Transforming Trees

Matt Might

University of Utah

matt.might.net

Transforming Trees

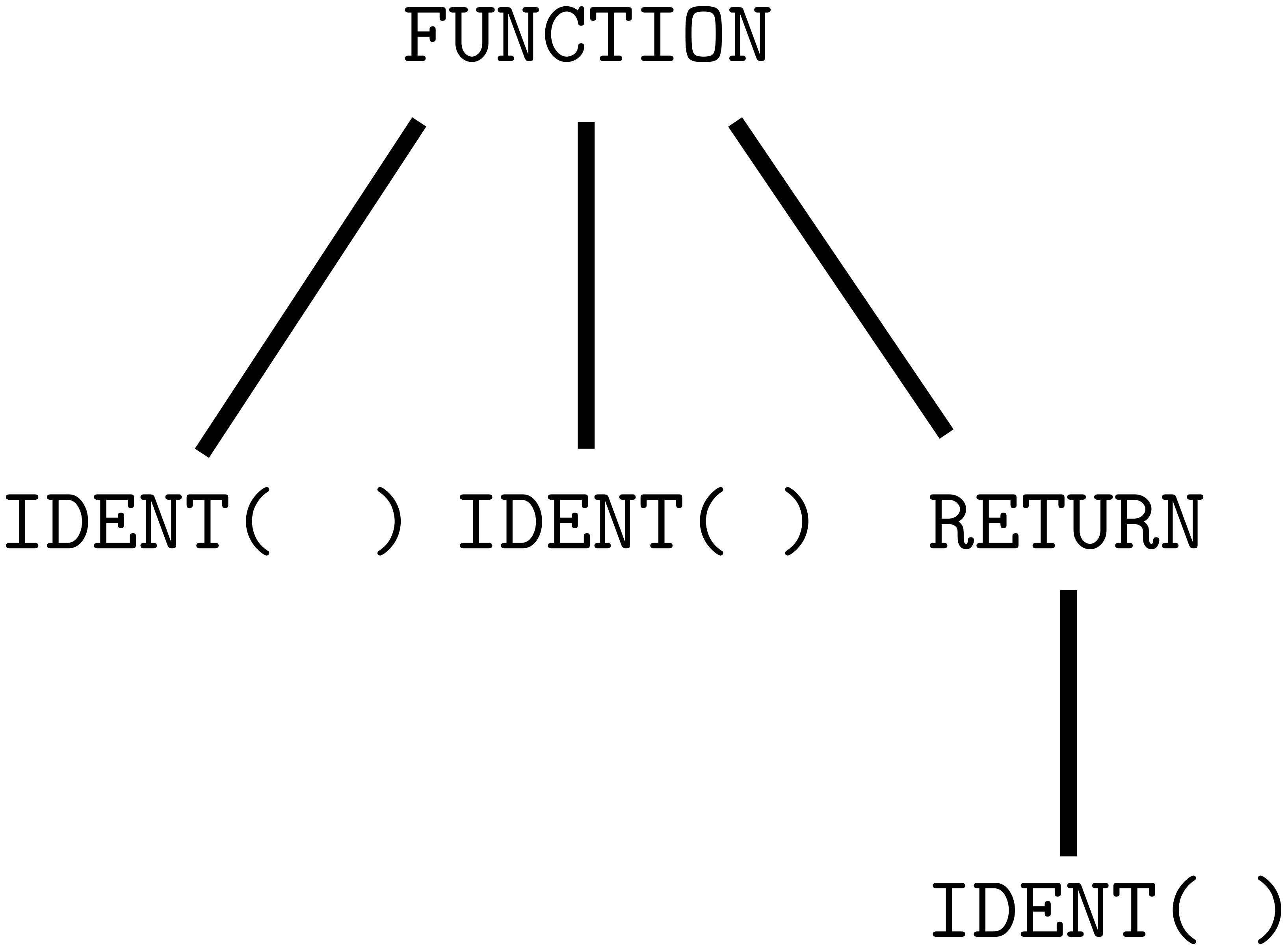
Matt Might

University of Utah

matt.might.net

```
function id(x)
{
 return x ;  // comment
}
```

```
FUNCTION

IDENT(id)

LPAR

IDENT(x)

RPAR

LBRACE

RETURN

IDENT(x)

SEMI

RBRACE
```



FUNCTION

IDENT( ) IDENT( ) RETURN

IDENT( )

