

Principles of Programming Languages Handout

As a first example of abstract syntax trees, and procedures that walk these trees, we consider the λ -calculus.

Recall that expressions in the λ -calculus are defined inductively as follows:

- if x is a symbol (variable), then x is also an expression;
- if x is a symbol, and M an expression, then $\lambda x.M$ is also an expression;
- if M and N are both expressions, then the application of the operator M to the operand N is also an expression.

Here is the inductive datatype for the abstract syntax of the λ -calculus:

```
(define-datatype expression expression?
  (var-exp
    (id symbol?))
  (lambda-exp
    (id symbol?)
    (body expression?))
  (app-exp
    (rator expression?)
    (rand expression?)))
```

This declaration says that the constructors of the datatype `expression` are `var-exp`, `lambda-exp` and `app-exp`; the procedure `expression?` tests whether its argument is an expression. Datatypes can be taken apart by a pattern-matching construct called `cases`.

This procedure computes whether a given variable occurs free in an expression:

```
(define occurs-free?
  (lambda (var exp)
    (cases expression exp
      (var-exp (id) (eqv? id var))
      (lambda-exp (id body)
        (and (not (eqv? id var))
              (occurs-free? var body)))
      (app-exp (rator rand)
        (or (occurs-free? var rator)
            (occurs-free? var rand))))))
```

The following procedure takes an abstract syntax tree, and returns its representation as a Scheme list:

```
(define unparse-expression
  (lambda (exp)
    (cases expression exp
      (var-exp (id) id)
      (lambda-exp (id body)
        (list 'lambda (list id)
              (unparse-expression body)))
      (app-exp (rator rand)
        (list (unparse-expression rator)
              (unparse-expression rand))))))
```

Conversely, the following procedure *parses* expressions, that is, it reads a concrete representation and builds the abstract syntax tree:

```
(define parse-expression
  (lambda (datum)
    (cond
      ((symbol? datum) (var-exp datum))
      ((pair? datum)
       (if (eqv? (car datum) 'lambda)
           (lambda-exp (caadr datum)
                       (parse-expression (caddr datum)))
           (app-exp
            (parse-expression (car datum))
            (parse-expression (cadr datum)))))
      (else (eopl:error 'parse-expression
                        "Invalid concrete syntax ~s" datum))))))
```