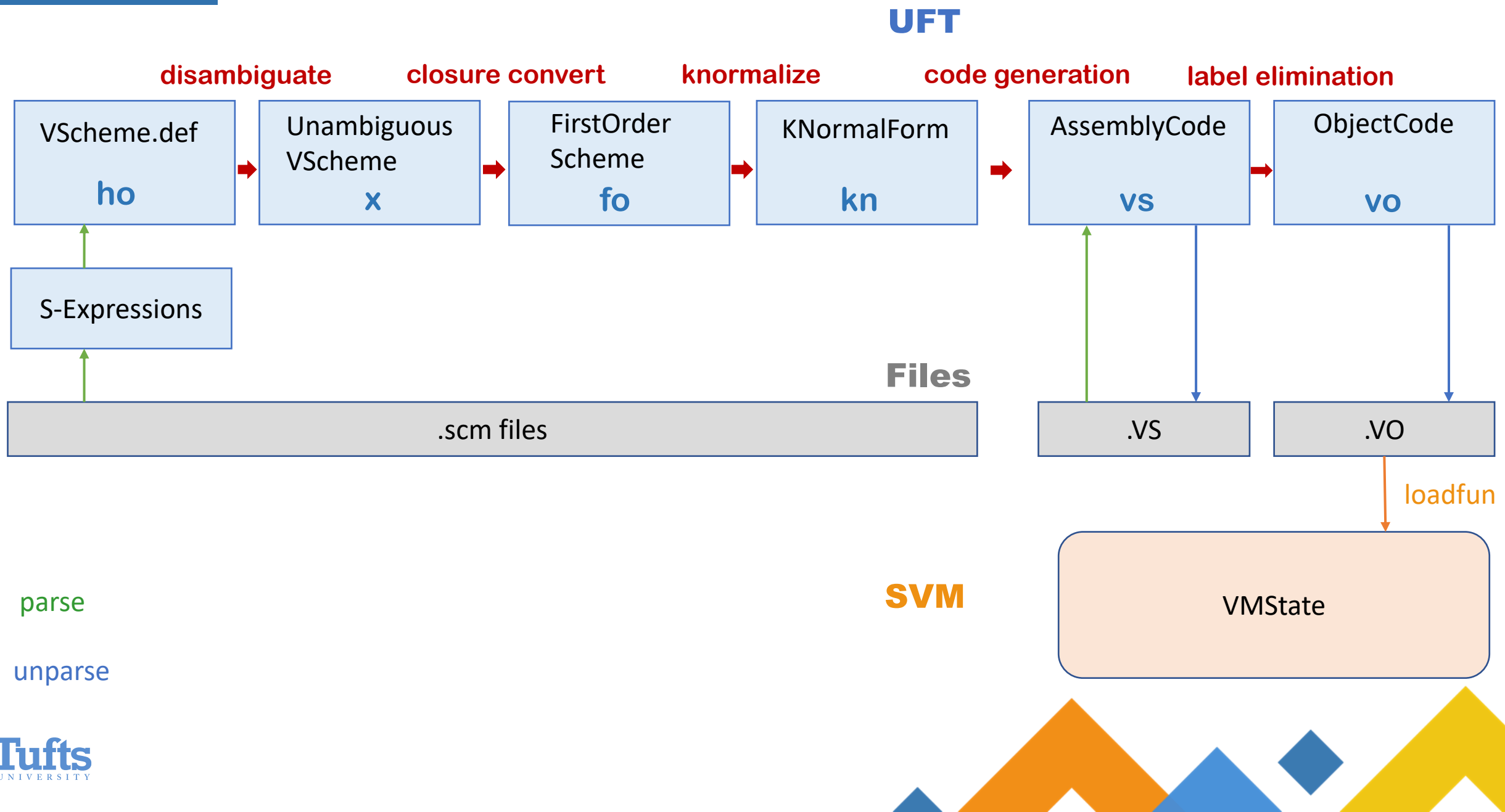
Tufts
UNIVERSITY

CS106 Final Workshop

- Matt Zhou

Overview



Tufts UNIVERSITY

Highlights

- Error Monad and Parsing Combinators
- Garbage Collection
- Stack Tracing 

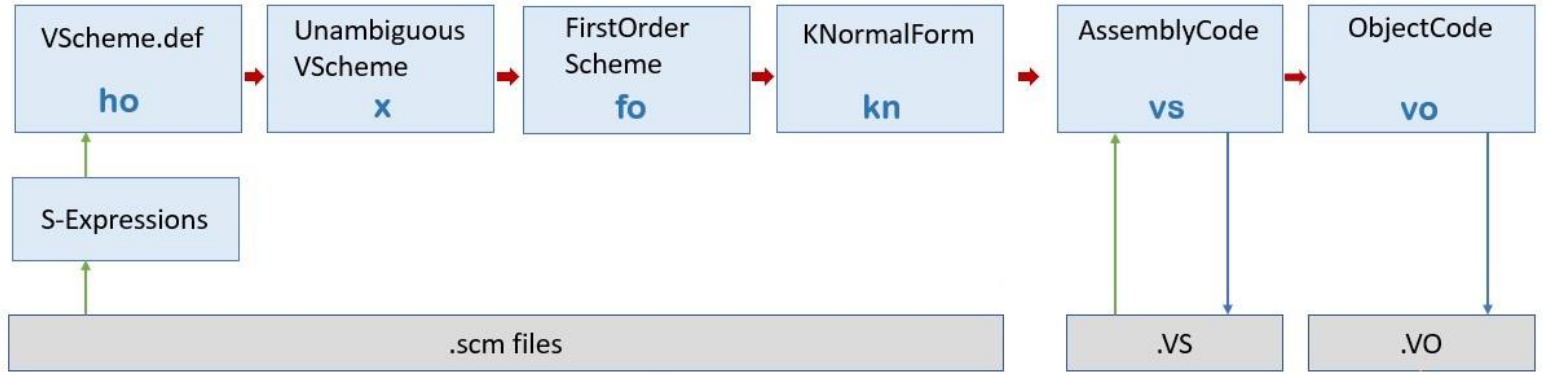
Stack Tracing

Goal:

