Review: Expressions

- They are usually tree-structured
- They can be defined over many value domains
  - numbers, booleans, strings, lists, sets, etc.
- They abstract away from evaluation order and use of temporaries (contrast with, e.g., stack machine)
- They may have unevaluated subexpressions (e.g. if)
- They may not be well-defined on all dynamic values
- They may be statically ill-formed
Today: Names (and Functions)

- Part of being a “high-level” language is letting the programmer name things:
  - variables, constants, types
  - functions, classes, modules
  - fields, operators, ...

- Generically, we call names identifiers

- An identifier binding makes an association between the identifier and the thing it names

- An identifier use refers to the thing named

- The scope of a binding is the part of the program where it can be used
Scala Example

```scala
object Printer {
  def print(expr: Expr): String = unparse(expr).toString()

  def unparse(expr: Expr): SExpr = expr match {
    case Num(n) => SNum(n)
    case Add(l, r) => SList(SSym('+')::unparse(l)::unparse(r)::Nil)
    case Mul(l, r) => SList(SSym('*')::unparse(l)::unparse(r)::Nil)
    case Div(l, r) => SList(SSym('/')::unparse(l)::unparse(r)::Nil)
  }
}
```

- Identifier syntax is language-specific
- Usually unbounded sequence of alpha|numeric|symbol(?)
- Further rules/conventions for different categories
- Identifiers are distinct from keywords! Some identifiers are pre-defined
Names for values

Most languages let us bind names to values computed by expressions.

typically (maybe confusingly) called “variables”

Why are variables useful?

- In imperative languages, they are used to refer to memory cells that can be read or updated

- They let us share expressions

- to save repeated writing and, maybe, evaluation

- They are needed to parameterize functions
Local Value Bindings

\[ \text{expr ::= num} \mid \text{expr} + \text{expr} \mid \ldots \mid (\text{expr}) \mid \text{id} \mid \text{let id = expr in expr} \]

(\text{let a = 8 + 5 in a * 3}) + 3

binding

use

scope

\[
\begin{array}{c}
\text{Add} \\
\text{Let}_a \\
\text{Add} \\
\text{Mul} \\
\text{Var}_a \\
\text{Num}_3
\end{array}
\]

\[
\begin{array}{c}
\text{Num}_8 \\
\text{Num}_5 \\
\text{Num}_3
\end{array}
\]
Semantics via Substitution

(\text{let } a = 8 + 5 \text{ in } a \times 3) + 3

“Rewrite the program text”

\(( (8 + 5) \times 3) + 3 \)
Bound vs. Free

A variable use $x$ is **bound** if it appears in the scope of a binding for $x$.

Otherwise, it is **free**.

Bound and free are relative to an enclosing subexpression, e.g.

Bound in: $a$

Free in: $a \times 3$

We cannot evaluate a free variable.
(let a = 8 + 5 in a * 3) +
(let b = 1 in b + 2)

What if both let’s bind a?
(let a = 8 + 5 in
  let b = a - 10 in
  a * b) + 2
Shadowing

\[(\text{let } a = 8 + 5 \text{ in } \text{let } a = a - 10 \text{ in } 36 + a) + 3\]

Need more careful definition of substitution:
Don’t substitute for variable \(x\) inside a nested let-binding for \(x\)
And that is still not quite good enough...see homework
Substitution Reconsidered

(\text{let } a = 8 + 5 \text{ in } a * a) + 3

"Rewrite the program text"

Is this a good idea?
- It gives the expected answer
- But it doesn’t reflect desired sharing of computations

(((8 + 5) * (8 + 5)) + 3)
Eager Evaluation Semantics

\((\text{let } a = 8 + 5 \text{ in } a * a) + 3\)

“Reduce body of let before substitution”

\((\text{let } a = 13 \text{ in } a * a) + 3\)

Note that this isn’t always a win...

\(\text{let } a = \text{do_giant_computation()} \text{ in } 42\)
Environments

- Substitutions are useful for giving high-level semantics
  - And they can be used to build interpreters
  - But these don’t have a very “realistic” flavor
  - In conventional implementations, the program itself does not change during execution

- An alternative to substitution is to maintain an environment: a map from variables to values
  - Evaluating a let binding extends the env.
  - Evaluating a variable use looks up in the env.
  - Can think of this as a “deferred substitution”
object Interp {
    type Env = Map[String,Value] // immutable map
    val emptyEnv = Map[String,Value]()
    def interpE(expr:Expr, env:Env) : Value = expr match {
        case Num(n) => NumV(n)
        case Add(l,r) => (interpE(l, env), interpE(r, env)) ...
        case Let(x,d,b) => {
            val v = interpE(d, env); interpE(b, env + (x -> v))
        }
        case Var(x) => (env get x) match {
            case Some(v) => v
            case None =>
                throw InterpException("Undefined variable:" + x)
        }
    }
    // evaluate root of expression tree
    val v = interpE(expr, emptyEnv)
}
Environment-based Semantics

- Behavior of this interpreter relies on semantics of Scala’s immutable maps

- `Map[String,Value]()` creates a fresh empty map

- `env + (x -> v)` creates a new map that is just like `env`, except that `x` is bound to `v`

- `env get x` returns either `Some(v)` where `v` is the value bound to `x` or `None` if `v` is not bound

- This gives us eager evaluation and nestable local scopes with shadowing!
Procedures and Functions

- Procedures have a long history as an essential programming tool

- Low-level view: *subroutines* give a way to avoid duplicating frequently-used code

- High-level view: procedural *abstraction* lets us divide large programs into smaller pieces with hidden internals

- Procedures can be *parameterized* over values, types,…

- A *function* is just a procedure that returns a value

- Or, conversely, a procedure is just a function whose result is uninteresting
Function parameterization

Consider adding functions to our toy expression language.

