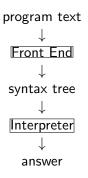
An Interpreter for a Simple Calculator (ARITH) CS496

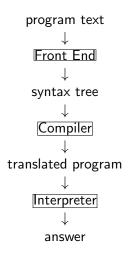
Expressions and Interpreters

- 1. Compiler vs Interpreter
- 2. A simple calculator: ARITH
- 3. Specification and Evaluation

Compiler vs Interpreter

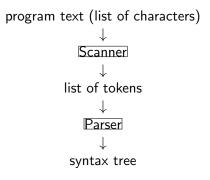


Compiler vs Interpreter



Compiler vs Interpreter

Both cases need a front end. The front end has two phases:



Syntax of ARITH

Semantics of ARITH: An Interpreter

Revising the Interpreter

ARITH: A Simple Language – The Concrete Syntax

```
\langle Expression \rangle ::= \langle Number \rangle

\langle Expression \rangle ::= \langle Expression \rangle - \langle Expression \rangle

\langle Expression \rangle ::= \langle Expression \rangle / \langle Expression \rangle

\langle Expression \rangle ::= (\langle Expression \rangle)
```

Examples of programs in concrete syntax:

> 7

Examples of programs in concrete syntax:

- **>** 7
- **>** 55 (9 7)

Examples of programs in concrete syntax:

- **>** 7
- **>** 55 (9 7)
- **>** (55 (9 11))

Examples of programs in concrete syntax:

- **>** 7
- **>** 55 (9 7)
- **>** (55 (9 11))
- **>** (55 (9/0))

Examples of programs in concrete syntax:

- **>** 7
- **>** 55 (9 7)
- **>** (55 (9 11))
- **>** (55 (9/0))

Non-examples

(55 - (7 - - 11))

Examples of programs in concrete syntax:

- **>** 7
- **>** 55 (9 7)
- **>** (55 (9 11))
- **>** (55 (9/0))

- **>** (55 (7 - 11))
- **4**-

Examples of programs in concrete syntax:

- **>** 7
- **>** 55 (9 7)
- **>** (55 (9 11))
- **>** (55 (9/0))

- **>** (55 (7 - 11))
- **4**-
- **▶** 1 // 2

ARITH: Abstract Syntax

Examples in Abstract Syntax

Let't revisit our earlier examples to translate them into the corresponding abstract syntax trees.

Concrete syntax:

► Abstract syntax (type prog):

```
Sub (Int 55, Sub (Int 9, Int 3))
```

Examples in Abstract Syntax

Concrete syntax:

$$(55 - (9 / 3))$$

Abstract syntax

1 Sub (Int 55, Div (Int 9, Int 3))

Syntax of ARITH

Semantics of ARITH: An Interpreter

Revising the Interpreter

Interpreter for Expressions

For each language construction we present two steps:

1. Specification of the interpreter (presented in a blue box in math-like language)

Specification here!

2. Implementation of the interpreter (presented as a standard box in OCaml)

Code here!

Evaluation judgements are expressions of the form:

- e is the expression, in abstract syntax, that is evaluated
- \triangleright *n* is the result
- For now, results are just integers $(n \in \mathbb{Z})$

Evaluation Rules

$$\frac{-1}{\operatorname{Int}(n) \Downarrow n} EInt$$

$$\frac{e1 \Downarrow m \quad e2 \Downarrow n \quad p = m - n}{\operatorname{Sub}(e1, e2) \Downarrow p} ESub$$

$$\frac{e1 \Downarrow m \quad e2 \Downarrow n \quad p = m/n}{\operatorname{Div}(e1, e2) \Downarrow p} EDiv$$

Evaluation rules determine an inductive set: the set of provable evaluation judgements

```
_____EInt
Int(n)↓n
```

```
let rec eval_expr : expr -> int = fun e ->
match e with
l Int n -> n
```

▶ Note the type of the interpreter:

```
eval_expr : expr -> int
```

```
\frac{\texttt{e}1 \!\!\!\! \downarrow \!\!\!\! m \quad \texttt{e}2 \!\!\!\! \downarrow \!\!\! n \quad p = m-n}{\texttt{Sub}(\texttt{e}1,\texttt{e}2) \!\!\!\! \downarrow \!\!\!\! p} \textit{ESub}
```

```
\frac{\texttt{e1} \Downarrow m \quad \texttt{e2} \Downarrow n \quad p = m/n}{\texttt{Div}(\texttt{e1},\texttt{e2}) \Downarrow p} \, \textit{EDiv}
```

```
let rec eval_expr : expr -> int = fun e ->
    match e with

Div(e1,e2) ->
eval_expr e1 / eval_expr e2
```

Problem: What if denominator is 0?