- When a run-time error occurs in the svm, produce a stack trace that gives the user the source code location of the error

```
Process terminating with default action of signal 6 (SIGABRT)
   at 0x5217AFF: raise (in /usr/lib64/libc-2.28.so)
   by 0x51EAEA4: abort (in /usr/lib64/libc-2.28.so)
   by 0x401AEB: Except_raise (except.c:30)
   by 0x40152E: retrieve (retrieve.c:49)
   by 0x400E7E: restore (restoration.c:84)
   by 0x400D5D: main (restoration.c:37)
```

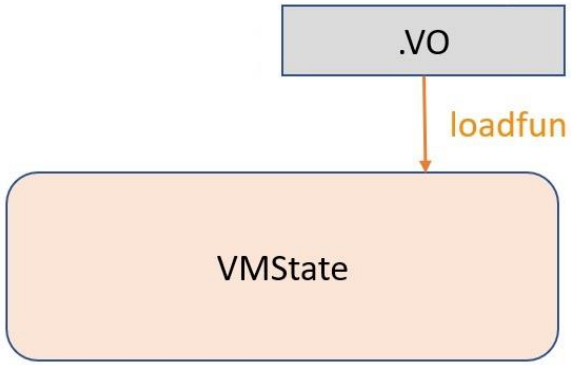

Stack Tracing



Hurdles (uft):

- Source code location (file name & line number)
  - Get the filename
  - Get line number for call sites from the file
  - Propagate the locations through all the passes (ho-vo)
- Function names
  - Keep the function name when translating to K-normal form
  - Propagate the name down the line to vo

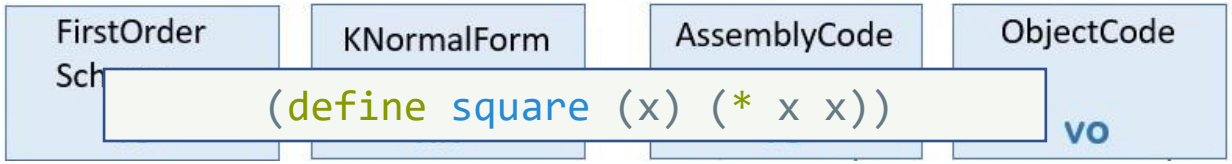
Stack Tracing



Hurdles (svm):

- Load function names and source code locations from vo
- Given a function, know its name and the source code location of its instructions
- Retrieve location based on instruction

Stack Tracing



Background (uft - Function names):

```
structure ClosedScheme = struct
(*closed first-order scheme*)
datatype exp = ...
             | CLOSURE of funcode * exp list

datatype def = ...
             | DEFINE of name * funcode
```

```
structure KNormalForm = struct

datatype 'a exp
    = ...
    | FUNCODE of 'a list * 'a exp
```

```
structure AssemblyCode = struct

datatype instr
    = ...
    | LOADFUNC of O.reg * int * instr list
```

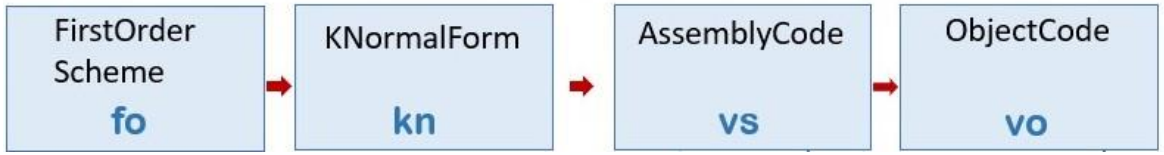
```
(let ([r0 (lambda (r1) (* r1 r1))])
                    (set square r0))
```

```
.loadfun r0 1
        r0 := r1 * r1
        return r0
el
%square := r0
```

Tufts UNIVERSITY

Stack Tracing

Function Names



- Change FUNCODE representation in knf

```
FUNCODE of 'a list * 'a exp  ⇨  FUNCODE of name * 'a list * 'a exp
```

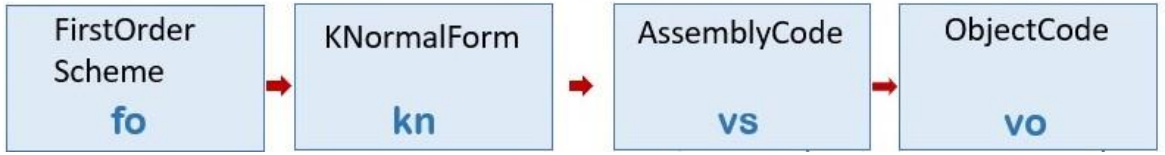
- Change LOADFUNC representation in asm & obj

```
LOADFUNC of O.reg * int * instr list  ⇨  LOADFUNC of O.reg * int * name * instr list
```

- Now we preserve function names when translating

```
(C.DEFINE (name, funcode)) => K.FUNCODE (name, normalize funcode)
(C.CLOSURE (funcode, captured)) => K.FUNCODE ("(anonymous_fun)", …)
```

Stack Tracing



Function Names

```
(define square (x) (* x x))
```

KNormal form:

```
FUNCODE of 'a list * 'a exp

(let ([r0 (lambda (r1) (* r1 r1))])
                    (set square r0))
```

```
FUNCODE of name * 'a list * 'a exp

(let ([r0 (lambda (r1) (* r1 r1))])
                    (set square r0))
```

Assembly:

```
LOADFUNC of O.reg * int * instr list

.loadfun r0 1
        r0 := r1 * r1
        return r0
el
%square := r0
```

```
LOADFUNC of O.reg * int * name * instr list

.loadfun r0 1 square
        r0 := r1 * r1
        return r0
el
%square := r0
```

