

CMSC 430 - 8 June 2026

Variables, binding, lexical addresses, and the stack

- Extort - Specifying error semantics; runtime support for errors
- Fraud - Variables and let expressions

CMSC 430 - 8 June 2026

Announcements

- New quiz released, due by 10AM Tues

Extort

(runtime support for signaling errors)

Extort: Compiling with Errors

Undefined behavior considered harmful

Just as in interpreter, have to check primitive arguments:
Welcome to [DrRacket](#) version 8.14 [64-bit]
Language: racket, with debugging; memory limit: 4096 MB.

```
> (compile-op1 'add1)
```

```
(list
```

```
  (Push 'rax)
```

```
  (And 'rax 1)
```

```
  (Cmp 'rax 0)
```

```
  (Pop 'rax)
```

```
  (Jne 'err)
```

```
  (Add 'rax 2))
```

```
>
```

Jumps to place that signals an error if the thing in rax is not an integer

```
> (exec (compile (parse '(add1 #f))))
```

Extorting the Compiler with Errors

Undefined behavior considered harmful

compile.rkt:28:2: match: no match
(Extern 'peek_byte) (Extern 'read_byte)

(Label 'entry) (Sub 'rsp 8) (Mov 'rax 7)
Just as in interpreter, have to check primitive arguments:

'rax) (Jne 'err) (Add 'rax 2) (Add 'rsp

```
> (exec (parse '(add1 #f)))
```

```
'err
```

```
>
```

Extort: Compiling with Errors

Undefined behavior considered harmful

```
3 compile e)
4
5 (require "../syntax/ast.rkt")
6 (require "compile-ops.rkt")
7 (require "../runtime/types.rkt")
8 (require a86/ast a86/registers)
9
10 ;; Expr -> Asm
11 (define (compile e)
12   (prog (Global 'entry)
13         (Label 'entry)
14         (Sub rsp 8)
15         (compile-e e)
16         (Add rsp 8)
17         (Ret))
18   ;; Error handler
19   (Label 'err)
20   (Extern 'raise_error)
21   (Call 'raise_error)))
22
23 ;; Expr -> Asm
```


Run-time system support

raise_error in standalone run-time system

```
void raise_error(void)
{
    printf("err\n");
    exit(1);
}
```

```
e (asm-interp/host asm)
```

```
parameterize #lang racket
```

```
([current-extend exec exec/io)
```

```
(list (require 'compiler/compile.rkt) -> _int64))
```

```
4 (require 'decode.rkt) peek-byte (_fun -> _int64))
```

```
5 (require 'host.rkt) prim-write-byte (_fun _int64 -> _int64))
```

```
6 (extern 'raise_error (λ () (raise 'err)) (_fun -> _void))))]
```

```
asm-interp asm
```

```
8
```

```
9 ;; Expr -> Answer
```

```
10 (define (exec e)
```

```
11   (with-handlers ([ (λ (x) (eq? x 'err)) identity ])
```

```
12     (bits->value (asm-interp/host (compile e))))))
```

```
13
```

```
14
```

```
15
```

```
16 ;; Asm String -> (cons Answer String)
```

```
17 (define (asm-interp/host asm)
```


Fraud

Learning objectives

Fraud

You should be able to answer:

- How are Fraud programs represented abstractly?
- How is the meaning of a variable determined?
- What changes in the interpreter with the addition of variables?
- What invariant can be observed about variable occurrences and their position in the environment?
- How can this invariant be used to compile variable occurrences?

Learning objectives

Fraud

You should be able to answer (cont'd):

- What role does the stack play in computing the meaning of variables at run-time?
- What role does the compile-time environment play in the compilation of variable occurrences?
- What is the relationship between the compile-time environment and the run-time stack?
- What's the difference between a static and a dynamic property?

Recipe for growing a language

- Write examples
- Extend concrete syntax
- Extend abstract syntax
- Extend parser
- Revise interpreter to specify semantics
- Revise compiler & run-time system to implement semantics
- Test against examples
- *Rinse and repeat*

Example Fraud program

Concrete syntax

```
1 | #lang racket
2 | (let ((x 40))
3 |   (add1 (add1 x)))
```

5

6 **Example** (require fraud)

7 **Abstract syntax**

(parse (let ((x 40)) (add1

8

(Let 'x (Lit 40)

10

(Prim1 'add1

11

(Prim1 'add1

12

(Var 'x))))

Eof Begin
Let Var)

Fraud AST

```
;; type Expr = ...  
;;           | (Let Id Expr Expr)  
;;           | (Var Id)  
  
;; type Id   = Symbol  
  
;; type ClosedExpr = { e ∈ Expr | e co
```

Fraud expression parser

Welcome to [DrRacket](#), version 9.2 [cs].