To be useful in that context, a function must have one or more value parameters (Why?)

We need value identifiers to name these parameters.

1. The scope of a parameter is the function body.
2. The value of each parameter is provided at the function call (or "application") site.
Semantics via Substitution

Given a function declaration AST:

\[ (f \ x \ (+ \ x \ 3)) \]

To evaluate a function application:

\[ (@ \ f \ (* \ 13 \ 3)) \]

We substitute a copy of the body for the application and then substitute the actual for the formal in that copy:
Call-by-name

In this substitution semantics, the actual parameter is re-evaluated each time it used:

\[(\text{\texttt{f x (\texttt{+ x x})}}) \quad \rightarrow \quad \text{\texttt{@ f (* 2 3)) \quad \rightarrow \quad (+ (* 2 3) (* 2 3))}}\]

This semantics is known as **call-by-name** evaluation.

It duplicates work if a parameter is used twice.

But it saves work if a parameter is not used at all.

Even more useful is a variant called **lazy** evaluation, which evaluates each parameter **at most** once.
Call-by-value

Let’s switch back to a semantics based on value environments

Gives better match to conventional implementations

Idea: to evaluate an application:

put bindings from actual parameters to formal parameters into the environment

then evaluate the function body in that environment

But our environments map variables to values!

So we must evaluate the actual parameters to values first, before we add the new bindings to the environment

This semantics is known as call-by-value evaluation
Hardware Calls

- A function is normally compiled to a machine-code subroutine
- A single sequence of code that can be invoked from multiple places
- Hardware gives support for remembering the return address to jump to when function is done
- Parameter values are typically passed in machine registers or on the stack
- Fairly close match to environment model
- Call-by-value is most efficient choice at hardware level
Environment-based Interpreter

object Interp {
  val emptyEnv = Map[String,Value]()
  def interpE(expr:Expr, env:Env) : Value = expr match {
    case Num(n) => NumV(n)
    case Add(l,r) => (interpE(l, env), interpE(r, env)) ...
    case App(f,a) => (functions get f) match {
      case Some((param,body)) => {
        val v = interpE(a, env)
        interpE(body, initialEnv + (param -> v))
      }
      case None => throw InterpException(…)
    }
    case Var(x) => (env get x) match ...
  }
  // evaluate root of expression tree
  val v = interpE(expr, emptyEnv)
}
object Interp {
  val emptyEnv = Map[String,Value]()
  def interpE(expr:Expr, env:Env) : Value = expr match {
    case Num(n) => NumV(n)
    case Add(l,r) => (interpE(l, env), interpE(r, env)) ...
    case App(f,a) => (functions get f) match {
      case Some((param, body)) => {
        val v = interpE(a, env) ??
        interpE(body, initialEnv + (param -> v))
      }
      case None => throw InterpException(...) 
    }
    case Var(x) => (env get x) match ...
  }
  // evaluate root of expression tree
  val v = interpE(expr, emptyEnv)
}
Environment-based Interpreter

object Interp {
  val emptyEnv = Map[String,Value]()
  def interpE(expr:Expr,env:Env) : Value = expr match {
    case Num(n) => NumV(n)
    case Add(l,r) => (interpE(l,env),interpE(r,env)) …
    case App(f,a) => (functions get f) match {
      case Some((param,body)) => {
        val v = interpE(a,env)
        interpE(body,env + (param -> v))
      }
      case None => throw InterpException(…)
    }
    case Var(x) => (env get x) match …
  }
  // evaluate root of expression tree
  val v = interpE(expr,emptyEnv)
}
“Dynamic scope”

What should happen in the following program?

(\(f\ \text{x}\ (\text{+}\ \text{x}\ \text{y})\))  \quad (\text{@}\ \text{f}\ 42)

How about this one?

(\(f\ \text{x}\ (\text{+}\ \text{x}\ \text{y})\))  \quad (\text{let}\ \text{y}\ 2\ \text{(}\text{@}\ \text{f}\ 42)\text{)})

One possible answer: let the value of \(y\) “leak” into \(f\)

But this is a bad idea! Why?
Environment-based Interpreter

object Interp {
    val emptyEnv = Map[String,Value]()
    def interpE(expr:Expr,env:Env) : Value = expr match {
        case Num(n) => NumV(n)
        case Add(l,r) => (interpE(l,env),interpE(r,env)) ...
        case App(f,a) => (functions get f) match {
            case Some((param,body)) => {
                val v = interpE(a,env)
                interpE(body,emptyEnv + (param -> v))
            }
            case None => throw InterpException(...)  
        }
        case Var(x) => (env get x) match ...
    }
    // evaluate root of expression tree
    val v = interpE(expr,emptyEnv)
}
“Static scope”/“Lexical scope”

This program remains erroneous

\[(f \ x \ (+ \ x \ y))\]

\[(let \ y \ 2 \ (@ \ f \ 42))\]

Looking at a function declaration, we can always determine if and where a variable is bound without considering the dynamic execution of the program!

Some scripting languages still use dynamic scope, but as programs get larger, its dangers become obvious