Tufts UNIVERSITY

Stack Tracing

Background (svm - Function names):

- Function representation

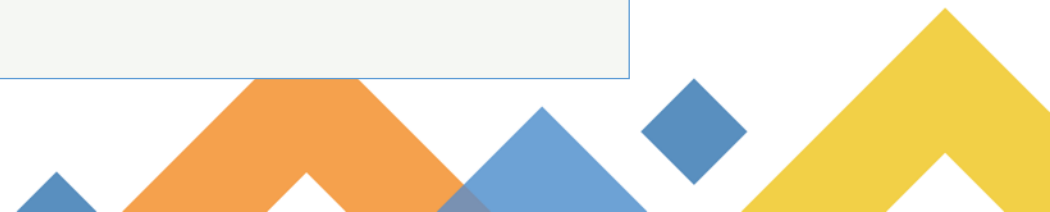
```
struct VMFunction {
  int arity; // number of args expected
  int size;  // number of instructions
  int nregs; // …
  Instruction instructions[];
};
```

```
.load module 1     (v number of instructions)
.load 0 function 1 2
                  (^ arity)
* 0 1 1
return 0
```

- Add name field to function

```
struct VMFunction {
  Name funname;

  int arity; // number of args expected
  int size;  // number of instructions
  int nregs; // …
  Instruction instructions[];
};
```

```
.load module 1
.load 0 function square 1 2
* 0 1 1
return 0
```



Tufts UNIVERSITY

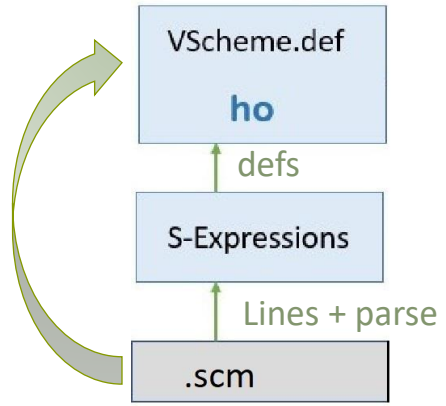
Stack Tracing

Background (uft - Source code loc):

```
val lines : TextIO.instream -> string list
```

```
val parse : string list -> Sx.sx list Error.error
```

```
val defs : Sx.sx -> VScheme.def list Error.error
```




```
val schemeOfFile :
instream ->
VScheme.def list error
```

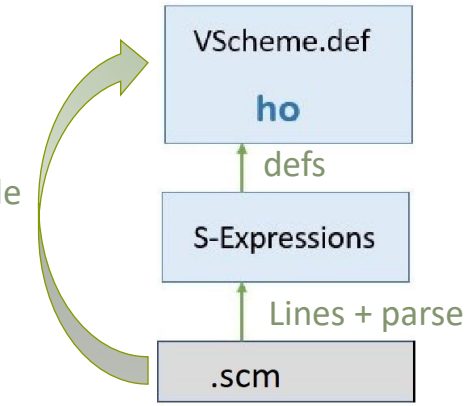
Stack Tracing

Background (uft - Source code loc):

```
val lines : TextIO.instream -> string list
```

```
val parse : string list -> Sx.sx l
```

```
val defs : Sx.sx -> VScheme.def li
```




```
structure VScheme = struct

 type name = string

 datatype exp
  = LITERAL of value
  | VAR     of name
  | SET     of name * exp
  | IFX     of exp * exp * exp
  | WHILEX  of exp * exp
  | BEGIN   of exp list
  | APPLY   of exp * exp list
  | LETX    of let_kind * (name * exp) list * exp
  | LAMBDA  of lambda
```

Tufts UNIVERSITY

Stack Tracing

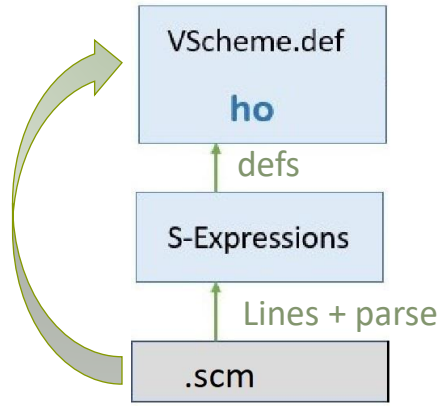
Background (uft - Source code loc):

```
val lines : TextIO.instream -> string list
```

```
val parse : string list -> Sx.sx list Error.error
```

```
val defs : Sx.sx -> VScheme.def list Error.error
```

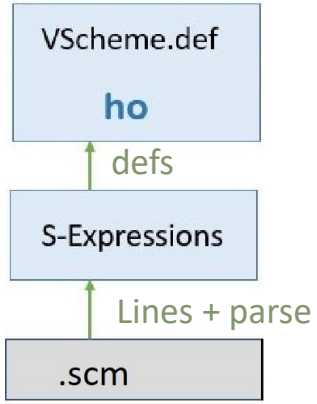
```
Language Specific Projection Functions
```




```
val schemeOfFile :
instream ->
VScheme.def list error
```

Stack Tracing

Srcloc



New types for srcloc:

```
type srcloc = string * int
        (* file name * line number *)
type 'a located = 'a * srcloc
```

Extend parser with the new types:

```
val lines : TextIO.instream -> string list
```

```
val locatedLines  : string -> TextIO.instream -> string located list
```

Accordingly:

```
val parse : string list -> Sx.sx list Error.error
```

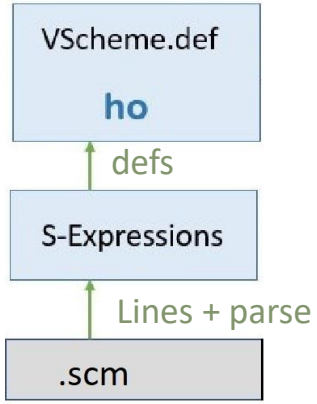
```
val parse : string located list -> Sx.sx list Error.error
```

