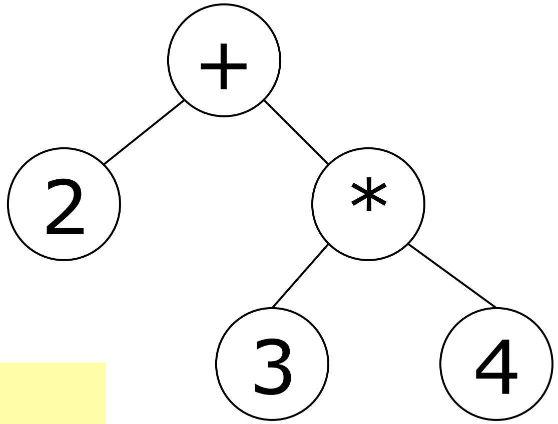
Programs as data 1+2
Overview, abstract syntax, interpretation and compilation

Peter Sestoft

Monday 2013-08-26*

Representing abstract syntax in F#

- Think of an expression "2+3*4" as a tree
- We can represent trees using datatypes:




```
type expr =
   | CstI of int
   | Prim of string * expr * expr
```

```
Prim("+", CstI 2, Prim("*", CstI 3, CstI 4))
```

```
CstI 17
Prim("-", CstI 3, CstI 4)
Prim("+", Prim("*", CstI 7, CstI 9), CstI 10)
```

What expressions?

How represent 6*0? (2+3)*4? 5+6+7? 8-9-10?

Evaluating expressions in F#

- Evaluation is a function from **`expr`** to **`int`**
- To evaluate a constant, return it
- To evaluate an operation (+,-,*)
  - evaluate its operands to get their values
  - use these values to find value of operation

RECURSION

```
let rec eval (e : expr) : int =
    match e with
      | CstI i -> i
      | Prim("+", e1, e2) -> eval e1 + eval e2
      | Prim("*", e1, e2) -> eval e1 * eval e2
      | Prim("-", e1, e2) -> eval e1 - eval e2
      | Prim _            -> failwith "unknown primitive";;
```

```
eval (Prim("-", CstI 3, CstI 4));;
```

Let’s change the meaning of minus

- Type **`expr`** is the *syntax* of expressions
- Function **`eval`** is the *semantics* of expressions
- We can change both as we like
- Let’s define subtraction so it never gives a negative result:

```
let rec eval (e : expr) : int =
    match e with
      | CstI i -> i
      | Prim("+", e1, e2) -> eval e1 + eval e2
      | Prim("*", e1, e2) -> eval e1 * eval e2
      | Prim("-", e1, e2) ->
        let res = eval e1 - eval e2
        if res < 0 then 0 else res
      | Prim _            -> failwith "unknown primitive";;
```

Expressions with variables

- Extend the **`expr`** type with a variable case:

```
type expr =
   | CstI of int
   | Var of string
   | Prim of string * expr * expr;;
```

```
CstI 17
Prim("+", CstI 3, Var "a")
Prim("+", Prim("*", Var "b", CstI 9), Var "a")
```

- We need to extend the **`eval`** function also

```
let rec eval e : int =
    match e with
      | CstI i              -> i
      | Var x               -> ???
      | Prim("+", e1, e2)  -> ...
```

How can we know the variable's value?

Use an environment

- An environment maps a name to its value
  - It is a simple dictionary or map
- Here use a list of pairs of name and value:

```
let env = [("a", 3); ("c", 78); ("baf", 666); ("b", 111)]
```

- How to look up a name in the environment:

```
let rec lookup env x =
    match env with
     | []          -> failwith (x + " not found")
     | (y, v)::r -> if x=y then v else lookup r x;;
```

Evaluation in an environment

- The environment is an extra argument
- Must pass the environment in recursive calls

