Full Lisp Interpreter

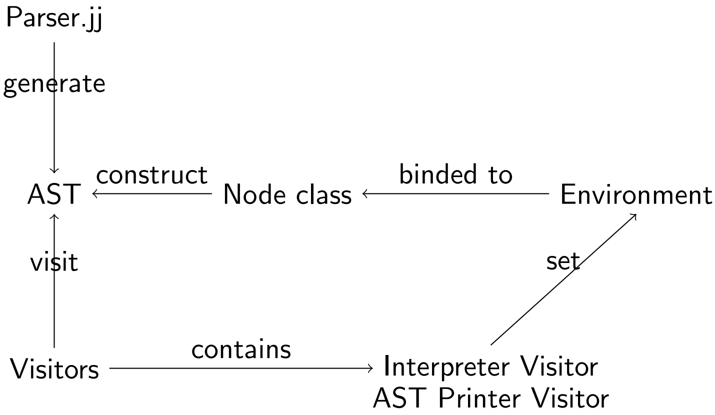
Shun Zhang
Wang Rao
Zeyuan Zhu

November 30, 2012

Features

- AST built by hand, without `jjtree`.
- Curried functions (i.e. every lambda expression on the AST has only one argument).
- Each node has its own environment.
- Multiple, extensible visitors (easy to add a `SQLInsertionVisitor`, if needed).
- Static/Dynamic scoping.

Structure



Parser.jj

generate

AST

construct

Node class

binded to

Environment

visit

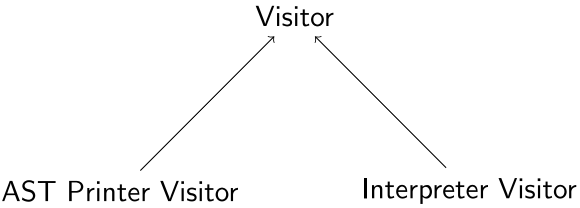
set

Visitors

contains

Interpreter Visitor
AST Printer Visitor

Visitor



Visitor

AST Printer Visitor

Interpreter Visitor

- Interpreter Visitor: Evaluate each node, print out its environment, and the final result.
- AST Printer Visitor: Print out what it sees on the AST.

Grammar

$$Lambda \rightarrow (lambda\ ((Id)*)\ Expression|Lambda)$$

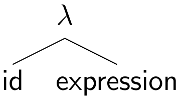
$$Application \rightarrow (Lambda|Application\ \{Expression|Lambda\})$$
$$|\ (let\ ((Id\ \ Expression|Lambda)*)\ Expression|Lambda)$$

$$Addition \rightarrow (+\ (Expression)*)$$

$$Expression \rightarrow Application|Addition|Id|Number$$

AST: Lambda

$(\mathit{lambda}\ (\mathit{id})\ \mathit{expression})$

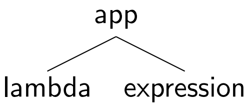


λ

id expression

AST: Application

(*lambda expression*)



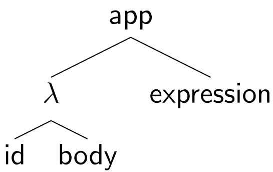
app

lambda

expression

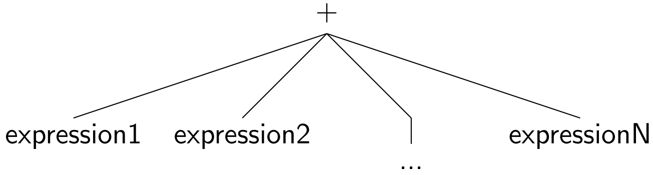
AST: Application

$(let\ ((id\ \ expression))\ body)$



AST: Addition

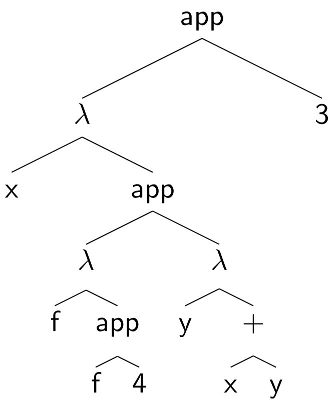
$(+\ expression1\ expression2\ ..\ expressionN)$



Example

(let ((x 3)) (let ((f (lambda (y) (+ x y)))) (f 4)))

Or ((lambda (x) ((lambda (f) (f 4)) (lambda (y) (+ x y)))) 3)



Demo: Input_sample

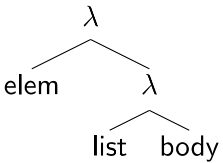
Environment

- Each node has its own environment.
- Envrionment is passed down along the AST when interpreter is visiting.
- When interpreter sees an Application, it constructs an ASub object and appends it to the environment.
- ASub has two types: simple ASub and closure ASub.