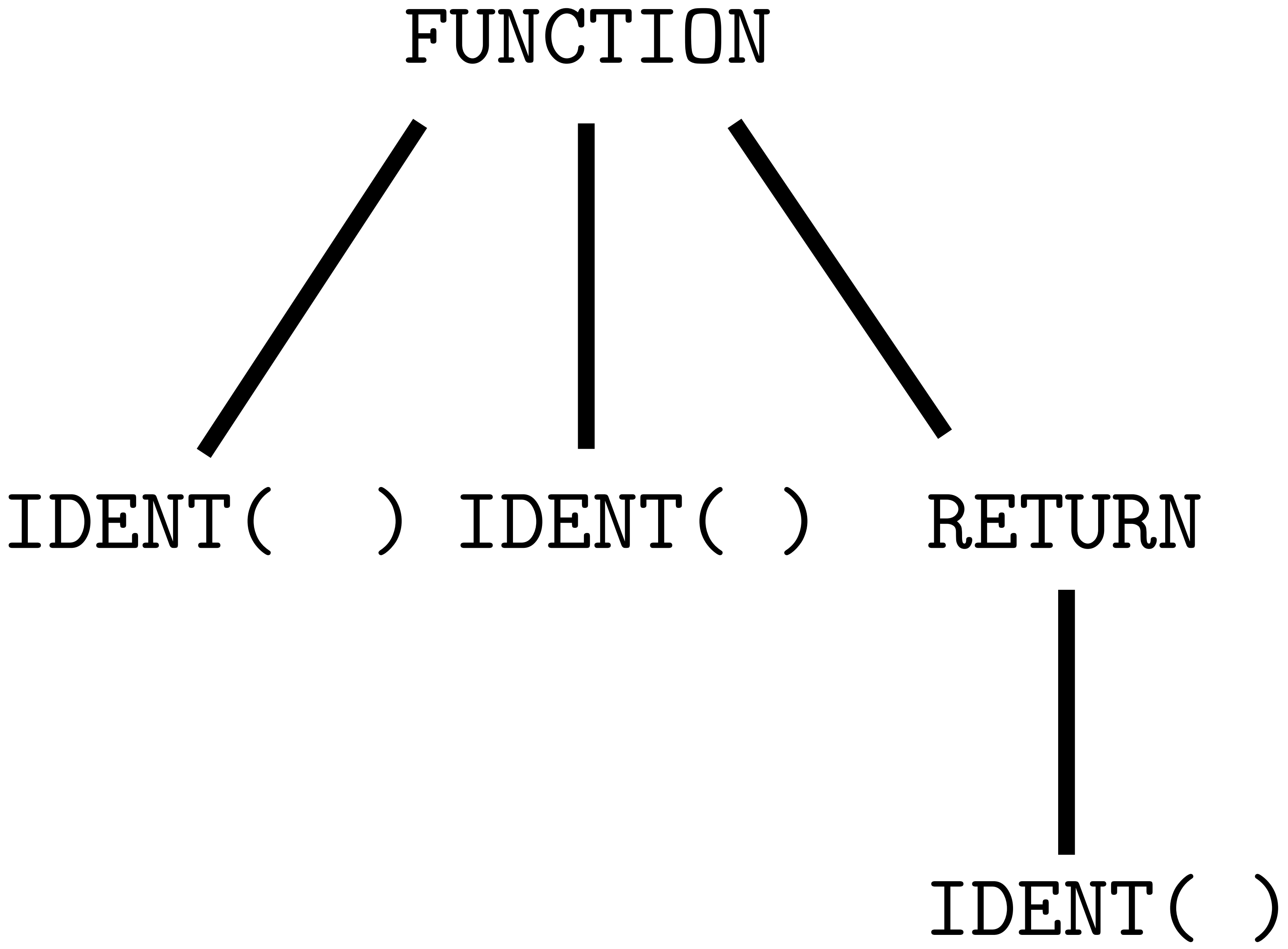
LPAR

RPAR

LBRACE

SEMI

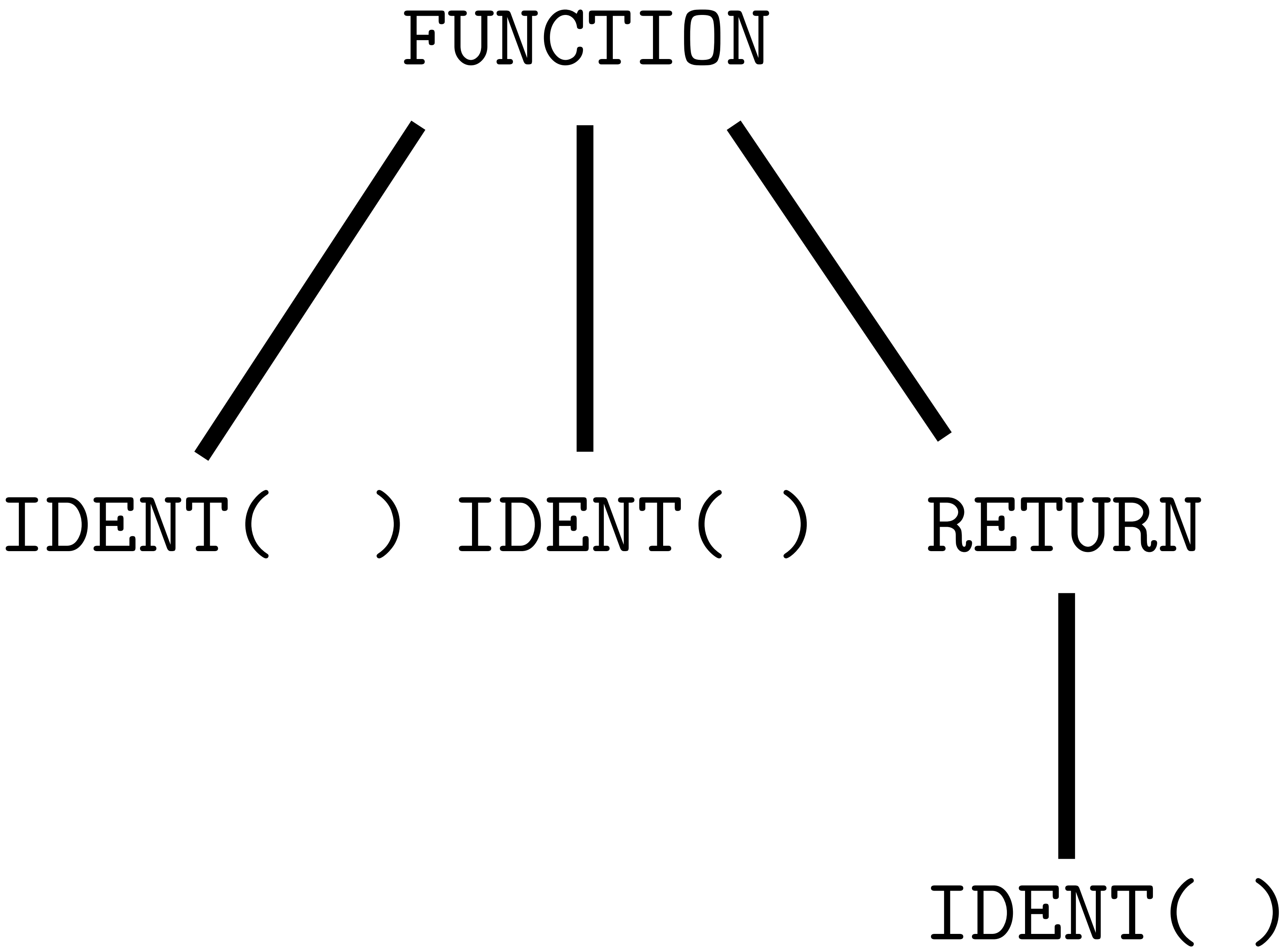