Judgements Revisited

Evaluation judgements are expressions of the form:

e∜*r*

- e is the expression, in abstract syntax, that is evaluated
- r is a result
- ▶ The set of results is $\mathbb{Z} \cup \{error\}$
- Thus a result is:
 - ► Either an integer
 - Or a special error value error

Evaluation Rules Revisited

$$\frac{-1}{\operatorname{Int}(n) \Downarrow n} EInt$$

$$\frac{e1 \Downarrow m \quad e2 \Downarrow n \quad p = m - n}{\operatorname{Sub}(e1, e2) \Downarrow p} ESub \qquad \frac{e1 \Downarrow m \quad e2 \Downarrow n \quad p = m/n}{\operatorname{Div}(e1, e2) \Downarrow p} EDiv$$

$$\frac{e1 \Downarrow error}{\operatorname{Sub}(e1, e2) \Downarrow error} ESubErr1 \qquad \frac{e1 \Downarrow m \quad e2 \Downarrow error}{\operatorname{Sub}(e1, e2) \Downarrow error} ESubErr2$$

$$\frac{e1 \Downarrow error}{\operatorname{Div}(e1, e2) \Downarrow error} EDivErr1 \qquad \frac{e1 \Downarrow m \quad e2 \Downarrow error}{\operatorname{Div}(e1, e2) \Downarrow error} EDivErr2$$

$$\frac{e1 \Downarrow m \quad e2 \Downarrow 0}{\operatorname{Div}(e1, e2) \Downarrow error} EDivErr3$$

First we model the result type

```
type 'a result = Ok of 'a | Error of string
```

Now consider the interpreter

```
1 let rec eval_expr : expr -> int result = fun e ->
    match e with
3 | Int n -> Ok n
```

```
\frac{\text{e1} \psi m \quad \text{e2} \psi n \quad p = m - n}{\text{Sub}(\text{e1}, \text{e2}) \psi p} ESub
\frac{\text{e1} \psi \text{error}}{\text{Sub}(\text{e1}, \text{e2}) \psi \text{error}} ESubErr1 \quad \frac{\text{e1} \psi m \quad \text{e2} \psi \text{error}}{\text{Sub}(\text{e1}, \text{e2}) \psi \text{error}} ESubErr2
\frac{\text{Sub}(\text{e1}, \text{e2}) \psi \text{error}}{\text{Sub}(\text{e1}, \text{e2}) \psi \text{error}} ESubErr2
```

```
\frac{\text{e1} \Downarrow m \quad \text{e2} \Downarrow n \quad p = m/n}{\text{Div}(\text{e1},\text{e2}) \Downarrow p} EDiv
\frac{\text{e1} \Downarrow error}{\text{Div}(\text{e1},\text{e2}) \Downarrow error} EDivErr1 \quad \frac{\text{e1} \Downarrow m \quad \text{e2} \Downarrow error}{\text{Div}(\text{e1},\text{e2}) \Downarrow error} EDivErr2
\frac{\text{e1} \Downarrow m \quad \text{e2} \Downarrow 0}{\text{Div}(\text{e1},\text{e2}) \Downarrow error} EDivErr3
```

Code Summary

```
let rec eval_expr : expr -> int result = fun e ->
2
     match e with
     Int n -> Ok n
     | Sub(e1,e2) ->
4
       (match eval_expr e1 with
        Error s -> Error s
6
        Ok m -> (match eval_expr e2 with
                    | Error s -> Error s
8
                    0k n \rightarrow 0k (m-n)
     | Div(e1,e2) ->
10
       (match eval_expr e1 with
        | Error s -> Error s
12
        Ok m -> (match eval_expr e2 with
                    | Error s -> Error s
14
                    0k n \rightarrow if n==0
                               then Error "Division by 0"
16
                               else Ok (m/n))
```