Language: racket, with debugging; memory limit: 4096 MB.

```
> (require fraud)
```

```
> (parse 'x)
```

```
'#s(Var x)
```

```
> (parse '(let ((x 42)) x))
```

```
'#s(Let x #s(Lit 42) #s(Var x))
```

```
> (parse-closed 'x)
```



../syntax/parse.rkt:16:18: unbound identifiers '(x)

```
> (parse-closed '(let ((x 42)) x))
```

```
'#s(Let x #s(Lit 42) #s(Var x))
```

```
>
```


Fraud program parser

Welcome to [DrRacket](#), version 9.2.1 for **Only closed expressions are well-formed programs**

Language: racket, with debugging; memory limit: 4096 MB.

```
> (require fraud)
```

```
> (parse 'x)
```

```
'#s(Var x)
```

```
> (parse '(let ((x 42)) x))
```

```
'#s(Let x #s(Lit 42) #s(Var x))
```

```
> (parse-closed 'x)
```



../syntax/parse.rkt:16:18: unbound identifiers '(x)

```
> (parse-closed '(let ((x 42)) x))
```

```
'#s(Let x #s(Lit 42) #s(Var x))
```

```
>
```

Fraud: adding variable bindings

Updated semantics

```
97 (define x 1)
98 (define means 1)
99 (define ... 1)
100
101 ;; Expr -> Answer
102 (define (interp-e e)
103   (match e
104     ;; ...
105     [(Var x) (... x ...)]
106     [(Let x e1 e2)
107      (... x
108        (interp-e e1)
109        (interp-e e2) ...)]))
110
111 (interp-e (Var 'x)) ;=> 42
```



What do variables mean?

It varies

```
(match (eq? x y)
  (#t 0)
  (#f (+ 8 (lookup x rest))))

(define (x) 42)
(define means 1)
(define interp-e 1)

(let ((x 10))
  (interp-e (var 'x)))
(interp-e (var 'x)))
```

```
(let ((x 42))
  (... x ...))
```

```
(let ((x 10))
  (... x ...))
```

Not a function!

42

10

;; Expr -> Answer

```
(define (interp-e
```


Need for an environment

The meaning of a variable depends on the environment in which it occurs

- It's not enough to specify the meaning of an expression in isolation
- Need to know the meaning of variables that occur in that expression
- Meaning of an expression is a function of:
 - the expression
 - the meaning of its (free) variables

What do variables mean?

It varies

24

(define (let (x 42) ...))

25

26

27

28

29

30

31

(let ((x 42))

(... x ...))

(let ((x 10))

(... x ...))

A function!

means 42 }) ;=> 42

means 10 }) ;=> 10

:: Expr -> Answer

(define (interp (Mem ...))

```

22 4 (define y 'y) 26 { x means 42
23 5 (define z 'z) 27 { y means 12
24 6 (define means 'means) 28 { z means #f }
Using an association list
25 7 ;; type Value =
26 8 ;; | Integer
27 9 { x means 42
28 10 ;; | Boolean
29 11 ;; | Character
30 12 ;; | Eof
31 13 { z means #f }
32 14 ;; | Void
33
34 15 ;; type Env = (Listof (List Id Value))
35
36 16 (let ((x 42))
37 17 ;; Expr -> Answer

```


11	:: Eof	23	(define (ext r x v) ...)
12	:: Void	24	
13	Operations on the environment	25	
14	Using an association list	26	(lookup '((x 42) (y 12) (z #f)))
15		27	(lookup '((x 42) (y 12) (z #f)))
16		28	(ext '((x 42) (y 12) (z #f)) 'v
17	:: Lookup up x's value in r	29	;=> '((v 0) (x 42) (y 12))
18	:: Env Id -> Value	30	
19	(define (lookup r x) ...)	31	:: Fundamental law:
20		32	:: ∀ r x v,
21	:: Extend r so that x means v	33	(lookup (ext r x v) x) ≡ v
22	:: Env Id Value -> Env	34	
23	(define (ext r x v) ...)	35	
24		36	:: Expr -> Answer
25		37	#;
26	(lookup '((x 42) (y 12) (z #f)))	38	(define (interp e)
27	(ext '((x 42) (y 12) (z #f)) 'v)	39	;=> (interp-env e '())
28	(ext '((v 0) (x 42) (y 12)))	40	
29	(define (= 'v)	41	(define (= 'v)

8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32

```
;; | Integer  
;; | Boolean  
;; | Character  
;; | Eof  
;; | Void
```

Interpreting variables and bindings

Environments tell us the meaning of variables

The interpreter uses an environment to manage associations between variables and their values.

```
(define (err? x) (eq? x 'err))
```

```
#lang racket  
(provide (let ([Prim0 Prim1 Prim2 If  
                Eof Begin  
                Let Var])  
  (with-handlers ([err? identity])  
    (interp-e e '()))))  
  
;; type Expr = ...  
;;           | (Let Id Expr Expr)  
;;           | (Var Id)  
  
;; type Id = Symbol  
  
;; type ClosedExpr =  
;;   { e ∈ Expr | e contains no free variables }
```