RBRACE



FUNCTION

IDENT(   ) IDENT( )   RETURN

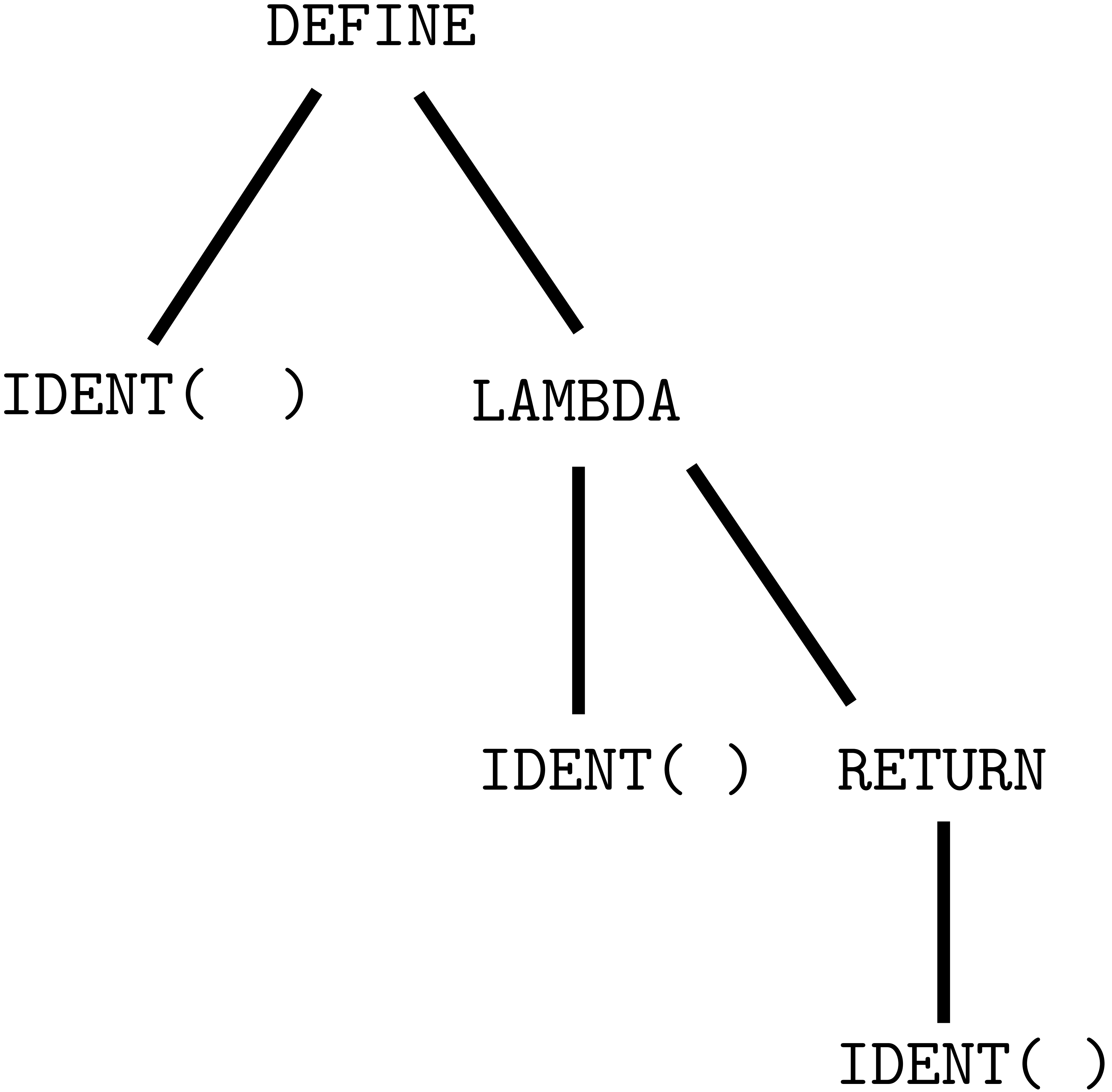
IDENT( )