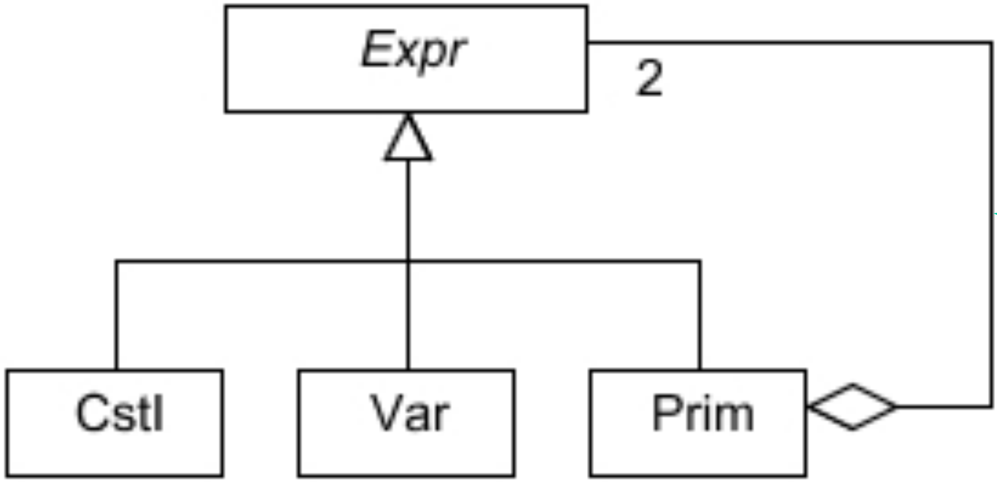
```
let rec eval e (env : (string * int) list) : int =
    match e with
      | CstI i            -> i
      | Var x             -> lookup env x
      | Prim("+", e1, e2) -> eval e1 env + eval e2 env
      | Prim("*", e1, e2) -> eval e1 env * eval e2 env
      | Prim("-", e1, e2) -> eval e1 env - eval e2 env
      | Prim _            -> failwith "unknown primitive";;
```

Representing abstract syntax in Java

```
type expr =
  | CstI of int
  | Var of string
  | Prim of string * expr * expr;;
```

Functional style

- Instead of a datatype, use an abstract class, inheritance, and composites:



Object-oriented style

The expression class declarations

```
abstract class Expr { }
class CstI extends Expr {
  protected final int i;
  public CstI(int i) {
    this.i = i;
  }
}
class Var extends Expr {
  protected final String name;
  public Var(String name) {
    this.name = name;
  }
}
class Prim extends Expr {
  protected final String oper;
  protected final Expr e1, e2;
  public Prim(String oper, Expr e1, Expr e2) {
    this.oper = oper; this.e1 = e1; this.e2 = e2;
  }
}
```

Only fields and constructors so far

Some expressions

```
Expr e1 = new CstI(17);
Expr e2 = new Prim("+", new CstI(3), new Var("a"));
Expr e3 =
   new Prim("+", new Prim("*", new Var("b"), new CstI(9)),
                 new Var("a"));
```

Evaluating expressions

```
abstract class Expr {
  abstract public int eval(Map<String,Integer> env);
}
class CstI extends Expr {
  protected final int i;
  public int eval(Map<String,Integer> env) {
    return i;
  }
}
class Var extends Expr {
  protected final String name;
  public int eval(Map<String,Integer> env) {
    return env.get(name);
  }
}
class Prim extends Expr {
  protected final String oper;
  protected final Expr e1, e2;
  public int eval(Map<String,Integer> env) {
    if (oper.equals("+"))
      return e1.eval(env) + e2.eval(env);
    else if ...
  }
}
```

Abstract eval method

Environment as map from String to int

Subclasses override eval

Back to expressions: let-bindings

let z = 17 in z + z

body

rhs = right-hand side

```
type expr =
   | CstI of int
   | Var of string
   | Let of string * expr * expr
   | Prim of string * expr * expr;;
```

```
Let("z", CstI 17, Prim("+", Var "z", Var "z"))
```

- How represent these?

  let z=x in z+x

  let z=3 in let y=z+1 in z+y

  let z=(let x=4 in x+5) in z*2

Evaluation of expressions with let

```
let rec eval e (env : (string * int) list) : int =
    match e with
    | CstI i            -> i
    | Var x             -> lookup env x
    | Let(x, erhs, ebody) ->
      let xval = eval erhs env
      let env1 = (x, xval) :: env
      in eval ebody env1
    | Prim("+", e1, e2) -> eval e1 env + eval e2 env
    | Prim("*", e1, e2) -> eval e1 env * eval e2 env
    | Prim("-", e1, e2) -> eval e1 env - eval e2 env
    | Prim _            -> failwith "unknown primitive";;
```

When let-bindings are nested, environment is a stack!

