# 2.2.2 Abstract Syntax

- Recall BNF definition of λ-calculus expressions:

<expression> ::= <identifier>

::= **(lambda (**<identifier>**)** <expression> **)**

::= **(**<expression> <expression>**)**

- Uses *concrete* syntax: includes parens, and keywords like **lambda**

- We can use this to define an *abstract* datatype without these redundant items:

```
(define-datatype expression expression?
    (var-exp  ::= <identifier>
        (id symbol?))

    (lambda-exp ::= (lambda (<identifier>) <expression>
        (id symbol?)
        (body expression?))

    (app-exp ::= (<expression> <expression>)
        (rator expression?)
        (rand expression?)))
```

- We can use the abstract data type together with the BNF definition, to make an *abstract syntax tree*

- First, we annotate the BNF with tree notation:

<expression> ::= <identifier>

**`var-exp (id)`**

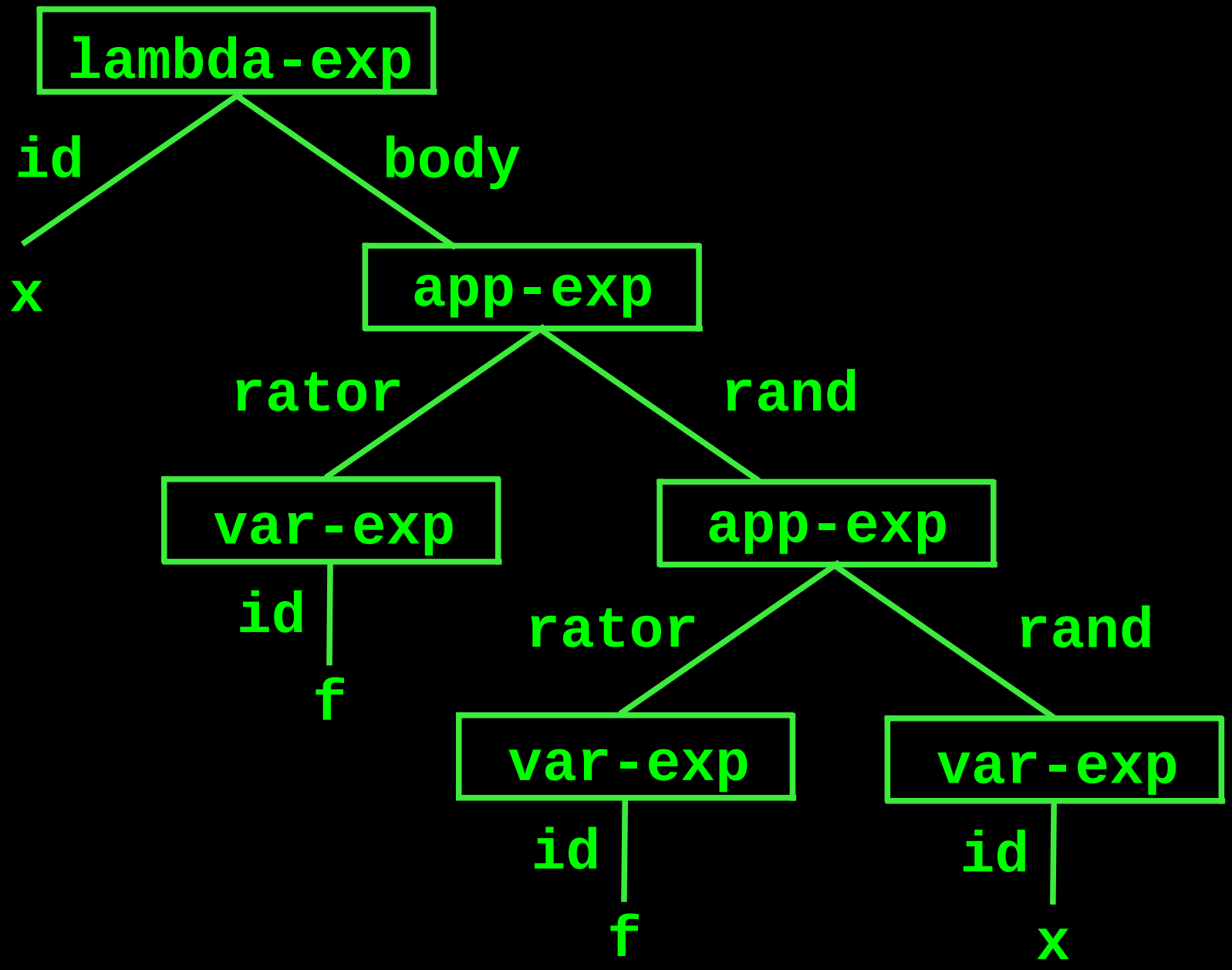
::= **`(lambda (`**<identifier>**`)`** <expression>

**`lambda-exp (id body)`**

::= **`(`**<expression> <expression>**`)`**

**`app-exp (rator rand)`**

# Example: `(lambda (x) (f (f x)))`



We can use the defined datatype to simplify functions:

```
(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))))))
```

Example:

```
> (define e (lambda-exp 'a (var-exp 'b)))
> (expression? e)
#t
> (occurs-free? 'a e)
#f
> (occurs-free? 'b e)
#t
```

But this is awkward.  We want to *parse* actual expressions into the data structure....

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

```
> (parse-expression '(lambda(a) (a b)))
#(struct:lambda-exp a #(struct:app-exp
              #(struct:var-exp a) #(struct:var-exp b)))
```