CMSC 430 - 17 Sept 2026

I/O effects; Errors

- Evildoer - I/O, system calls, ABI
- Extort - Specifying error semantics

CMSC 430 - 17 June 2026

Announcements

- Jury duty, watch summer recording on ELMS

Evildoer

Learning objectives

Evildoer

You should be able to answer:

- How are Evildoer programs represented abstractly?
- How does I/O change the meaning of programs?
- How can our host and stand-alone runtime systems provide support for I/O operations?
- What is an ABI? What is the System V ABI?
- How can we make calls to external code according to the ABI?

Syntactic additions in Evildoer

Concrete syntax	Abstract syntax
`(begin e1 e2)`	`(Begin e1 e2)`
`(read-byte)`	`(Prim0 'read-byte)`
`(peek-byte)`	`(Prim0 'peek-byte)`
`(void)`	`(Prim0 'void)`
`(write-byte e)`	`(Prim1 'write-byte e)`
`(eof-object? e)`	`(Prim1 'eof-object? e)`

Syntactic additions in Evildoer

```
;; type Expr = (Lit Datum)
;;           | (Eof)
;;           | (Prim0 Op0)
;;           | (Prim1 Op1 Expr)
;;           | (If Expr Expr Expr)
;;           | (Begin Expr Expr)
;; type Datum = Integer
;;            | Boolean
;;            | Character
;; type Op0 = 'read-byte | 'peek-byte | 'void
;; type Op1 = 'add1 | 'sub1
;;          | 'zero?
;;          | 'char? | 'integer->char | 'char->integer
;;          | 'write-byte | 'eof-object?

(struct Eof () #:prefab)
(struct Lit (d) #:prefab)
(struct Prim0 (p) #:prefab)
(struct Prim1 (p e) #:prefab)
(struct If (e1 e2 e3) #:prefab)
(struct Begin (e1 e2) #:prefab)
```

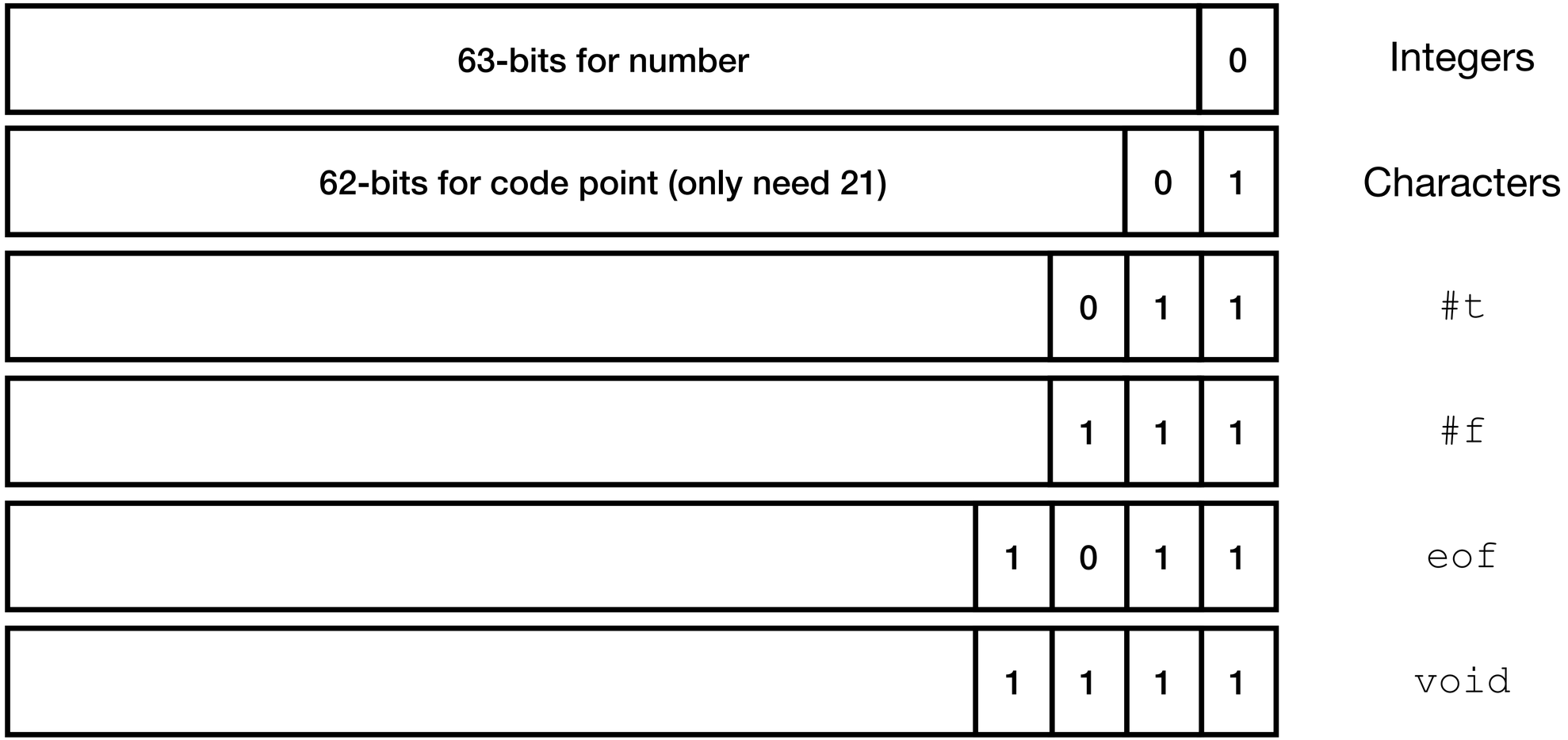
Semantic additions in Evildoer

```
;; type Value =
;; | Integer
;; | Boolean
;; | Character
;; | Eof
;; | Void
```

```
Welcome to DrRacket, version 8.6 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (read-byte)
a
97
> (read-byte)
10
> (read-byte)
#<eof>
> (void)
> (cons (void) (void))
'(#<void> . #<void>)
> (begin 1 2)
2
> (write-byte 97)
a
> (begin (write-byte 97)
         (write-byte 98))
ab
>
```