- To evaluate "**let x=erhs in ebody**":
  - Evaluate **erhs** in given environment to get **xval**
  - Extend **env** with binding **(x, xval)** binding to get **env1**
  - Evaluate **ebody** in **env1**

Set operations in F#

- We represent a set as a list without duplicates; simple but inefficient for large sets
- The empty set $\varnothing$ is represented by []
- Set membership: $x \in vs$

```
let rec mem x vs =
    match vs with
    | []    -> false
    | v::vr -> x=v || mem x vr;;
```

```
> mem 42 [2; 5; 3];;
val it : bool = false
> mem 42 [];;
val it : bool = false
> mem 42 [2; 67; 42; 5];;
val it : bool = true
```

Set union and difference in F#

- Set union: A U B

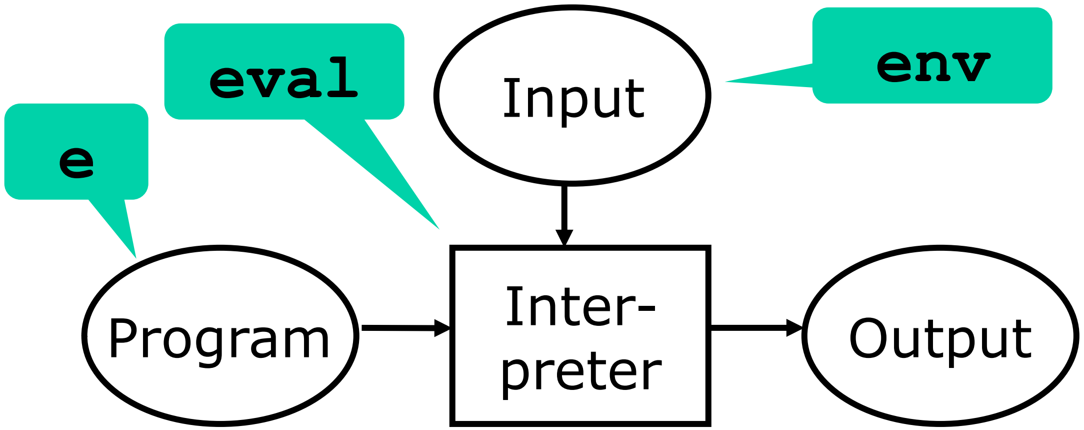
```
let rec union (xs, ys) =
    match xs with
    | []    -> ys
    | x::xr -> if mem x ys then union(xr, ys)
               else x :: union(xr, ys);;
```

- Set difference: A \ B

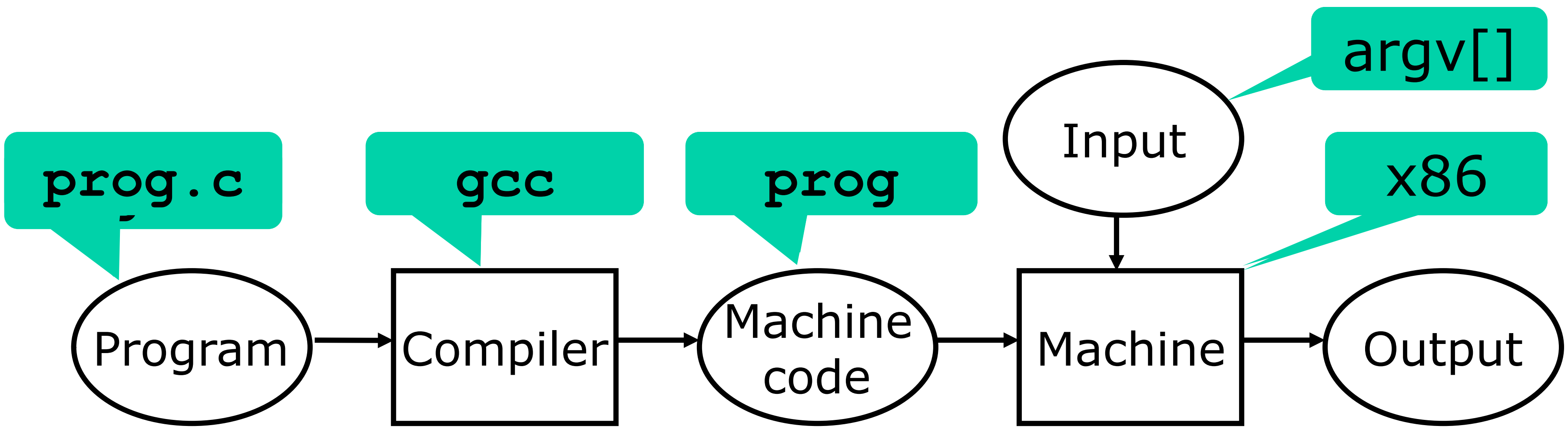
```
let rec minus (xs, ys) =
    match xs with
    | []    -> []
    | x::xr -> if mem x ys then minus(xr, ys)
               else x :: minus(xr, ys);;
```