FUNCTION

IDENT(   ) IDENT( )   RETURN

IDENT( )



DEFINE

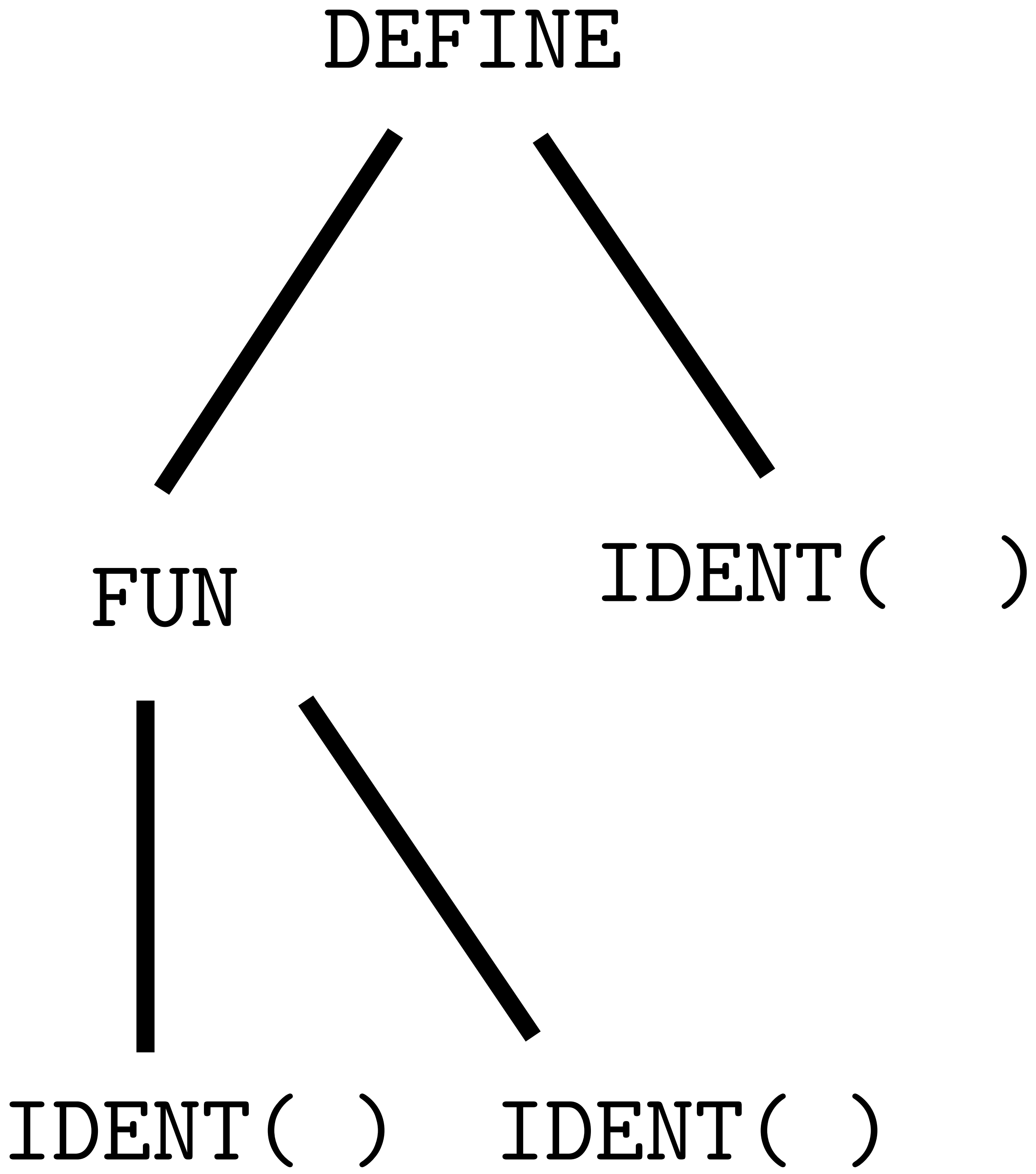
IDENT( )

