntax ←parser—

### Unparser implementation in RACKET

```

;;; unparse : ast? ----> any/c
> (define (unparse a)
  (cases ast a
    [num (n) n]
    [plus (left right)
      (list '+
            (unparse left)
            (unparse right))]))
> (unparse (num 5))

```

5

```
> (unparse (plus (num 5) (num 4)))  
'(+ 5 4)
```

### Parser implementation in RACKET

```
;;; parse : any/c ---> ast? || error
```

```
> (define (parse d)  
  (cond [(number? d) (num d)]  
        [(and (list? d)  
              (= (length d) 3)  
              (eq? (first d) '+))  
         (plus (parse (second d))  
               (parse (third d)))]  
        [else (error 'parse "invalid syntax" d)]
```

```
> (parse 5)  
(num 5)  
> (parse '(+ 2 3))  
(plus (num 2) (num 3))
```