Interpretation versus compilation

- Interpretation = one-stage execution/evaluation:



- Compilation = two-stage execution/evaluation:



Why compilation?

- Better correctness and safety. The compiler can:
  - check that all names are defined: classes, methods, fields, variables, types, functions, …
  - check that the names have the correct type
  - check that it is legal to refer to them (not private etc)
  - improve the code, e.g. inline calls to private methods
- Better performance
  - The compiler checks are performed *once*, but the machine code gets executed again and again
- Why *not* compilation?
  - Compilation reduces flexibility by imposing static type checks and static name binding
  - Web programming often requires more flexibility
  - … hence PHP, Python, Ruby, JavaScript, VB.NET, …

Replacing variable names with indices

- Goal: At runtime, there should be no variable names, only indices (locations)
- Instead of symbolic names:

```
Let("z", CstI 17, Prim("+", Var "z", Var "z"))
```

we shall use variable indexes:

```
Let(CstI 17, Prim("+", Var 0, Var 0))
```

No variable name

0 means closest variable binding

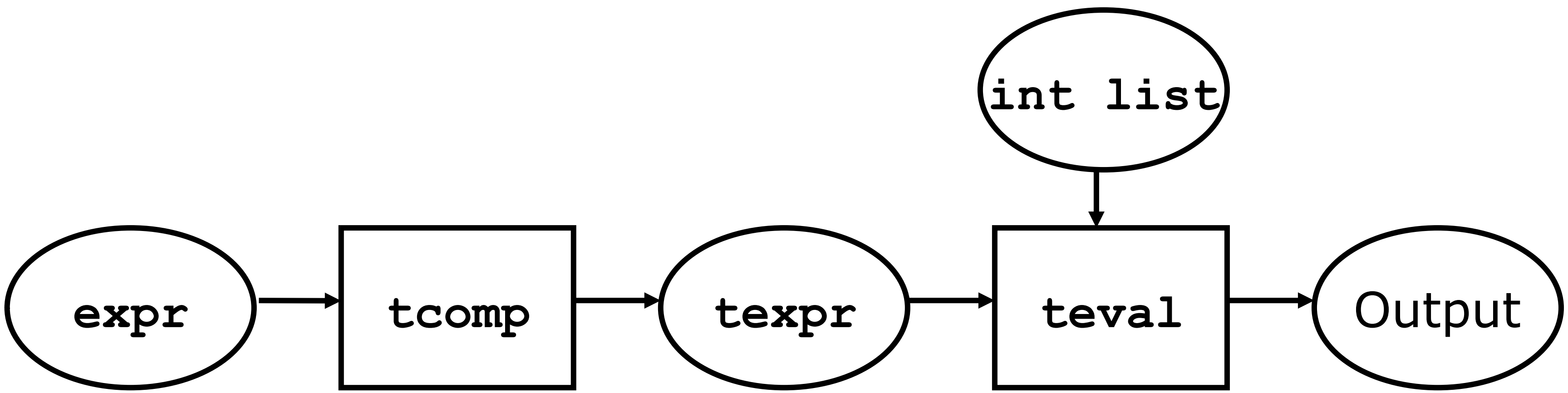
- Index = number of let-bindings to cross:

```
Let("z", CstI 17, Let("y", CstI 25,
                      Prim("+", Var "z", Var "y")))
```

