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

$$\begin{array}{c}\text{program text}\\ \downarrow \\ \boxed{\text{Front End}} \\ \downarrow \\ \text{syntax tree} \\ \downarrow \\ \boxed{\text{Interpreter}} \\ \downarrow \\ \text{answer}\end{array}$$

Compiler vs Interpreter

program text

$\downarrow$

$\boxed{\text{Front End}}$

$\downarrow$

syntax tree

$\downarrow$

$\boxed{\text{Compiler}}$

$\downarrow$

translated program

$\downarrow$

$\boxed{\text{Interpreter}}$

$\downarrow$

answer

Compiler vs Interpreter

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

$$\begin{array}{c}
\text{program text (list of characters)} \\
\downarrow \\
\boxed{\text{Scanner}} \\
\downarrow \\
\text{list of tokens} \\
\downarrow \\
\boxed{\text{Parser}} \\
\downarrow \\
\text{syntax tree}
\end{array}$$

Syntax of ARITH

Semantics of ARITH: An Interpreter

Revising the Interpreter

ARITH: A Simple Language – The Concrete Syntax

$$\begin{aligned}
\langle \textit{Expression} \rangle \quad &::= \quad \langle \textit{Number} \rangle \\
\langle \textit{Expression} \rangle \quad &::= \quad \langle \textit{Expression} \rangle \text{ - } \langle \textit{Expression} \rangle \\
\langle \textit{Expression} \rangle \quad &::= \quad \langle \textit{Expression} \rangle \ / \ \langle \textit{Expression} \rangle \\
\langle \textit{Expression} \rangle \quad &::= \quad (\langle \textit{Expression} \rangle)
\end{aligned}$$

Examples of Programs in ARITH

Examples of programs in concrete syntax:

- 7

Non-examples

Examples of Programs in ARITH

Examples of programs in concrete syntax:

- 7
- 55 - (9 - 7)

Non-examples

Examples of Programs in ARITH

Examples of programs in concrete syntax:

- 7
- 55 - (9 - 7)
- (55 - (9 - 11))

Non-examples

Examples of Programs in ARITH

Examples of programs in concrete syntax:

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

Non-examples

Examples of Programs in ARITH

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 ARITH

Examples of programs in concrete syntax:

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

Non-examples

- (55 - (7 - - 11))
- 4-

Examples of Programs in ARITH

Examples of programs in concrete syntax:

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

Non-examples

- (55 - (7 - - 11))
- 4-
- 1 // 2

ARITH: Abstract Syntax

```
type expr =
  | Int of int
  | Sub of expr*expr
  | Div of expr*expr
```

Examples in Abstract Syntax

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

- Concrete syntax:

  55 -(9-3)

- Abstract syntax (type `prog`):

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

Examples in Abstract Syntax

- Concrete syntax:

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

- Abstract syntax

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

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!
```

Specifying the Behavior of the Interpreter for Expressions

Evaluation judgements are expressions of the form:

$$\mathsf{e} \Downarrow n$$

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

Evaluation Rules

$$\frac{}{\texttt{Int(n)}\Downarrow n}\ \mathit{EInt}$$

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

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

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

Specifying the Behavior of the Interpreter for Expressions

$$\frac{}{\texttt{Int(n)} \Downarrow n}\ \textit{EInt}$$

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

- Note the type of the interpreter:

`eval_expr : expr -> int`

Specifying the Behavior of the Interpreter for Expressions

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

```
let rec eval_expr :  expr -> int = fun e ->
    match e with
     ...
    | Sub(e1,e2)    ->
      eval_expr e1 - eval_expr e2
```

Specifying the Behavior of the Interpreter for Expressions

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

```
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:

$$\mathsf{e} \Downarrow r$$

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

Evaluation Rules Revisited

$$\frac{}{\texttt{Int(n)} \Downarrow n}\ \mathit{EInt}$$

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

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

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

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

Specifying the Behavior of the Interpreter for Expressions

First we model the result type

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

Now consider the interpreter

$$\frac{}{\texttt{Int(n)} \Downarrow n}\ \mathit{EInt}$$

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

Specifying the Behavior of the Interpreter for Expressions

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

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

```
let rec eval_expr : expr -> int result = fun e ->
  match e with
  ...
  | Sub(e1,e2) ->
    (match eval_expr e1 with
     | Error s -> Error s
     | Ok m -> (match eval_expr e2 with
                | Error s -> Error s
                | Ok n -> Ok (m-n)))
```

Specifying the Behavior of the Interpreter for Expressions

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

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

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

```
  | Div(e1,e2) ->
   (match eval_expr e1 with
    | Error s -> Error s
    | Ok m -> (match eval_expr e2 with
                | Error s -> Error s
                | Ok n -> if n==0
                          then Error "Division by 0"
                          else Ok (m/n)))
```

Code Summary

```
let rec eval_expr : expr -> int result = fun e ->
  match e with
  | Int n -> Ok n
  | Sub(e1,e2) ->
    (match eval_expr e1 with
     | Error s -> Error s
     | Ok m -> (match eval_expr e2 with
                | Error s -> Error s
                | Ok n -> Ok (m-n)))
  | Div(e1,e2) ->
    (match eval_expr e1 with
     | Error s -> Error s
     | Ok m -> (match eval_expr e2 with
                | Error s -> Error s
                | Ok n -> if n==0
                          then Error "Division by 0"
                          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

let error : string -> 'a result = fun s ->Error s

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

```
let rec eval_expr : expr -> int result = fun e ->
  match e with
  | Int n -> ~~Ok n~~ return n
  | Sub(e1,e2) ->
    (match eval_expr e1 with
     | Error s -> Error s
     | Ok m -> (match eval_expr e2 with
                | Error s -> Error s
                | Ok n -> Ok (m-n)))
  | Div(e1,e2) ->
    (match eval_expr e1 with
     | Error s -> Error s
     | Ok m -> (match eval_expr e2 with
                | Error s -> Error s
                | Ok n -> if n==0
                          then ~~Error "Division by 0"~~ error "Division by 0"
                          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 -> 'a result = fun v -> Ok 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 -> 'a result = fun s -> 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 =
  fun c f ->
  match c with
  | Error s -> Error s
  | Ok v -> f v
```

Note: `>>=` is infix and left-associative

Summary of Helper Functions

```
let return : 'a -> 'a result = fun v ->
  Ok v

let error : string -> 'a result = fun s ->
  Error s

let (>>=) : 'a result -> ('a -> 'b result) -> 'b result =
  fun c f ->
  match c with
  | Error s -> Error s
  | Ok v -> f v
```

Specifying the Behavior of the Interpreter for Expressions

First we model the result type

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

Now consider the interpreter

$$\frac{}{\texttt{Int(n)} \Downarrow n}\ \mathit{EInt}$$

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

Specifying the Behavior of the Interpreter for Expressions

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

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

```
let rec eval_expr : expr -> int result = fun e ->
  match e with
  ...
  | Sub(e1,e2) ->
    eval_expr e1 >>= fun n1 ->
    eval_expr e2 >>= fun n2 ->
    return (n1-n2)
```

Specifying the Behavior of the Interpreter for Expressions

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

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

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

```
  | 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
  | Int(n) -> return n
  | Sub(e1,e2) ->
    eval_expr e1 >>= fun n1 ->
    eval_expr e2 >>= fun n2 ->
    return (n1-n2)
  | 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)
```

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`