CMSC 430: Introduction to Compilers

Extort: Error Handling

Extort: when errors exist

- We ignored type mismatches.
- `(add1 (add1 #t)`, our compiler returns `#f`
- In extort, we will redesign the semantics to specify the error behavior

Extort: Specifying Errors in the Interpreter

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

```
;; type Answer = Value | 'err
;; Expr -> Answer
(define (interp e)
        ...
```

Extort: Specifying Errors in the Interpreter

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

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

Extort: Specifying Errors in the Interpreter

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

```
(define (interp-prim1 op v)
(match (list op v)
[(list 'add1 (? integer?)) (add1 v)]
[(list 'sub1 (? integer?)) (sub1 v)]
...
[(list 'integer->char (? codepoint?))
                        (integer->char v)]

[_ (raise 'err)]))
```

Extort: Compiling with Errors

- Just as in interpreter, have to check primitive arguments:

```
(define (compile-op1 p)
  (match p
  ['add1
    (seq
      (Push rax)
      (And rax 1)
      (Cmp rax 0)
      (Pop rax)
      (Jne 'err)
      (Add rax (value->bits 1)))]
```

Assert that rax holds an integer; jump to error if not.

Extort: Compiling with Errors

- Don't need update encodings of values (errors are not values)
- Do need to add a function to be called when an error happens:

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

```
(define (compile e)
(prog (Global 'entry)
(Label 'entry)
 ...
(Ret)
;; Error handler
(Label 'err)
(Extern 'raise_error)
(Call 'raise_error)))
```