Encoding values so far in Evildoer

Type tag in least significant bits



How can we implement I/O operations?

Let's have the run-time system provide functionality

Runtime system provides I/O operations.

Compiler emits calls to run-time system.

This means:

- calling Racket functions in the host run-time system
- Calling C functions in the standalone run-time system

But how?

ABI

Application Binary Interface

An ABI standardizes:

- How function arguments are passed
- How return values are returned
- Which registers a function must preserve
- Stack layout and alignment

System V ABI

The UNIX ABI

System V ABI standardizes:

- How function arguments are passed: in rdi, rsi, rdx, rcx, r8, r9, stack
- How return values are returned: in rax, rdx
- Which registers a function must preserve: rsp, rbx, rbp, r12— r15
- Stack layout and alignment: aligned to 16-bytes

Used by all major flavors of UNIX: Linux, Mac OS, BSD, etc.

System V ABI

The UNIX ABI

The System V ABI is what allows us to write asm-interp!

The System V ABI is what allows us to write the stand-alone run-time

The System V ABI is what allows us to call…

…Racket and C from compiled code.

System V ABI

The UNIX ABI

The ABI is what allows C to call assembly code as though it were a C function.

This is how our standalone runtime can call the compiled code.

```c
#include <stdio.h>
#include <inttypes.h>

int64_t entry();

int main(int argc, char** argv)
{
  int64_t result = entry();
  printf("%" PRId64, result);
  putchar('\n');
  return 0;
}
```

Let's make an example

Calling a C function from assembly

```
(define (c-abs x)
  (asm-interp
   (prog (Global 'entry)
         (Label 'entry)
         (Sub rsp 8)
         (Mov rdi x)
         (Extern 'abs) ; defined in libc
         (Call 'abs)
         (Add rsp 8)
         (Ret))))
```

Let's make an example

Calling a Racket function from assembly

```
(require ffi/unsafe)
(current-externs ; set extern functions avaiable to asm-interp
 (list
  ;; each extern has
  ;; - a label name
  ;; - a Racket function
  ;; - a C type signature
  (extern 'gcd gcd (_fun _int64 _int64 -> _int64))))

(asm-interp
 (prog (Global 'entry)
       (Label 'entry)
       (Sub rsp 8)
       (Mov rdi 945)
       (Mov rsi 18)
       (Extern 'gcd)
       (Call 'gcd)
       (Add rsp 8)
       (Ret)))
```

Evildoer

I/O support in hosted run-time system

executor/host.rkt

Our own asm-interp that supports these primitives:

```
(define (asm-interp/host asm)
  (parameterize
      ([current-externs
        (list (extern 'read_byte prim-read-byte (_fun -> _int64))
              (extern 'peek_byte prim-peek-byte (_fun -> _int64))
              (extern 'write_byte prim-write-byte (_fun _int64 -> _int64)))])
    (asm-interp asm)))
```

I/O support in hosted run-time system

executor/host.rkt

Definition of the primitives:

```
(define (prim-read-byte)
  (value->bits (read-byte)))
(define (prim-peek-byte)
  (value->bits (peek-byte)))
(define (prim-write-byte bs)
  (value->bits (write-byte (bits->value bs))))
```

I/O support in hosted run-time system

executor/host.rkt

An example:

```
> (with-output-to-string
    (λ ()
      (asm-interp/host
       (prog (Global 'entry)
             (Label 'entry)
             (Sub rsp 8)
             (Mov rdi (value->bits 97))
             (Extern 'write_byte)
             (Call 'write_byte)
             (Add rsp 8)
             (Ret)))))
"a"
```