40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67

```
;; type Env = (Listof (List Id Value))  
  
;; Expr Env -> Value { raises 'err }  
(define (interp-e e r) ;; where r closes e  
  (match e  
    [(Var x) (lookup r x)]  
    [(Let x e1 e2)  
     (let ((v (interp-e e1 r)))  
       (interp-e e2 (ext r x v)))]  
    [#;...))  
  
#|  
[(Lit d) d]  
[(Eof) eof]  
[(Prim0 n)
```


Compiling variables and bindings

Using the interpreter the guide us

The interpreter uses an environment to manage associations between variables and their values.

Appears we need run-time representation of environments, identifier names, and implementation of `lookup` and `ext` in assembly. Seems... hard.

```
;; type Env = (Listof (List Id Value))

;; Expr Env -> Value { raises 'err }
(define (interp-e e r) ;; where r closes e
  (match e
    [(Var x) (lookup r x)]
    [(Let x e1 e2)
     (let ((v (interp-e e1 r)))
       (interp-e e2 (ext r x v)))]
    #;...))

#|
[(Lit d) d]
[(Eof) eof]
[(Prim0 n)
```

3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

```
(define e 2)
```

```
(define e1 1)
```

```
(define e2 2)
```

Observe something about environments

The text of the program tells you a lot about the structure of the environment

```
(let ((x ...))
```

```
  (let ((y ...))
```

```
    (let ((z ...))
```

```
      ;; what do you know about the
```

```
      ;; environment used to evaluate e?
```

```
    e)))
```

```
(let ((x ...))
```

```
  (let ((y ...))
```

```
    (+ (let ((z ...))
```

```
      ;; what do you know about the
```

```
118 (compile-e (Var 'y) '(x y z)) ;=> (Mov rax (Mem
119 (compile-e (Var 'z) '(x y z)) ;=> (Mov rax (Mem
```

Observe something about environments

The text of the program tells you a lot about the structure of the environment

```
121
122 (let ((x ...))
123   (let ((y ...))
124     (begin
125       (let ((z ...))
126         ;; What do you know about the
127         ;; environment used to evaluate e1?
128         e1)
129       ;; What about e2?
130       e2)))
131
132
```


20
21
22
23
24
25
26
27
28
29
30
31
32

e1)

;; what about e2?

Observe something about environments

The text of the program tells you a lot about the structure of the environment

```
(let ((x ...))  
  (let ((y ...))  
    (let ((z ...))  
      y)))
```

;; where will y's binding
;; be in the environment?

```
y)))
```

30 Observe something about environments

31 The text of the program tells you a lot about the structure of the environment

32

```
33 (let ((x ...))
```

```
34     (let ((y ...))
```

```
35         (let ((z ...))
```

```
36             ;; where will z's binding
```

```
37             ;; be in the environment?
```

```
38 z)))
```

39

40

41

51

52 Observe something about environments

53 The text of the program tells you a lot about the structure of the environment

54

55 (let ((x ...))

56 (let ((y ...))

57 (let ((z ...))

58 ;; where will z's binding

59 ;; be in the environment?

60 x)))

61

62

63


```
118 (compile-e (Var 'y) '(x y z)) ;=> (Mov rax (R
119 (compile-e (Var 'z) '(x y z)) ;=> (Mov rax (R
```

Observe something about environments

The text of the program tells you a lot about the structure of the environment

```
121
```

```
122 (let ((x ...))
```

```
123   (let ((y ...))
```

```
124     (begin
```

```
125       (let ((z ...))
```

```
126         ...)
```

```
127       ;; Where will y's binding
```

```
128       ;; be in the environment?
```

```
129       y))))
```

```
130
```

```
131
```



```

44 13      (match e
45 14        [(Var x) (lookup r x)]
46 15        [(Let x e1 e2)
47 16          (match (interp-env e1 r)
48 17            [err err]
49 18            [v (interp-env e2 (ext r x v))]])]
50 19        #;...))

```

Observing an invariant about bindings

Using the interpreter the guide us

Suppose we get to this point in interpreting a program:

```

50 22 (interp-e (var x) "x") ((y 1) ((x 99) (p 7)) (p 7))
51 23
52 24 (define ... 0)

```

What can you say about the program surrounding this occurrence of x ?

```

53 26
54 27 (let ((p ...))
55 28 ;; type Env = (Listof (List Id Value))
56 29 (let ((x ...))
57 30 ...
58 31 ;; Expr Env -> Value { raises 'err }
59 32 (define (interp-e e r) ;; where r closes e

```

It has to have looked like this!