Indexes instead of variable names

- We shall compile to this “target” language:

```
type texpr =                            (* target expressions *)
   | TCstI of int
   | TVar of int                        (* index at runtime   *)
   | TLet of texpr * texpr
   | TPrim of string * texpr * texpr
```



Evaluating texprs

- The runtime environment of a texpr is a list of values – not (name, value) pairs

```
let rec teval (e : texpr) (renv : int list) : int =
    match e with
    | TCstI i -> i
    | TVar n  -> List.nth renv n
    | TLet(erhs, ebody) ->
      let xval = teval erhs renv
      let renv1 = xval :: renv
      teval ebody renv1
    | TPrim("+", e1, e2) -> teval e1 renv + teval e2 renv
    | TPrim("*", e1, e2) -> teval e1 renv * teval e2 renv
    | TPrim("-", e1, e2) -> teval e1 renv - teval e2 renv
    | TPrim _            -> failwith "unknown primitive"
```

Replacing variable names with indices

```
let rec getindex vs x =
    match vs with
      | []     -> failwith "Variable not found"
      | y::yr -> if x=y then 0 else 1 + getindex yr x;;

let rec tcomp (e : expr) (cenv : string list) : texpr =
   match e with
   | CstI i -> TCstI i
   | Var x  -> TVar (getindex cenv x)
   | Let(x, erhs, ebody) ->
     let cenv1 = x :: cenv
     in TLet(tcomp erhs cenv, tcomp ebody cenv1)
   | Prim(ope, e1, e2) -> TPrim(ope, tcomp e1 cenv, tcomp e2 cenv)
```

let z=3 in let y=z+1 in z+y

`[]`

`["z"]`

`["y"; "z"]`

- What if the expression **e** is not closed?

Binding-times in the environment

- Run-time environment in expr interpreter:
  `[("y", 4); ("z", 3)]`
- Compile-time environment in expr compiler:
  `["y"; "z"]`
- Run-time environment of texpr “machine”:
  `[4; 3]`

- The interpreter runtime environment splits to
  - A compile-time environment in the compiler
  - A runtime environment in the “machine”
- We meet such “binding-time” separation again later…

Towards more machine-like code

- Consider expression `2 * 3 + 4 * 5`
- Write it in *postfix*: `2 3 * 4 5 * +`
- Postfix is sequential code for a *stack machine*:

Instructions:	**Stack contents:**
`2 3 * 4 5 * +`	`2`
`3 * 4 5 * +`	`2 3`
`* 4 5 * +`	`6`
`4 5 * +`	`6 4`
`5 * +`	`6 4 5`
`* +`	`6 20`
`+`	`26`

7-minute exercises

- What is the postfix of

  2 * 3 + 4

  2 + 3 * 4

  2 * (3 + 4)

  2 – 3 – 4 – 5

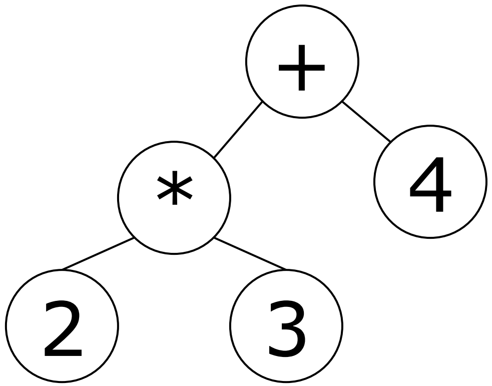
  2 – (3 – (4 – 5))

  2 + 3 * 4 / 5

- Evaluate the postfix versions using a stack

Expression stack machine without variables

Instruction	Stack before	Stack after	Effect
RCSTI n	s	s, n	Push const
RADD	s, n1, n2	s, n1+n2	Add
RSUB	s, n1, n2	s, n1-n2	Subtract
RMUL	s, n1, n2	s, n1*n2	Multiply
RDUP	s, v	s, v, v	Duplicate top elem
RSWAP	s, v1, v2	s, v2, v1	Swap