Tufts UNIVERSITY

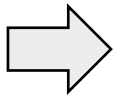
Stack Tracing

Srcloc

```
val parse : string located list -> Sx.sx list Error.error
```




```
datatype sx
  = INT  of int
  | BOOL of bool
  | SYM  of string
  | REAL of real
  | LIST of sx list
```

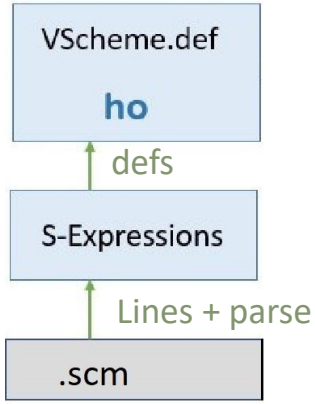


```
type srcloc = string * int
type 'a located = 'a * srcloc

  datatype sx
    = INT  of int
    | BOOL of bool
    | SYM  of string
    | REAL of real
    | LIST of sx list
    | MARK of sx located
```

Tufts UNIVERSITY

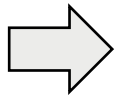
Stack Tracing

Srcloc




```
val defs : Sx.sx -> VScheme.def list Error.error
```

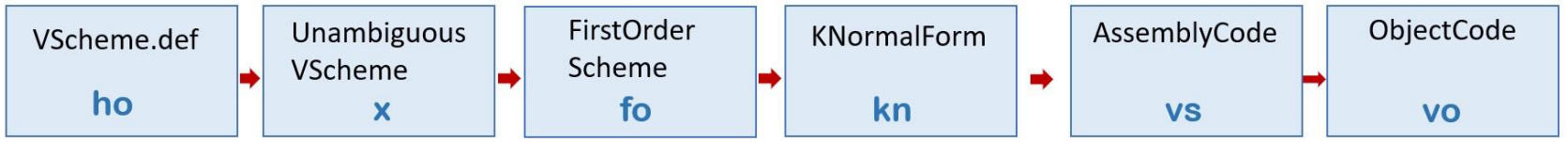
```
structure VScheme = struct

 type name = string

 datatype exp
  = LITERAL of value
  | VAR     of name
  | SET     of name * exp
  | IFX     of exp * exp * exp
  | WHILEX  of exp * exp
  | BEGIN   of exp list
  | APPLY   of exp * exp list
   …
```



```
structure VScheme = struct

 type name = string

 datatype exp
  = LITERAL of value
  | VAR     of name
  | SET     of name * exp
  | IFX     of exp * exp * exp
  | WHILEX  of exp * exp
  | BEGIN   of exp list
  | APPLY   of exp * exp list
  | MAPPLY  of srcloc * exp * exp list
   …
```

Tufts UNIVERSITY

Stack Tracing



Ripple downstream

- x & fo

```
datatype exp
  = FUNCALL   of exp * exp list
  | MFUNCALL  of srcloc * exp * exp list
  | PRIMCALL  of P.primitive * exp list
  | MPRIMCALL of srcloc * P.primitive * exp list
  …
```

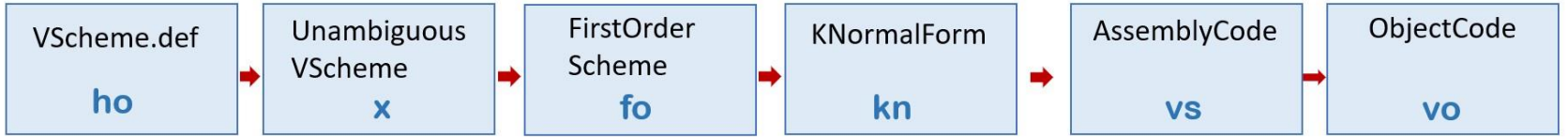
- kn

```
datatype 'a exp
  = MVMOP of srcloc * vmop * 'a list
  | MVMOPL of srcloc * vmop * 'a list * literal
  | MFUNCALL of srcloc * 'a * 'a list
  …
```

Tufts UNIVERSITY

Stack Tracing

Srcloc

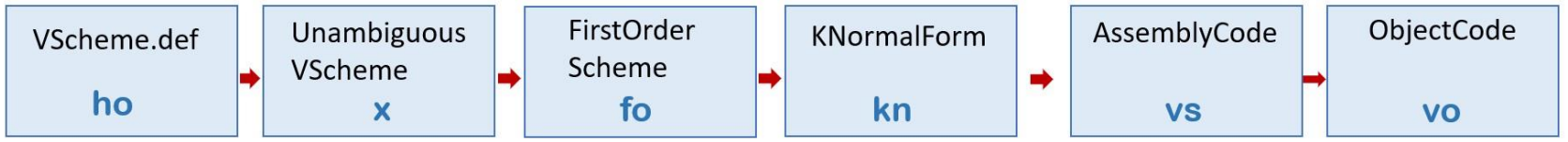


- vs & vo

```
datatype instr
    = REGS     of operator * reg list
    | MREGS    of srcloc * operator * reg list
    | REGSLIT  of operator * reg list * literal
    | MREGSLIT of srcloc * operator * reg list * literal
    …
```