Preloaded Functions: car, cdr, cons

Functions of `car, cdr, cons` can be preloaded in to environment by `-p` flag. They will appear at the envrionment of the root of AST.

Example: function cons appears as



Demo: Input_list

Preloaded Functions: Combinators

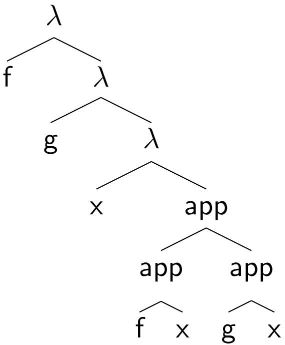
All combinators are also preloaded. Their definitions are stored in a `Preload` file in the following format. `Parser.jj` will parse `Preload` file first to load these functions into environment.

Parse them by `(<ID> <ASSIGN> lambda())*`

```
s := (lambda (f g x) (f x (g x)))
k := (lambda (x y) x)
b := (lambda (f g x) (f (g x)))
c := (lambda (f g x) (f x g))
y := (lambda (f x) (f (y f) x))
...
pradd1 := (lambda (x z) (y (b (condzero (k z)) (b (s
(b plus (k 1))) (c b pred))) x))
...
```

Preloaded Functions: Combinators

For example: `s f g x = f x (g x)`. The structure it appears in the environment is



Multiple Binding of Let, Lambda

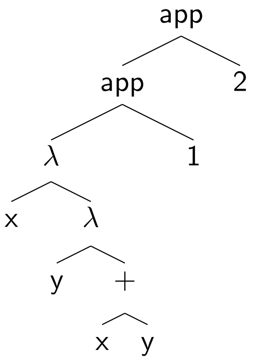
- Translate multiple binding of lambda to curried function in the parser.
- Application can return function.
- Clean AST structure.

Multiple Binding of Let, Lambda : Example

(let ((x 1) (y 2)) (+ x y)

Or ((lambda (x y) (+ x y)) 1 2)

→ (((lambda (x) (lambda (y) (+ x y))) 1) 2)



Demo: Input_multi_lambda Input_multi_let

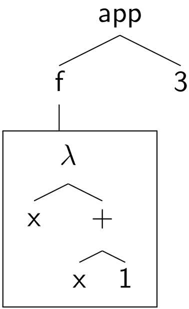
Multiple Binding of Let, Lambda : Implementation

```
<LPAR> <LAMBDA> <LPAR> (t = <ID> { tlist.add(t.image);
})* <RPAR>
(LOOKAHEAD(2) exp = expression() | exp = lambda() ) <RPAR>

{
for ( int i = tlist.size() - 1; i >=0; i-- )
{
exp = new Lambda(tlist.get(i), exp);
}
return (Lambda)exp;
}
```

Static/Dynamic Scoping

For example, f is found in the envrionment as `(lambda (x) (+ x 1))` and interpreter sees `(f 3)`



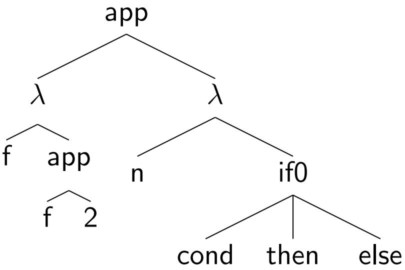
- Static scoping : as it is.
- Dynamic scoping : + node using the environment of *app* node.

Demo: Input_scope

Recursion

Example

```
(letrec ((f (lambda (n) (if0 n 1 (+ (f (+ n -1))
n))))) (f 2))
```



Recursion

- If defined by `letrec`, put a copy of the function definition into its environment.
- Example

  ```
  (letrec ((f (lambda (n) (if0 n 1 (+ (f (+ n -1))
  n))))) (f 2))
  ```

  Intuition: $f = \begin{cases} 1, & \mathit{if} \;\; n = 0 \\ f(n-1) + n, & \mathit{otherwise} \end{cases}$

  $f(2) = 2 + f(1) = 2 + 1 + f(0) = 2 + 1 + 1 = 4$

Demo: Input_rec

What we learnt

- Visitor design pattern.
- Mechanism of interpreter.
- Deep understanding of object-oriented programming.