Generalizing the observation

The text of the program tells you a lot about the structure of the environment

For each variable occurrence, we can precisely calculate the location in the environment *before interpreting the program*.

Lookup doesn't need to be a linear search, we can compute the index of the value in the list.

Static vs dynamic properties

Variable bindings are partly static and partly dynamic

- A static property of a program can be determined without running
- A dynamic property of a program cannot be determined without running

A variables location in the environment is static

A variables value in the environment is dynamic

Static vs dynamic properties

Variable bindings are partly static and partly dynamic

```
117 (compile-e (Var 'x) '(x y z)) => (Mov rax (Mem 1))
118 (compile-e (Var 'y) '(x y z)) => (Mov rax (Mem 1))
119 (compile-e (Var 'z) '(x y z)) => (Mov rax (Mem 1))
```

```
121
122 (let ((x ...))
123   (let ((y (read-byte)))
124     (begin
125       (let ((z ...))
126         ...))
127     ;; Where will y's binding
128     ;; be in the environment?
129     y)))
```

y's binding will be the second element in the environment

y's value is unknowable without running the program

133

Key idea

Compile-time environment and run-time stack

- Compute lexical addresses at compile-time
- Use stack to hold value of bindings at run-time
- Use compile-time environment to describe binding structure
- Maintain agreement w/ compile-time environment and run-time stack

```
33 (define ... '...)
```

Compile-time environments

compile-e now takes an environment

```
38 ;; type CEnv = (Listof Id)
```

```
40 93;; Expr CEnv -> Asm
```

```
41 94(define (compile-e e c) ...)
42 95      [#f (+ 8 (lookup x rest))]))
```

```
43 96#|
44 97
45 98
46 99
47 100
48 101
49 102
50 103
```

Describes the binding context of e

```
(match e
```

```
  [(Lit d) (compile-value d)]
  [(Var x) (compile-value (x y z))]
  [(Eof) (compile-value eof)]
  [(Var x) (compile-variable (x y z))]
  [(Prim0 p) (compile-prim0 p)]
  [(Prim1 p e) (compile-prim1 p e c)]
```

```
    (Mov rax (Mem rsp 0))
    (Mov rax (Mem rsp 8))
    (Mov rax (Mem rsp 16))
```

```
    (Mov rax (Mem rsp 16))
```

```
    [(Prim0 p) (compile-prim0 p)]
```

```
    [(Prim1 p e) (compile-prim1 p e c)]
```


Compile-time environments

compile-e now takes an environment

```
;; Id CEnv -> Asm
(define (compile-variable x c)
  (let ((i (lookup x c)))
    (seq (Mov rax (Mem rsp i))))))

;; Id Expr Expr CEnv -> Asm
(define (compile-let x e1 e2 c)
  (seq (compile-e e1 c)
        (Push rax)
        (compile-e e2 (cons x c))
        (Add rsp 8)))
```

Compile-time environments

compile-e now takes an environment

```
101 (define (compile-begin e1 e2 c)
102   (seq (compile-e e1 c)
103        (compile-e e2 c)))
104
105
106
107 ;; Id CEnv -> Integer
108 (define (lookup x cenv)
109   (match cenv
110     [(cons y rest)
111      (match (eq? x y)
112        [#t 0]
113        [#f (+ 8 (lookup x rest))])])])
114
115 #|
116
117 (define x 1)
```


(Label 'err)
(Extern 'raise_error)
Success
(Call 'raise_error))

Our compiler works for programs with variables

> (exec (let ((x 40)) (add1 (add1 x))))

 *match: no matching clause for '(let ((x 40)) (add1 x))'*

> (exec (parse (let ((x 40)) (add1 (add1 x)))))

42

>



What about this?

Are we still minding the ABI requirements...

```
Welcome to DrRacket, version 9.9.10.1.
Language: racket, with debugging; memory limit: 4096 MB.
> (require fraud)
> (compile (parse '(let ((x 40)) (write-byte x))))
(list
 (Global 'entry)
 (Label 'entry)
 (Mov 'rax 80)
 (Push 'rax)
 (Mov 'rax (Mem 'rsp 0))
 (Extern 'write_byte)
 (Push 'rax)
 (And 'rax 1)
 (Cmp 'rax 0)
 (Pop 'rax)
 (Jne 'err)
 (Cmp 'rax 0)
 (Jl 'err)
 (Cmp 'rax 510)
 (Jg 'err)
 (Mov 'rdi 'rax)
 (Call 'write_byte)
 (Add 'rsp 8)
 (Ret)
 (Label 'err)
 (Extern 'raise_error)
 (Call 'raise_error))
>
```



For next time

- Study Fraud notes
- Take ELMS quiz