LAMBDA

IDENT( )

RETURN

IDENT( )



DEFINE

FUN

IDENT( )

IDENT( )

IDENT( )

Today

- Project 3 hints and tips
- Trees, tree transforms
- Live coding transforms

Project 3:Tips

```
; <program> ::= (program <stmt>*)


; <funcdef> ::= (def (<NAME> <NAME>*) <suite>)
; <stmt> ::= <simple_stmt> | <compound_stmt>
; <simple_stmt> ::= <small_stmt> | (begin <small_stmt>+)
; <small_stmt> ::= <expr_stmt>
;               |  <del_stmt>
;               |  <pass_stmt>
;               |  <flow_stmt>
;               |  <global_stmt>
;               |  <nonlocal_stmt>
;               |  <assert_stmt>
; <expr_stmt> ::= (<augassign> (<test>+) <tuple_or_test>)
;              |  (=           (<test>+) <tuple_or_test>)
;              |  (expr <tuple_or_test>)
; <augassign> ::= "+=" | "-=" | "*=" | "/="  | "%="
;              |  "&=" | "|=" | "^=" | "<<=" | ">>=" | "**=" | "//="
; <del_stmt> ::= (del <star_expr>)
; <pass_stmt> ::= (pass)
; <flow_stmt> ::= <break_stmt> | <continue_stmt> | <return_stmt> | <raise_stmt>
; <break_stmt> ::= (break)
; <continue_stmt> ::= (continue)
; <return_stmt> ::= (return <test>*)
; <raise_stmt> ::= (raise [ <test> [ <test> ] ])
; <global_stmt> ::= (global <NAME>+)
; <nonlocal_stmt> ::= (nonlocal <NAME>+)
; <assert_stmt> ::= (assert <test> [ <test> ])
; <compound_stmt> ::= <if_stmt> | <while_stmt> | <for_stmt> | <try_stmt> | <funcdef>
; <if_stmt> ::= (cond (<test> <suite>)+ [ (else <suite>) ])
; <while_stmt> ::= (while <test> <suite> [ <suite> ])
; <for_stmt> ::= (for <NAME> <test> <suite> [ <suite> ])
; <try_stmt>      ::= (try <suite> ((<catch> <suite>)*) <maybe-else> <maybe-finally>)
; <maybe-else>    ::= <suite> | #f
; <maybe-finally> ::= <suite> | #f
; <catch> ::= (except [ <test> [ <NAME> ] ])
; <suite> ::= <simple_stmt> | (suite <stmt>+)


; <test> ::= (if <or_test> <or_test> <test>)
;         |  <or_test>
;         |  <lambdef>
; <lambdef> ::= (lambda (<NAME>*) <test>)
; <or_test> ::= <and_test> | (or <and_test>+)
; <and_test> ::= <not_test> | (and <not_test>+)
; <not_test> ::= <comparison> | (not <not_test>)
```

```
; <comparison> ::= <star_expr> | (comparison <star_expr> (<comp_op> <star_expr>)+)
; <comp_op> ::= "<" | ">" | "==" | ">=" | "<=" | "<>" | "!= " | "in"
;            |  "not-in" | "is" | "is-not"
; <star_expr> ::= <expr> | (star <expr>)
; <expr> ::= <xor_expr> | (bitwise-or <xor_expr>+)
; <xor_expr> ::= <and_expr> | (bitwise-xor <and_expr>+)
; <and_expr> ::= <shift_expr> | (bitwise-and <shift_expr>+)
; <shift_expr> ::= <arith_expr> | (shift <arith_expr> (<shift_op> <arith_expr>)+)
; <shift_op> ::= "<<" | ">>"
; <arith_expr> ::= <term> | (arith <term> (<arith_op> <term>)+)
; <arith_op> ::= "+" | "-"
; <term> ::= <factor> | (term <factor> (<factor_op> <factor>)+)
; <factor_op> ::= "*" | "/" | "%" | "//"
; <factor> ::= <power> | (<unary_op> <factor>)
; <unary_op> ::= "+" | "-" | "~"
; <indexed> ::= <atom> | (indexed <atom> <trailer>+)
; <power> ::= <indexed> | (power <indexed> <factor>)
; <atom> ::= <tuple_or_test> | (tuple)
;         |  (list [ <testlist> ])
;         |  <dict>
;         |  <set>
;         |  <NAME>
;         |  <NUMBER>
;         |  <STRING>
;         |  Ellipsis
;         |  None
;         |  True
;         |  False

; <trailer> ::= (called [ <arglist> ])
;            |  (subscript <tuple_or_test>)
;            |  (dot <NAME>)

; <testlist> ::= <test>+
; <tuple_or_test> ::= <test> | (tuple <test>+)
; <dict> ::= (dict (<test> <test>)*)
; <set> ::= (set <test>*)
; <arglist> ::= <test>+
```

