CMSC 430 - 22 Sept 2026

Variables, binding, lexical addresses, and the stack

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

CMSC 430 - 22 Sept 2026

Announcements

- Jury duty cont., lecture recording on ELMS

Extort

(runtime support for signaling errors)

Extort: Compiling with Errors

Undefined behavior considered harmful

Just as in interpreter, have to check primitive arguments:

```
> (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

Extort: Compiling with Errors

Undefined behavior considered harmful

Just as in interpreter, have to check primitive arguments:

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

Extort: Compiling with Errors

Undefined behavior considered harmful

```
;; Expr -> Asm
(define (compile e)
  (prog (Global 'entry)
        (Label 'entry)
        (Sub rsp 8)
        (compile-e e)
        (Add rsp 8)
        (Ret)
        ;; Error handler
        (Label 'err)
        (Extern 'raise_error)
        (Call 'raise_error)))
```

Run-time system support

raise_error in standalone run-time system

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

Run-time system support

raise_error in hosted run-time system

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

```
;; Expr -> Answer
(define (exec e)
  (with-handlers ([(λ (x) (eq? x 'err)) identity])
    (bits->value (asm-interp/host (compile e)))))
```

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

```
#lang racket
(let ((x 40))
  (add1 (add1 x)))
```

Example Fraud program

Abstract syntax

```
(Let 'x (Lit 40)
  (Prim1 'add1
    (Prim1 'add1
      (Var 'x))))
```

Fraud AST

```
;; type Expr = ...
;;           | (Let Id Expr Expr)
;;           | (Var Id)

;; type Id  = Symbol
```

Fraud expression parser

```
> (parse 'x)
'#s(Var x)
> (parse '(let ((x 42)) x))
'#s(Let x #s(Lit 42) #s(Var x))
```

Fraud program parser

Only closed expressions are well-formed programs

```
> (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

```
;; Expr -> Answer
(define (interp-e e)
  (match e
    ;; ...
    [(Var x) (... x ...)]
    [(Let x e1 e2)
     (... x
          (interp-e e1)
          (interp-e e2) ...)]))
```



What do variables mean?

It varies

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

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

Not a function!

```
(interp-e (Var 'x)) ;=> 42
(interp-e (Var 'x)) ;=> 10
```



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

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

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

A function!

```
(interp-e (Var 'x) { x means 42 }) ;=> 42
(interp-e (Var 'x) { x means 10 }) ;=> 10
```



Representing the environment

Using an association list

```
{ x means 42
  y means 12
  z means #f }
```

```
'((x 42)
  (y 12)
  (z #f))
```

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

Operations on the environment

Using an association list

```
;; Lookup up x's value in r
;; Env Id -> Value
(define (lookup r x) ...)

;; Extend r so that x means v
;; Env Id Value -> Env
(define (ext r x v) ...)
```

```
;; Fundamental law:
;; ∀ r x v,
(lookup (ext r x v) x) ≡ v
```

Interpreting variables and bindings

Environments tell us the meaning of variables

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

```
;; ClosedExpr -> Answer
(define (interp e)
  (with-handlers ([err? identity])
    (interp-e e '())))
```

```
;; type ClosedExpr =
;;       { e ∈ Expr | e contains no free variables }
```

```
;; 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)))]
    #;...))
```

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)))]
    #;...))
```

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)))
```

Observe something about environments

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

```
(let ((x ...))
  (let ((y ...))
    (begin
      (let ((z ...))
        ;; What do you know about the
        ;; environment used to evaluate e1?
        e1)
      ;; What about e2?
      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 ...))
      ;; where will y's binding
      ;; be in the environment?
      y)))
```

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 ...))
      ;; where will z's binding
      ;; be in the environment?
      z)))
```

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 ...))
      ;; where will z's binding
      ;; be in the environment?
      x)))
```

Observe something about environments

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

```
(let ((x ...))
  (let ((y ...))
    (begin
      (let ((z ...))
        ...)
      ;; Where will y's binding
      ;; be in the environment?
      y)))
```

Observing an invariant about bindings

Using the interpreter the guide us

Suppose we get to this point in interpreting a program:

```
(interp-e (Var 'x) '((y 1) (x 99) (p 7)))
```

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

```
(let ((p ...))
  ...
  (let ((x ...))
    ...
    (let ((y ...))
      ... x ...)))
```

It has to have looked like this!

Observing an invariant about bindings

Using the interpreter the guide us

Suppose we know the program looks like this:

```
(let ((p ...))
  ...
  (let ((x ...))
    ...
    (let ((y ...))
      ... x ...)))
```

What can you say about the environment that will be used when `x` is interpreted?

```
'((y ??) (x ??) (p ??))
```

It must look like this!

But now we can see that `lookup` will retrieve the second element;

The location of a variables binding in the environment is a static property

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

```
(let ((x ...))
  (let ((y (read-byte)))
    (begin
      (let ((z ...))
        ...)
      ;; Where will y's binding
      ;; be in the environment?
      y)))
```

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

y's value is unknowable without running the program

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

Compile-time environments

compile-e now takes an environment

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

;; Expr CEnv -> Asm
(define (compile-e e c) ...)
```

Describes the binding context of e

```
(compile-e (Var 'x) '(x y z)) ;=> (Mov rax (Mem rsp 0))
(compile-e (Var 'y) '(x y z)) ;=> (Mov rax (Mem rsp 8))
(compile-e (Var 'z) '(x y z)) ;=> (Mov rax (Mem rsp 16))
```

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

```
;; Id CEnv -> Integer
(define (lookup x cenv)
  (match cenv
    [(cons y rest)
     (match (eq? x y)
       [#t 0]
       [#f (+ 8 (lookup x rest))])]))
```

Success

Our compiler works for programs with variables

```
> (exec (parse '(let ((x 40)) (add1 (add1 x)))))
42
```



What about this?

Are we still minding the ABI requirements…

```
> (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