Stack Tracing

Srcloc



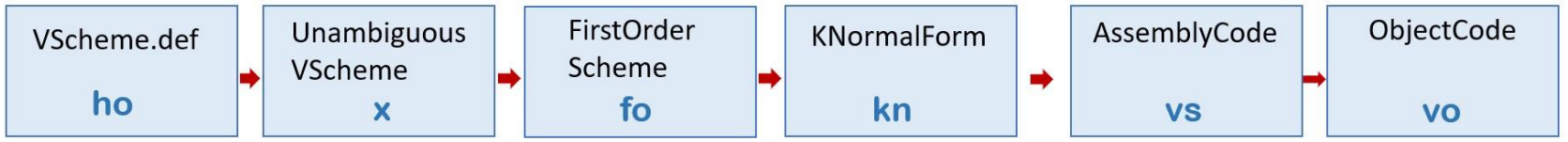

```
File: sum_squares.scm

1 (define square (x)
2     (* x x))
3
4 (define sum_squares (x y)
5     (+ (square x)
6        (square y)))
7
8 (sum_squares 10 2)
```

ho-vs

```
.loadfun r0 1 square
    r0 := r1 * r1 (sum_squares.scm - 2)
    return r0
el
%square := r0
.loadfun r0 2 sum_squares
    r3 := %square
    r4 := r1
    r3 := call r3 (r4, ..., r4) (sum_squares.scm - 5)
    r4 := %square
    r5 := r2
    r4 := call r4 (r5, ..., r5) (sum_squares.scm - 6)
    r0 := r3 + r4 (sum_squares.scm - 5)
    return r0
el
%sum_squares := r0
r0 := %sum_squares
r1 := 10
r2 := 2
r0 := call r0 (r1, ..., r2) (sum_squares.scm - 8)
```

Tufts UNIVERSITY

Stack Tracing




```
.loadfun r0 1 square
    r0 := r1 * r1 (sum_squares.scm - 2)
    return r0
el
%square := r0
.loadfun r0 2 sum_squares
    r3 := %square
    r4 := r1
    r3 := call r3 (r4, ..., r4) (sum_squares.scm - 5)
    r4 := %square
    r5 := r2
    r4 := call r4 (r5, ..., r5) (sum_squares.scm - 6)
    r0 := r3 + r4 (sum_squares.scm - 5)
    return r0
el
%sum_squares := r0
r0 := %sum_squares
r1 := 10
r2 := 2
r0 := call r0 (r1, ..., r2) (sum_squares.scm - 8)
```

vs-vo

```
.load module 8
.load 0 function square 1 2
.at sum_squares.scm 2 * 0 1 1
return 0
setglobal 0 string 6 ...
.load 0 function sum_squares 2 8
getglobal 3 string 6 ...
copy 4 1
.at sum_squares.scm 5 call 3 3 4
getglobal 4 string 6 ...
copy 5 2
.at sum_squares.scm 6 call 4 4 5
.at sum_squares.scm 5 + 0 3 4
return 0
setglobal 0 string ...
getglobal 0 string ...
loadliteral 1 10
loadliteral 2 2
.at sum_squares.scm 8 call 0 0 2
```

Stack Tracing

Background (svm - srcloc):

- Activation Record representation

```
struct Activation {
// all relative to caller
    ...
    struct VMFunction *fun;
    int return_to;
}
```

- Function representation

```
struct VMFunction {
  ...
  int size; // number of instructions
  ...
  Instruction instructions[];
};
```

- Stack representation

```
struct VMState {
    struct VMFunction* curr_prog;
    ...
    int num_activations;
    struct Activation stack[STACK_SIZE];
};
```

- Error generation

```
void typeerror(VMState state,
const char *expected, Value got, ...);

void runerror(VMState state,
const char *format, ...);
```

Tufts UNIVERSITY

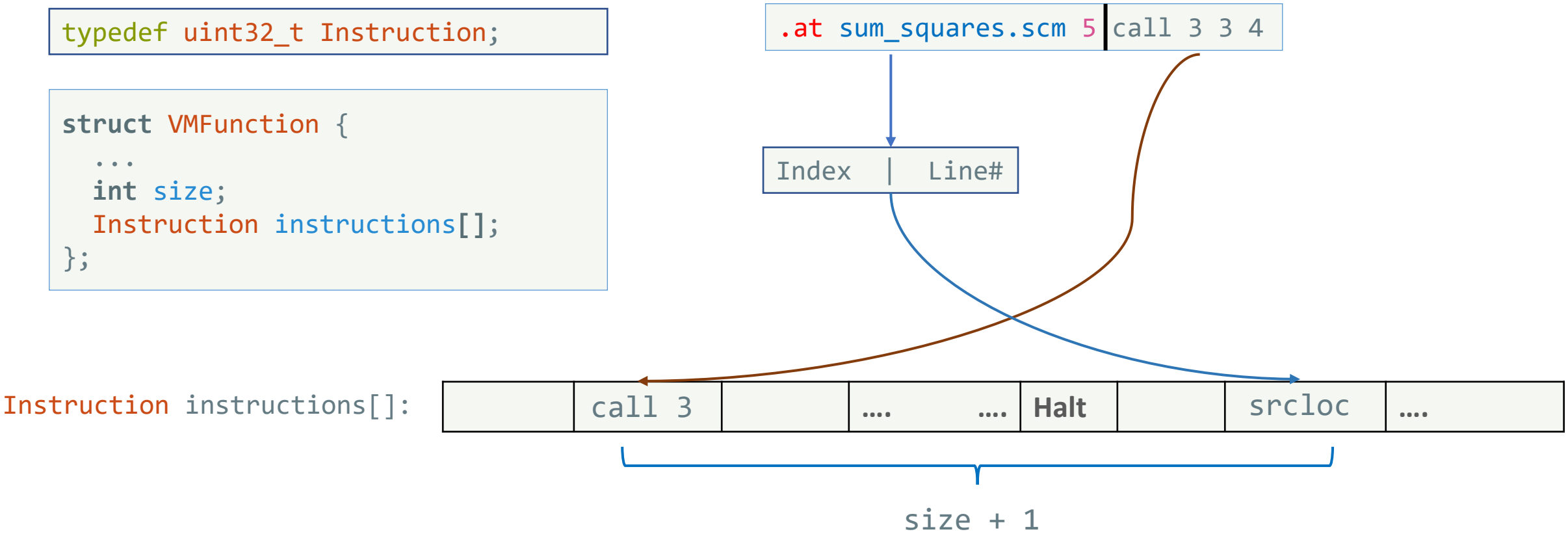
Stack Tracing

Srcloc

- Store code location inside the function

```
typedef uint32_t Instruction;
```

```
struct VMFunction {
  ...
  int size;
  Instruction instructions[];
};
```