python-ast-spec.txt

```
<program> ::= (program <top-form>*)

<top-form> ::= <vardef> | <exp>

<vardef> ::= (define <var> <exp>)

<exp> ::=  (void)
       |   (error <exp>)

       |   (lambda (<var>*) <exp>)
       |   (call/ec (lambda (<var>) <exp>))

       |   <var>
       |   <number>
       |   <string>

       |   integer?
       |   string?
       |   tuple?
       |   dict?
       |   py-list?
       |   set?

       |   (set <exp>*)
       |   (dict (<exp> <exp>) ...)
       |   (tuple <exp>*)
       |   (py-list* <exp>*)

       |   (let ((<var> <exp>)*) <exp>*)
       |   (set! <var> <exp>)

       |   (py-list-ref <exp> <exp>)
       |   (py-list-set! <exp> <exp> <exp>)
       |   (py-list-remove! <exp> <exp>)

       |   (tuple-ref <exp> <exp>)
       |   (tuple-set! <exp> <exp> <exp>)

       |   (dict-ref <exp> <exp>)
       |   (dict-set! <exp> <exp> <exp>)
       |   (dict-remove! <exp> <exp>)

       |   (get-field <exp> <var>)
       |   (set-field! <exp> <var> <exp>)
```

```
       |   (remove-field! <exp> <var>)

       |   (get-global <var>)
       |   (set-global! <var> <exp>)

       |   (throw <exp>)
       |   (try <exp> <exp>)

       |   (assert <exp> [ <exp> ])
       |   (cond (<exp> <exp>)* [ (else <exp>) ])

       |   (if <exp> <exp> <exp>)
       |   (and <exp>*)
       |   (or <exp>*)
       |   (not <exp>)
       |   (cond (<exp> <exp>) ... [ (else <exp>) ])
       |   (while <exp> <exp> [ <exp> ])
       |   (for-each <var> <exp> <exp> [ <exp> ])
       |   (break)
       |   (continue)
       |   (begin <exp> ...)

       |   (<multop> <exp>*)
       |   (<binop> <exp> <exp>)
       |   (<unop> <exp>)

       |   py-print | Exception | Object

       |   None | Ellipsis | #t | #f


<multop> ::= bitwise-and | bitwise-ior | bitwise-xor

<binop> ::= < | > | equal? | >= | <= | not-equal? | in? | not-in? | eq? | not-eq?
         |  << | >>
         |  + | -
         |  * | / | quotient | modulo
         |  expt

<unop> ::= bitwise-not | + | -
```

hir-spec.rkt

```
(program (expr 3))
```

(program 3)

```
x = 3
print(x)
```

```
(program (= (x) 3)
         (expr (indexed print
                        (called x))))
```

```
(program
 (define x (void))
 (set-global! x 3)
 (py-print (get-global x)))
```

```
(program
 (define x (void))
 (set! x 3)
 (py-print x))
```

```
(program
 (define x (void))
 (set! x 3)
 (py-print x))
```

hir-header.rkt

```
(program
 (define x (void))
 (set! x 3)
 (py-print x))
```

hir-header.rkt

```
(program
 (define x (void))
 (set! x 3)
 (py-print x))
```

hir-header.rkt

```
(program
 (define x (void))
 (set! x 3)
 (py-print x))
```

$\xrightarrow{\texttt{racket}}$

```
3
```

pytrans-stub.rkt

Three ingredients

- Tree traversal
- List operations
- Quasiquotation

Tree-transforms all the way down.

