PoPL Lecture 5

Zubair Abid

2020-08-25

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:

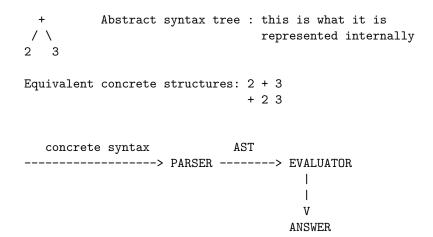
```
def f(x):
```

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



From concrete to abstract

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

Say we have this concrete syntax:

The abstract syntax for that language is: ¹

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

| Now, the valid expressions V in this language are

 $^{^1}$ 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.

usable for rating? judgements

Judgement: | e AST |

An example of the judgement:

Judgement	rating	justification
3 AST	sound	num, 3 N
2 AST	sound	num, 2 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

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
```

 ${\tt num}$ and ${\tt plus}$ get autodefined as $constructor\ functions$ with the following signatures:

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

```
Eg:
> (num 5); --> an AST
                                       num 5
;; example:
                                           plus
> (ast? (plus (num 5)
              (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

Parser implementation in RACKET

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