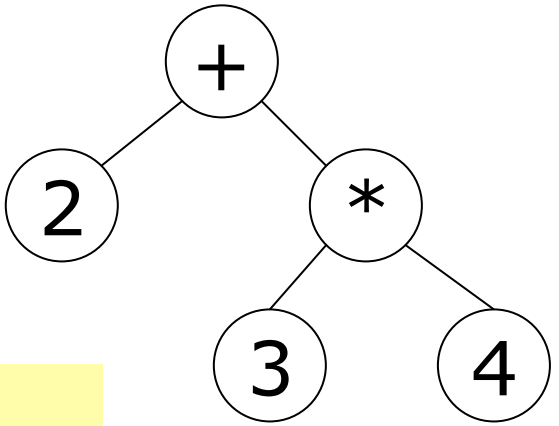
Programs as data 1
Overview, F# programming, abstract syntax

Peter Sestoft

Monday 2012-08-27**

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 operator

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 say that subtraction 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";;
```

How convert expression to a string?

- We want a function like this:

```
let rec fmt (e : expr) : string =
    ...
```

What goes here?

- For instance

  `fmt (CstI 654)` gives `"654"`

  `fmt (Prim("-", CstI 3, CstI 4))` gives `"(3-4)"`

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;;
```

- How to put x with value 42 into an env?

Evaluation in an environment

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

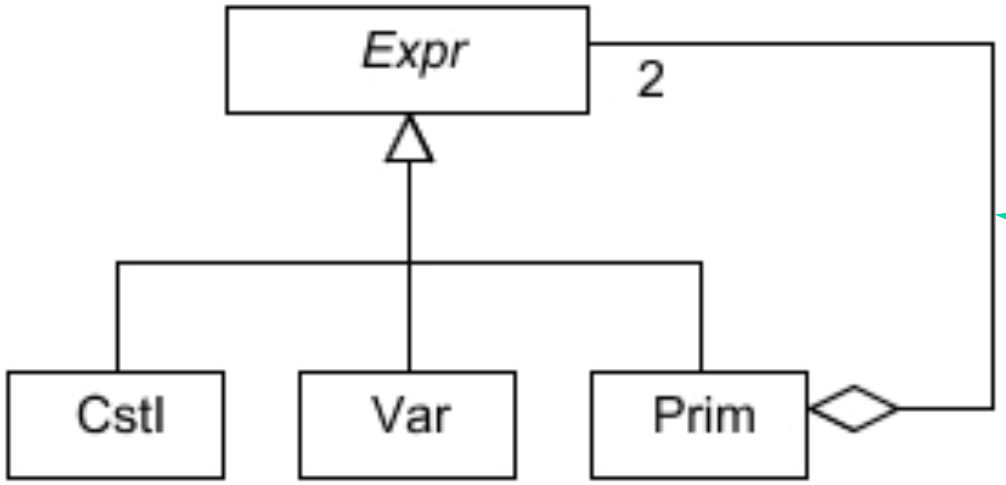
```
let rec eval e (env : (string * int) list) : int =
    match e with
      | CstI i                -> i
      | Var x                 -> lookup env x
      | Prim("+", e1, e2) -> eval e1 env + eval e2 env
      | Prim("*", e1, e2) -> eval e1 env * eval e2 env
      | Prim("-", e1, e2) -> eval e1 env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

Evaluating an expression

```
int r1 = e1.eval(env0);
```

- How format an expression as a **`String`**?

Functional vs object-oriented

	Functional	Object-oriented
Expression variant	Datatype constructor	Subclass
Choice in operation	Pattern matching in function	Virtual method in subclasses
Adding a new expression variant	Edit *several* functions (add new variant to each one)	Add *one* subclass (with all operations)
Adding a new expression operation	Add *one* function (operation on all variants)	Edit *several* classes (add new operation to each one)
Match composite expressions	Easy	Hard

The Expression Problem

Example: Expression simplification

- 0+e2 gives e2; e1+0 gives e1; 1*e2 gives e2
- Easy with pattern matching:

```
let rec simp e =
    match e with
      | Prim("+", CstI 0, e2) -> e2
      | Prim("+", e1, CstI 0) -> e1
      | Prim("*", CstI 1, e2) -> e2
      | ...                   -> ...
```

- Difficult with C++/Java/C#-style single virtual dispatch
- Newer OO languages such as Scala make this easier than Java and C#