How to write transforms?

How to encode trees?

The C way

dec ::= **var** *v* ;

| **function** $v(v_1, \ldots, v_n)$ *stmt*

stmt ::= **while** (*exp*) *stmt*

| **if** (*exp*) *stmt* **else** *stmt*

| *v* = *exp* ;

| { $stmt_1$ ... $stmt_n$ }

| **return** *exp* ;

exp ::= *v*

| *n*

| *exp* + *exp*

```
typedef enum { FUN_DEC
             , VAR_DEC
             , IF_STMT
             , WHILE_STMT
             , ASSIGN_STMT
             , BLOCK_STMT
             , RETURN_STMT
             , SUM_EXP
             , INT_EXP
             , REF_EXP } tag_t ;

union Node ;
typedef union Node Node ;
```

```
union Node {

  tag_t tag ;

  struct {
    tag_t tag ;
    char* name ;
    unsigned int num_params ;
    char** params ;
    Node* body ;
  } fun_dec ;

  // ...
```

```
struct {
  tag_t tag ;
  char* name ;
} var_dec ;

struct {
  tag_t tag ;
  char* name ;
  Node* value ;
} assign_stmt ;

// ...
```

```
struct {
  tag_t tag ;
  Node* condition ;
  Node* consequent ;
  Node* alternate ;
} if_stmt ;

struct {
  tag_t tag ;
  Node* condition ;
  Node* body ;
} while_stmt ;

// ...
```

```
struct {
  tag_t tag ;
  unsigned int num_stmts ;
  Node** stmts ;
} block_stmt ;

struct {
  tag_t tag ;
  Node* ret_value ;
} return_stmt ;

// ...
```

```
struct {
  tag_t tag ;
  Node* lhs ;
  Node* rhs ;
} sum_exp ;

struct {
  tag_t tag ;
  char* name ;
} ref_exp ;

// ...
```

```
 struct {
  tag_t tag ;
  int value ;
 } int_exp ;
} ;
```

The object-oriented way

```
dec ::= var v ;
     |  function v($v_1$,...,$v_n$) stmt

stmt ::= while (exp) stmt
      |  if (exp) stmt else stmt
      |  v = exp ;
      |  { $stmt_1$ ... $stmt_n$ }
      |  return exp ;

exp ::= v
     |  n
     |  exp + exp
```

abstract class Dec {}

dec ::= **var** *v* ;
    | **function** $v(v_1, \ldots, v_n)$ *stmt*

abstract class Stmt {}

stmt ::= **while** (*exp*) *stmt*
    | **if** (*exp*) *stmt* **else** *stmt*
    | *v* = *exp* ;
    | { $stmt_1$ ... $stmt_n$ }
    | **return** *exp* ;

abstract class Exp {}

exp ::= *v*
    | *n*
    | *exp* + *exp*

```
abstract class Dec {}
```

```
abstract class Stmt {}
```

```
abstract class Exp {}
abstract class Dec {}
class VarDec extends Dec {...}
class FunDec extends Dec {...}
abstract class Stmt {}
class WhileStmt extends Stmt {...}
class IfStmt extends Stmt {...}
class AssignStmt extends Stmt {...}
class BlockStmt extends Stmt {...}
class ReturnStmt extends Stmt {...}
abstract class Exp {}
class RefExp extends Exp {...}
class IntExp extends Exp {...}
class SumExp extends Exp {...}
```

```
abstract class Dec {}
abstract class Stmt {}
abstract class Exp {}

class VarDec extends Dec {
  public String name ;
}

class FunDec extends Dec {
  public String f ;
  public String[] params ;
  public Stmt body ;
}

class AssignStmt extends Stmt {
  public String name ;
  public Exp value ;
}

class IfStmt extends Stmt {
  public Exp condition ;
  public Exp consequent ;
  public Exp alternate ;
}
```

```
class WhileStmt extends Stmt {
  public Exp condition ;
  public Stmt body ;
}

class BlockStmt extends Stmt {
  public Stmt[] stmts ;
}

class ReturnStmt extends Stmt {
  public Exp value ;
}

class RefExp extends Exp {
  public String name ;
}

class IntExp extends Exp {
  public int value ;
}

class SumExp extends Exp {
  public Exp lhs ;
  public Exp rhs ;
}
```