Stack Tracing

Srcloc

- Change in function representation

```
struct VMFunction {
  GCMETA(VMFunction)
  Name funname;

  int arity; // number of args expected
  int num_insturctions; // number of instructions in the function
  int nregs; // one more than the number of highest register read or written
  Instruction instructions[];
};
```


Stack Tracing

Background (svm - srcloc):

- Activation Record representation

```
struct Activation {
// all relative to caller
    ...
    struct VMFunction *fun;
    int return_to;
}
```

- Stack representation

```
struct VMState {
    struct VMFunction* curr_prog;
    ...
    int num_activations;
    struct Activation stack[STACK_SIZE];
};
```

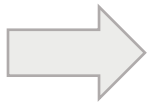
Tufts UNIVERSITY

Stack Tracing

Srcloc

- Changes in Activation record

```
struct Activation {
// all relative to caller
    ...
    struct VMFunction *fun;
    int return_to;
}
```



```
struct Activation {
    /*caller*/
    ...
    int return_to;
    /*callee*/
    struct VMFunction *fun;
    /*tailcall identifier*/
    uint8_t tailcall_info;
};
```

- Changes on the Stack

```
struct VMState {
    struct VMFunction* curr_prog;
    ...
    int num_activations;
    struct Activation stack[STACK_SIZE];
};
```

Tufts UNIVERSITY

Stack Tracing

Srcloc

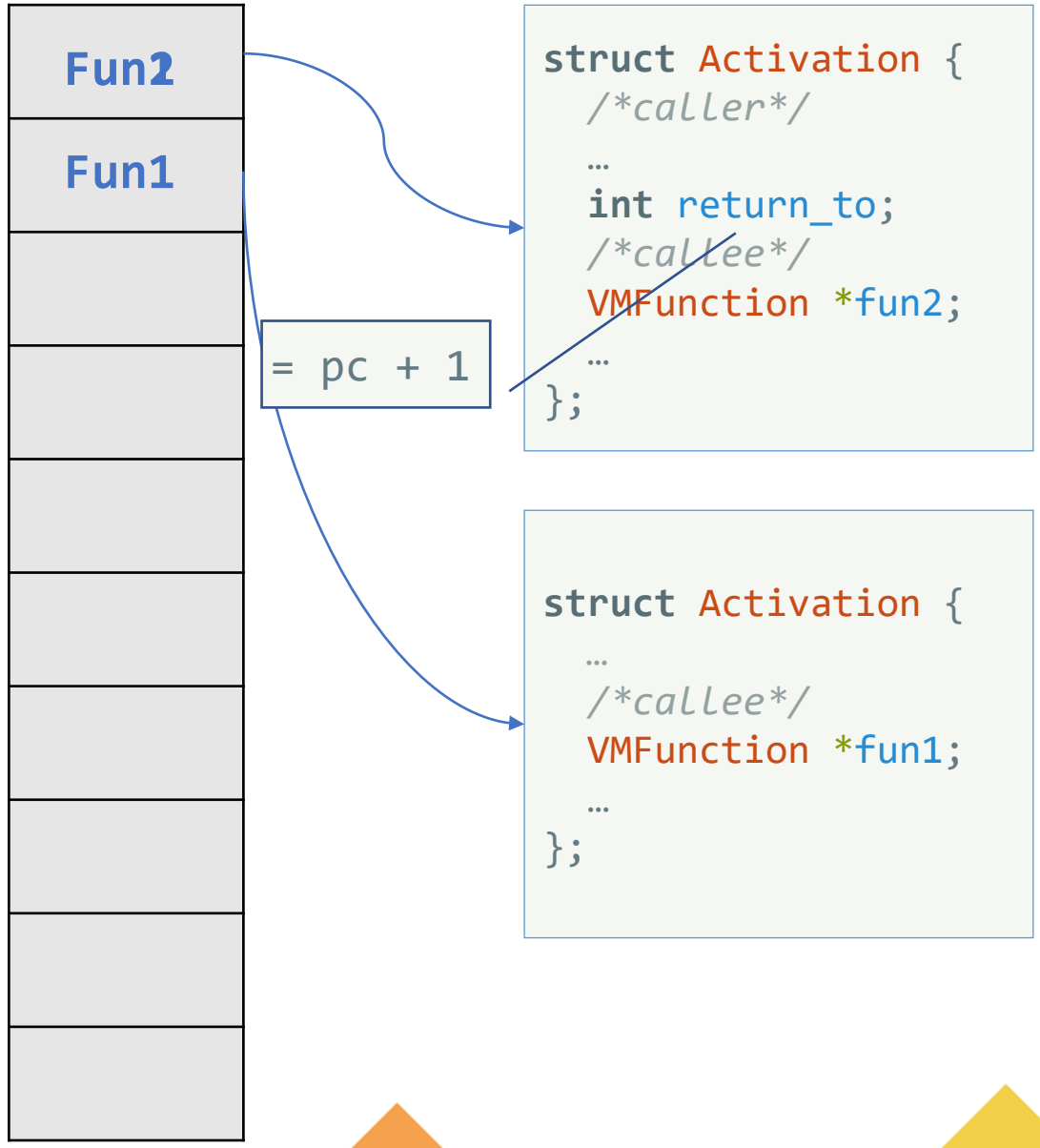
- Getting the call sites
  - Allows us to index into the Instructions array to retrieve the location

Call site?

