

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:

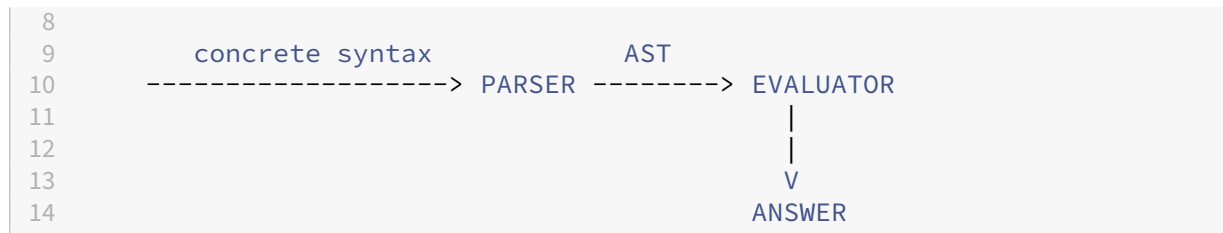
```
1 def f(x):
2     return x+2
```

- 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

```
1      +      Abstract syntax tree : this is what it is
2      / \      represented internally
3     2  3
4
5  Equivalent concrete structures: 2 + 3
6                                + 2 3
7
```



From concrete to abstract

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

Say we have this concrete syntax:

```

1  <exp> ::= <num> |           \_ prefix
2          '(+ ' <exp> ' ' <exp> ')' /
3
4  <exp> ::= <num> |           \_ infix
5          '(' <exp> '+' <exp> ')' /

```

The abstract syntax for that language is:¹

```

1  <ast> ::= <num> |           \_ grammar of trees
2          + <ast> <ast>      /
3
4  We pass judgement on the astness of an ast using
5  this:
6  |-----|
7  | if n N   N   | num rule
8  | ⇒ n AST
9  |           OR
10 | if e1 AST & e2 AST | plus rule
11 | ⇒ + e1 e2 AST
12 |-----|
13
14 -----
15 | Now, the valid expressions
16 | V in this language are
17 | usable for rating? judgements
18
19 Judgement: | e AST |
20

```

An example of the judgement:

¹at 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.

Judgement	rating	justification
3 AST	sound	num, $3 \in \mathbb{N}$
2 AST	sound	num, $2 \in \mathbb{N}$
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

```

1 <ast> ::= <num> | + <ast> <ast>
2         -----v-----
3         |
4         V
5         Variant (sum) type
6         base case | Inductive case

```

Implementing it in racket:

```

1 > (define-datatype ast ast? ;; the second is the type predicate
2   [num (n number?)]
3   [plus (left ast?) (right ast?)])
4
5 > (number? 5)
6 #t
7 > symbol?
8 > procedure?
9 ;; so
10 > (check-true (ast? (num 5)))
11 #t

```

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

- `num ::= number? -> ast?`
- `plus ::= [ast?, ast?] -> ast?`

Eg:

```

1 > (num 5); --> an AST          num 5
2
3 ;; example:
4 > (ast? (plus (num 5)          ;      plus
5           (num 6)))           ;      /  \
6 #t                           ;      num5  num6
7
8 ; example of more complex????
9 > (let ([e1 (plus (num 5) (num 6))] ;      +
10       [e2 (plus (num 3) (num 3))] ;      /  \
11       (plus e1 e2))              ;      +    +
12                                     ;      /  \ /  \
13                                     ;      n5 n6 n2 n3

```

Now something something looking at abstract to concrete syntax

Abstract syntax —unparser→ concrete syntax ←—parser—

Unparser implementation in RACKET

```

1 ;;; unparse : ast? ----> any/c
2 > (define (unparse a)
3   (cases ast a
4     [num (n) n]
5     [plus (left right)
6       (list '+
7             (unparse left)
8             (unparse right))]))
9 > (unparse (num 5))
10 5
11 > (unparse (plus (num 5) (num 4)))
12 '(+ 5 4)

```

Parser implementation in RACKET

```

1 ;;; parse : any/c ----> ast? || error
2
3 > (define (parse d)
4   (cond [(number? d) (num d)]
5         [(and (list? d)
6               (= (length d) 3)
7               (eq? (first d) '+))
8         (plus (parse (second d))
9               (parse (third d)))]
10          [else (error 'parse "invalid syntax" d)]
11          )
12 > (parse 5)

```

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