The functional way

```
dec ::= var v ;
     |  function v(v1,...,vn) stmt

stmt ::= while (exp) stmt
      |  if (exp) stmt else stmt
      |  v = exp ;
      |  { stmt1 ... stmtn }
      |  return exp ;

exp ::= v
     |  n
     |  exp + exp
```

dec ::= (**vardec** *v*)

| (**fundec** *v* (v_1 ... v_n) *stmt*)

stmt ::= (**while** *exp* *stmt*)

| (**if** *exp* *stmt* *stmt*)

| (**=** *v* *exp*)

| (**block** $stmt_1$ ... $stmt_\mathrm{n}$)

| (**return** *exp*)

exp ::= *v*

| *n*

| (**+** *exp* *exp*)

```
data dec = VarDec var
         | FunDec var [var] stmt

data stmt = While exp stmt
          | If exp stmt stmt
          | Assign var exp
          | Block [stmt]
          | Return exp

data exp = Ref var
         | Int int
         | Sum exp exp
```

Tree transformation

Constant-folding

```
x + 3 + 4 * 7
```

```
x + 31
```

```
Node* fold_constants(Node* exp) {
  Node* l ;
  Node* r ;
  Node* n ;

  switch (exp->tag) {
    case SUM_EXP:
       n = malloc(sizeof(Node)) ;
       l = fold_constants(exp->sum_exp.lhs) ;
       r = fold_constants(exp->sum_exp.lhs) ;
       if ((l->tag == INT_EXP) && (r->tag == INT_EXP)) {
         n->tag = INT_EXP ;
         n->int_exp.value = l->int_exp.value + r->int_exp.value ;
       } else {
         n->tag = SUM_EXP ;
         n->sum_exp.lhs = l ;
         n->sum_exp.rhs = r ;
       }
    // free resources in exp?
    return n ;

    default:
    return exp ;
  }
}
```

```
abstract class Exp {
  public abstract Exp foldConstants() ;
}

class RefExp extends Exp {
  public String name ;

  public Exp foldConstants() {
    return this ;
  }
}

class IntExp extends Exp {
  public int value ;

  public IntExp(int value) {
    this.value = value ;
  }

  public Exp foldConstants() {
    return this ;
  }
}
```

```
class SumExp extends Exp {
  public Exp lhs ;
  public Exp rhs ;

  public SumExp(Exp lhs, Exp rhs) {
    this.lhs = lhs ;
    this.rhs = rhs ;
  }

  public Exp foldConstants() {
    Exp l = lhs.foldConstants() ;
    Exp r = rhs.foldConstants() ;
    if (l instanceof IntExp &&
        r instanceof IntExp)
      return
       new IntExp(((IntExp)l).value +
                  ((IntExp)r).value) ;
    else
      return new SumExp(l,r) ;
  }
}
```

```
(define (fold-constants exp)
  (match exp
    [`(+ ,l ,r)
      (let ((l (fold-constants l))
            (r (fold-constants r)))
        (if (and (number? l) (number? r))
            (+ l r)
            `(+ ,l ,r)))]

    [else exp]))
```

```
(define (fold-constants exp)
  (match exp
    [`(+ ,(app fold-constants (and (? number?) l))
         ,(app fold-constants (and (? number?) r)))
     (+ l r)]

    [`(+ ,(app fold-constants l) ,(app fold-constants r))
     `(+ ,l ,r)]

    [else exp]))
```

Transform exercises

Code: SExp -> SXML

`quasiquote`

‘*sx* = (quote *sx*)

```
‘num => num
‘symbol => ‘symbol
‘bool => bool
‘string => string
```

$$\texttt{`(}e_1\ e_2\ \texttt{...)}$$

$$\texttt{=>}$$

$$\texttt{(}\textbf{\texttt{cons}}\ \texttt{`}e_1\ \texttt{`(}e_2\ \texttt{...))}$$

`sx = (quasiquote sx)

,sx = (unquote sx)

,@sx = (unquote-splicing sx)

```
`num => num
`symbol => ‘symbol
`bool => bool
`string => string
```

```
`,exp  =>  exp
```

$$\texttt{`(}e_1\ e_2\ \texttt{...)}$$

$$\texttt{=>}$$

$$\texttt{(}\textbf{\texttt{cons}}\ \texttt{`}e_1\ \texttt{`(}e_2\ \texttt{...))}$$

```
`(,@$e_1$ $e_2$ ...)
=>
(append $e_1$ `($e_2$ ...))
```

```
`()
=>
‘()
```

Reminder on lists

(cons e_1 e_2**)**

=>

(e_1 . e_2)

$$\texttt{(}e\ \ \texttt{.}\ \ \texttt{(}e_1\ \ \texttt{...}\ \ e_n\texttt{))}$$

$$=>$$

$$\texttt{(}e\ \ e_1\ \ \texttt{...}\ \ e_n\texttt{)}$$

```
(e . ())
   ==
  (e)
```

match

```
(match exp
  [pattern body]
  ...)
```

Code: Quasiquotation