Fun1

Fun2

We need the current PC!




```
struct VMFunction {
  Name funname;

  int arity;
  int num_insturctions;
  int nregs;
  Instruction instructions[];
};
```

Tufts UNIVERSITY

Stack Tracing

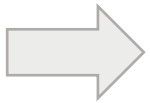
Srcloc

- Getting the call sites
  - Allows us to index into the Instructions array to retrieve the location

```
if (num_args != arity) {
 runerror(vm, pc, "fun %s expected %d args, but got %d args",
          nametostr(newfun->funname), arity, num_args);
}
```

- Error generation

```
void typeerror(VMState state,
const char *expected, Value got, ...);

void runerror(VMState state,
const char *format, ...);
```



```
void typeerror(VMState state, int pc,
const char *expected, Value got, ...);

void runerror(VMState state, int pc,
const char *format, ...);
```

Stack Tracing

Demos

File: unzip_ho.error

```
1 (define list1 (x)     (cons x '()))
2 (define list2 (x y)   (cons x (list1 y)))
3 (define cadr (xs)
4        (car
5            (cdr xs)))
6 (define map (f xs)
7   (if (null? xs)
8     `()
9     (cons (f (car xs))
10          (map f (cdr xs)))))
11
12 (define unzip (ps)
13     (if (null? ps)
14         '(() ())
15         (list2 (map car ps)
16                (map cadr ps))))
17
18
19 (check-expect (unzip '((I Magnin) (U ) (E Coli)))
20               '((I U E) (Magnin Thant Coli)))
```

Stack Tracing

Demos

```
File: tailcall_ho.error

1 (define child3 (x)
2    (car x))
3
4 (define child2 (x)
5    (child3 x))
6
7 (define child1 (x)
8    (child2 x))
9
10 (define mother (x)
11     (child1 x))
12
13 (mother 5)
```

```
struct Activation {
    /*caller*/
    int dest;
    Value* window_start;
    int return_to;
    /*callee*/
    struct VMFunction *fun;
    /*tailcall identifier*/
    uint8_t tailcall_info;
};
```

Tufts UNIVERSITY

Stack Tracing

Reflections

- **What can I lean on when writing code**
  - uft (sml) => Types
  - svm (c) => Invariants

Stack Tracing

Reflections

- **Types**
  - Types I have
  - Types I want
  - Rest should fall in naturally

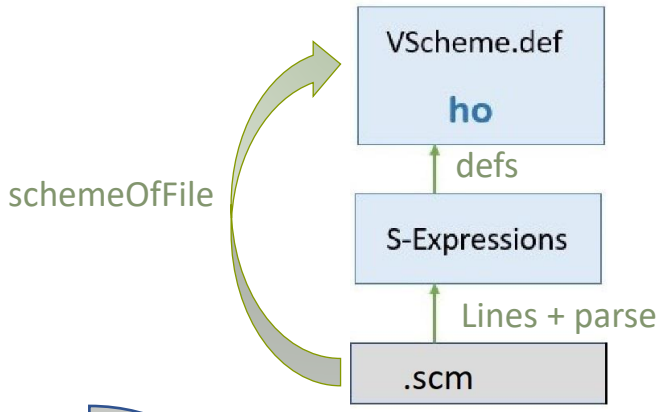
Stack Tracing

Reflections

- **Types**

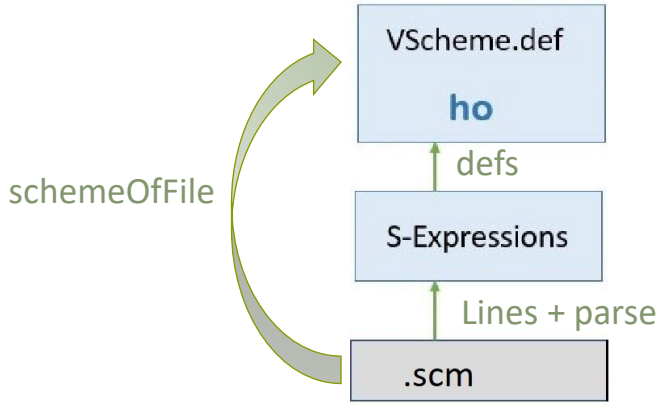
```
val parse : string list -> Sx.sx list Error.error
```

```
val parse : string located list -> Sx.sx list Error.error
```



Stack Tracing

Reflections



- **Types**

```
val parse : string list -> Sx.sx list Error.error =
    emap L.tokenize_line >> ! List.concat >=> ...
```

```
token list list error
```

```
val concat = 'a list list -> 'a list
```

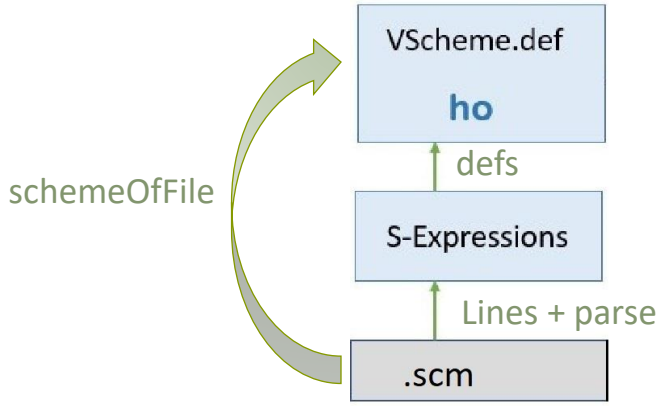
```
val tokenize_line : string -> token list Error.error
```

```
fun emap f = map f >> Error.list  where
val Error.list : 'a error list -> 'a list error
```