I/O support in standalone run-time system

runtime/io.c

```c
#include "types.h"
#include "values.h"
#include "runtime.h"

val_t read_byte(void)
{
  char c = getc(in);
  return (c == EOF) ? val_wrap_eof() : val_wrap_byte(c);
}

val_t peek_byte(void)
{
  char c = getc(in);
  ungetc(c, in);
  return (c == EOF) ? val_wrap_eof() : val_wrap_byte(c);

}

val_t write_byte(val_t c)
{
  putc((char) val_unwrap_int(c), out);
  return val_wrap_void();
}
```

Compiling I/O primitive operations

compiler/compile-ops.rkt

```
;; Op0 -> Asm
(define (compile-op0 p)
  (match p
    ;; ...
    ['read-byte (seq (Extern 'read_byte) (Call 'read_byte))]
    ['peek-byte (seq (Extern 'peek_byte) (Call 'peek_byte))]))

;; Op1 -> Asm
(define (compile-op1 p)
  (match p
    ;; ...
    ['write-byte
     (seq (Extern 'write_byte)
          (Mov rdi rax)
          (Call 'write_byte))]))
```

What about correctness?

Meaning comes from environment, too

The meaning of a program depends on more than just the program now… It implicitly depends on the input and output port. We can make that explicit:

```
;; String Expr -> (Cons Value String)
;; Interpret e with given string as input,
;; return value and collected output as string
(define (interp/io e input)
  (parameterize ((current-input-port (open-input-string input))
                 (current-output-port (open-output-string)))
    (cons (interp e)
          (get-output-string (current-output-port)))))
```

What about correctness?

Meaning comes from environment, too

We can define a similar function for compilation + execution:

```
;; Expr -> Value
(define (exec e)
  (bits->value (asm-interp/host (compile e))))

;; Asm String -> (cons Value String)
(define (exec/io asm in)
  (parameterize ((current-output-port (open-output-string))
                 (current-input-port  (open-input-string in)))
    (cons (exec asm)
          (get-output-string (current-output-port)))))
```

What about correctness?

Meaning comes from environment, too

Correct means producing the same value and output given same input:

```
;; Expr String -> Void
(define (check-compiler e i)
  (let ((r (with-handlers ([exn:fail? identity])
             (interp/io e i))))
    (unless (exn? r)
      (check-equal? r (exec/io e i)))))
```

Extort

Learning objectives

Extort

You should be able to answer:

- Why is undefined behavior convenient for compiler writers?
- Why is undefined behavior dangerous?
- How can we semantically specify error behaviors?
- What run-time techniques will be required to ensure safety?

Extort: Specifying Errors

Undefined behavior considered harmful

Let's revise `interp` to specify results for programs that error.

- Syntax doesn't change
- Semantic result extended to include errors (distinct from values)
- If `interp` is a *total function*, then no undefined behavior

```
;; type Error = ...
;; type Answer = Value | Error

;; Expr -> Answer
(define (interp e) ...)
```

Extort: Specifying Errors

Undefined behavior considered harmful

We'll use `'err` to represent errors and use exceptions to signal them.

```
;; Expr -> Answer
(define (interp e)
  (with-handlers ([(λ (x) (eq? x 'err)) identity])
    (interp-e e)))
```

`interp-e` is the main helper function that does recursive evaluation

Extort: Specifying Errors

Undefined behavior considered harmful

We'll use `'err` to represent errors and use exceptions to signal them.

```
;; Expr -> Value { raises 'err }
(define (interp-e e)
  (match e
    [(Lit d) d]
    [(Eof)   eof]
    [(Prim0 p)
     (interp-prim0 p)]
    [(Prim1 p e)
     (interp-prim1 p (interp-e e))]
    [(If e1 e2 e3)
     (if (interp-e e1)
         (interp-e e2)
         (interp-e e3))]
    [(Begin e1 e2)
     (begin (interp-e e1)
            (interp-e e2))]))
```

Extort: Specifying Errors

Undefined behavior considered harmful

Errors arise from primitives that are applied to arguments not in their domain:

```
;; Op1 Value -> Value { raises 'err }
(define (interp-prim1 op v)
  (match (list op v)
    [(list 'add1 (? integer?))            (add1 v)]
    [(list 'sub1 (? integer?))            (sub1 v)]
    [(list 'zero? (? integer?))           (zero? v)]
    [(list 'char? v)                      (char? v)]
    [(list 'integer->char (? codepoint?)) (integer->char v)]
    [(list 'char->integer (? char?))      (char->integer v)]
    [(list 'write-byte    (? byte?))      (write-byte v)]
    [(list 'eof-object? v)                (eof-object? v)]
    [_ (raise 'err)]))
```

Extort: Compiling with Errors

Undefined behavior considered harmful

Specifying errors ties the compiler's hands:

```
;; Expr String -> Void
(define (check-compiler e i)
  (check-equal? (interp/io e i)
                (exec/io e i)))
```

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
```

For next time

- Study Extort notes
- Read Fraud notes
- Take ELMS quiz