Syntax of ARITH

Semantics of ARITH: An Interpreter

Revising the Interpreter

Restructing the Interpreter

- Substantial amount of code checks for errors and then propagates them
- ► Although necessary, there is no interesting computational content in error propagation
- ▶ Best to have it be handled behind the scenes, by appropriate error propagation helper functions.
 - ► Three important helper functions introduced next

A Useful Metaphor

- ➤ We dub expressions of type int result, "structured programs"
- A structured program is more than just a program
- It is a program that may
 - either produce an expressed value (integer)
 - or produce an error (due to division by zero)
- Structured programs are programs that manipulate additional structure such as error handling, state, non-determinism, etc.

Helper Functions for Structured Programs

```
let return : 'a -> 'a result = fun v -> Ok v
2
    let error : string -> 'a result = fun s ->Error s
4
    let (>>=) : 'a result -> ('a -> 'b result) -> 'b result =
        (* defined later *)
6
    let rec eval_expr : expr -> int result = fun e ->
2
      match e with
      Int n -> Ok n return n
      Sub(e1.e2) ->
       (match eval_expr e1 with
6
         | Error s -> Error s
         Ok m -> (match eval_expr e2 with
8
                  | Error s -> Error s
                   | \Omega k n \rightarrow \Omega k (m-n))
10
      Div(e1.e2) ->
       (match eval_expr e1 with
12
         | Error s -> Error s
         Ok m -> (match eval expr e2 with
14
                   | Error s -> Error s
                   | 0k n \rightarrow if n==0
16
                            then Error "Division by O" error "Division by O"
                            else Ok (m/n) return (m/n)))
```

Helper Function: return and: error

▶ return: Function that creates a trivial structured program that returns a non-error result

```
let return : 'a \rightarrow 'a result = fun v \rightarrow 0k v
```

Note: non-error results are called expressed values

error: Function that creates a trivial structured program that returns an error result

```
let error : string \rightarrow 'a result = fun s \rightarrow Error s
```

Helper Function: >>= ("bind")

Function that models composition of

- 1. a structured program; and
- 2. a function that given an expressed value, produces a structured program

```
let (>>=) : 'a result -> ('a -> 'b result) -> 'b result =

2  fun c f ->
  match c with

4  | Error s -> Error s
  | Ok v -> f v
```

Note: >>= is infix and left-associative

Summary of Helper Functions

First we model the result type

```
type 'a result = Ok of 'a | Error of string
```

Now consider the interpreter

```
\frac{\text{e1} \Downarrow m \quad \text{e2} \Downarrow n \quad p = m/n}{\text{Div}(\text{e1},\text{e2}) \Downarrow p} EDiv
\frac{\text{e1} \Downarrow error}{\text{Div}(\text{e1},\text{e2}) \Downarrow error} EDivErr1 \quad \frac{\text{e1} \Downarrow m \quad \text{e2} \Downarrow error}{\text{Div}(\text{e1},\text{e2}) \Downarrow error} EDivErr2
\frac{\text{e1} \Downarrow m \quad \text{e2} \Downarrow 0}{\text{Div}(\text{e1},\text{e2}) \Downarrow error} EDivErr3
```

```
Div(e1,e2) ->
    eval_expr e1 >>= fun n1 ->
    eval_expr e2 >>= fun n2 ->
    if n2==0
then error "Division by zero"
else return (n1/n2)
```

Summary

```
let rec eval_expr : expr -> int result = fun e ->
     match e with
2
     Int(n) -> return n
     | Sub(e1,e2) ->
       eval_expr e1 >>= fun n1 ->
       eval_expr e2 >>= fun n2 ->
6
       return (n1-n2)
     | Div(e1,e2) ->
8
       eval_expr e1 >>= fun n1 ->
       eval_expr e2 >>= fun n2 ->
10
       if n2 = 0
       then error "Division by zero"
12
       else return (n1/n2)
```

The Interpreter for ARITH

- Code available in Canvas Modules/Interpreters
- ▶ Directory arith-lang
- ▶ Compile from root dir with dune utop or dune utop src
- ► Type, for eg., interp "55 (9 / 3)";; in utop