Tufts UNIVERSITY

Stack Tracing

Reflections



- **Types**

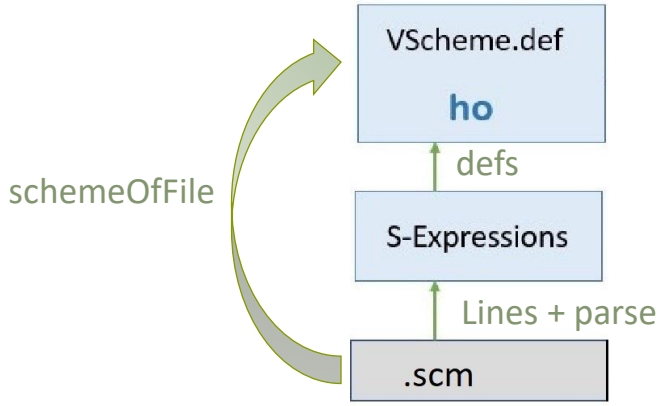
```
val parse : string located list -> Sx.sx list Error.error =
    emap L.tokenize_line >> ! List.concat >=> ...
```

```
token located list list error
```

```
val tokenize_line : string -> token list Error.error
```

Stack Tracing

Reflections



- **Types**

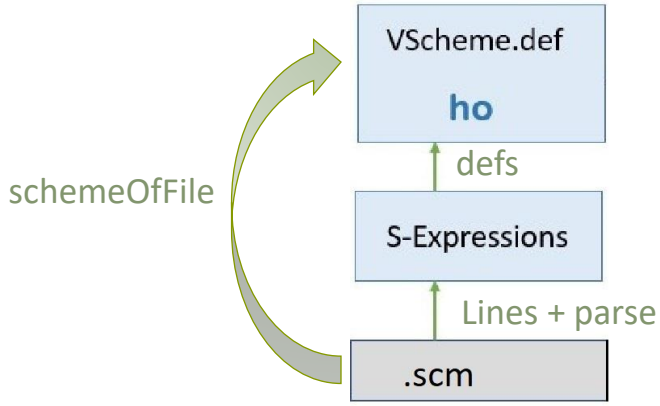
```
val parse : string located list -> Sx.sx list Error.error =
    emap L.tokenize_line >> ! List.concat >=> ...
```

```
token located list list error
```

```
val _ : string located -> L.token located list Error.error
```

Stack Tracing

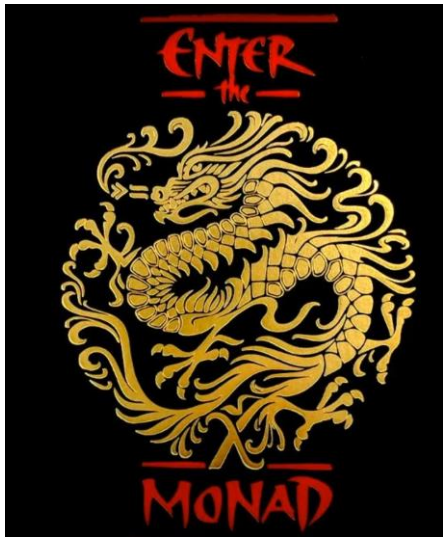
Reflections



- **Types**

```
val parse : string located list -> Sx.sx list Error.error =
    emap L.tokenize_line >> ! List.concat >=> ...
```

```
token located list list error
```




```
val _ : string located -> L.token located list Error.error
```

```
fun tokWithLoc (line, loc) =
   let fun mktlpair tks = succeed (map (fn t => (t, loc)) tks)
   in
     L.tokenize_line line >>= mktlpair
   end
```

Stack Tracing

Reflections

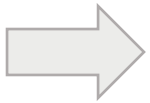
- **Invariants**
  - Am I changing any invariant?
  - Am I obeying all relevant invariants?

Stack Tracing

Reflections

- Changes in Activation record

```
struct Activation {
// all relative to caller
    int dest;
    Value* reg_window;
    struct VMFunction *fun;
    int return_to;
}
```



```
struct Activation {
    /*caller*/
    int dest;
    Value* window_start;
    int return_to;
    /*callee*/
    struct VMFunction *fun;
    /*tailcall identifier*/
    uint8_t tailcall_info;
};
```

- Changes on the Stack

```
struct VMState {
    struct VMFunction* curr_prog;
    ...
    int num_activations;
    struct Activation stack[STACK_SIZE];
};
```

INVARIANT: current function is always at the "top" of the stack

Tufts UNIVERSITY

Stack Tracing

Reflections

```
void vmrun(VMState vm, struct VMFunction *fun) {
    . . .
    struct VMFunction *curr_fun = fun;
    // INVARIANT: current function is always at the "top" of the stack
    struct Activation new_Act = {-1, reg0, pc+1, curr_fun, 0};
    vm->stack[vm->num_activations++] = new_Act;
    . . .

        case Call:{
                ...
                struct VMFunction* newfun = …;
                // push the new function on stack, maintain
                 invaraint since we're running it
                struct Activation new_Act = {…, newfun, …};
                vm->stack[vm->num_activations++] = new_Act;
                ...
            }
    . . .
 }
```

Tufts UNIVERSITY

Stack Tracing

Reflections

```
        case Call:{
                ...
                struct VMFunction* newfun = …;
                // push the new function on stack, maintain
                 invaraint since we're running it
                struct Activation new_Act = {…, newfun, …};
                vm->stack[vm->num_activations++] = new_Act;
                ...
            }
        case Tailcall:{

                struct VMFunction* newfun = ...;
                ...
                /* updating fun while we go, maintain invariant */
                struct Activation top_act =
                 vm->stack[(vm->num_activations -1)];
                top_act.fun = newfun;
                ...
                vm->stack[(vm->num_activations - 1)] = top_act;
            }
}
```

Tufts UNIVERSITY

Stack Tracing

Reflections

- **Invariants**
  - Am I changing invariants?
  - Am I obeying all relevant invariants?
  - Safely make changes without introducing bugs

Thank you!

Tufts
UNIVERSITY