Compilation of expr to stack machine code

- A constant **`i`** compiles to code **`[RCst i]`**
- An operator application **`e1+e2`** compiles to:
  - code for operand **`e1`**
  - code for operand **`e2`**
  - code for the operator **+**

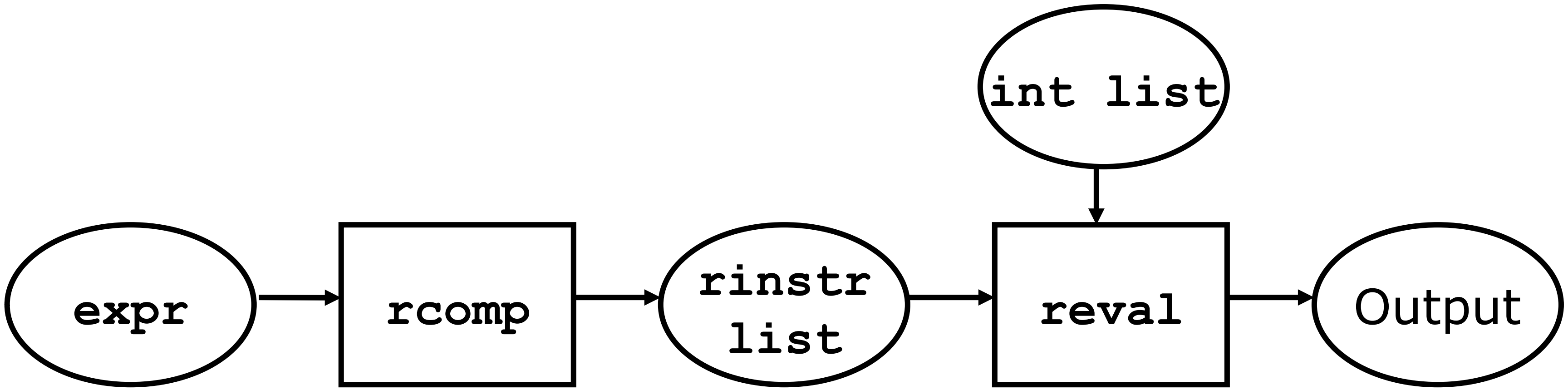
```
let rec rcomp (e : expr) : rinstr list =
    match e with
    | CstI i             -> [RCstI i]
    | Var _              -> failwith "rcomp cannot do Var"
    | Let _              -> failwith "rcomp cannot do Let"
    | Prim("+", e1, e2) -> rcomp e1 @ rcomp e2 @ [RAdd]
    | Prim("*", e1, e2) -> rcomp e1 @ rcomp e2 @ [RMul]
    | Prim("-", e1, e2) -> rcomp e1 @ rcomp e2 @ [RSub]
    | Prim _             -> failwith "unknown primitive";;
```

```
rcomp (Prim("+", Prim("*", CstI 2, CstI 3), CstI 4));;
val it : rinstr list = [RCstI 2; RCstI 3; RMul; RCstI 4; RAdd]
```

Stack machine (without variables)

- A direct implementation of state transitions:

```
let rec reval (inss : rinstr list) (stack : int list) =
    match (inss, stack) with
    | ([], v :: _) -> v
    | ([], [])      -> failwith "reval: no result on stack!"
    | (RCstI i :: insr,           stk)   -> reval insr (i::stk)
    | (RAdd    :: insr, i2::i1::stkr) -> reval insr ((i1+i2)::stkr)
    | (RSub    :: insr, i2::i1::stkr) -> reval insr ((i1-i2)::stkr)
    | (RMul    :: insr, i2::i1::stkr) -> reval insr ((i1*i2)::stkr)
    | (RDup    :: insr,     i1::stkr) -> reval insr (i1 :: i1 :: stkr)
    | (RSwap   :: insr, i2::i1::stkr) -> reval insr (i1 :: i2 :: stkr)
    | _ -> failwith "reval: too few operands on stack";;
```



Concepts

- An expression e is compiled to a sequence of instructions
- **Net effect principle**:
  - The *net effect* of executing the instructions is to leave the expression's value on the stack
- *Compiler correctness* relative to interpreter
  - Executing the compiled code gives the same result as executing the original expression
  - That is:

    `reval (rcomp e []) []` equals **`eval e []`**