

# Programming Languages

## Standard ML

Utah Tech University

Fall 2023

# Evaluation on Paper

TODO

# Representing Arithmetic

TODO

# Representing Abstract Syntax

We will create a new tree datatype to represent ASTs:

```
(define-type Exp
  [num (n : Number)]
  [plus (left : Exp) (right : Exp)])
```

This says:

- ▶ We are defining a new type, `Exp`
- ▶ There are two ways of making an `Exp`
- ▶ One way is through the constructor `num`:
  - ▶ A `num` takes one argument
  - ▶ That argument must be an actual number
- ▶ The other way is through the constructor `plus`:
  - ▶ A `plus` takes two arguments
  - ▶ Both arguments must be `Exps`

## Defining the Evaluator

What is the type of an evaluator program? For now, every expression evaluates to a number:

```
(calc : (Exp -> Number))
```

It is a function, so:

```
(define (calc e)  
  ...)
```

What does it do? There are two kinds of expressions, so we handle each independently:

```
(type-case Exp e  
  [(num n) ...]  
  [(plus l r) ...])
```

Ask what should be returned in each case:

```
(type-case Exp e  
  [(num n) n]  
  [(plus l r) (+ l r)])
```

## Defining the Evaluator

Bringing it together we get:

```
(define (calc e)
  (type-case Exp e
    [(num n) n]
    [(plus l r) (+ l r)]))
```

Oops! A type error here:  $(+ \text{ l } \text{ r})$ . Addition is defined on numbers, but  $\text{l}$  and  $\text{r}$  are not numbers, they are expressions. How to convert an expression into a number?

```
(define (calc e)
  (type-case Exp e
    [(num n) n]
    [(plus l r) (+ (calc l) (calc r))]))
```

This is an example of *structural recursion*. The structure of the AST and the structure of the program that walks it mirror each other.

# Defining the Evaluator

We can test it:

```
(calc (num 1))
```

```
(calc (plus (num 1) (num 2)))
```

```
(plus (num 1)  
      (plus (num 2) (num 3)))
```

Note that our definition of `+` is taken directly from the host language: we are not defining it explicitly. What about a language like Python where `+` can mean numeric addition or string concatenation?

## Testing the Evaluator

We can do *ad hoc* testing, but it is better to have repeatable tests:

```
(test (calc (num 1)) 1)
(test (calc (num 2.3)) 2.3)
(test (calc (plus (num 1) (num 2))) 3)
(test (calc (plus (plus (num 1) (num 2))
                  (num 3)))
      6)
(test (calc (plus (num 1)
                  (plus (num 2) (num 3))))
      6)
(test (calc (plus (num 1)
                  (plus (plus (num 2)
                              (num 3))
                        (num 4))))
      10)
```

We can also tell it to only report failed tests:

```
(print-only-errors #true)
```



## Some Subtler Tests

Try the following test:

```
(test (calc (plus (num 0.1) (num 0.2))) 0.3)
```

It succeeds! What about:

```
(test (calc (plus (num 0.1) (num 0.2))) 1/3)
```

This reinforces an earlier point: by adopting plait's primitives, we also inherit its semantics. We should be mindful of this, and be ready to implement our own explicit behavior if this is not what we want.

# The Problem

We have an AST representation for programs, but how do we get programs into that representation? We could write each expression out directly as an AST data type:

```
(num 1)
(plus (num 1) (num 2))
(plus (num 1)
      (plus (num 2) (num 3)))
```

This works, but it is tedious, and will get even worse as we start writing more complex tests. We want a program to convert a convenience surface syntax into an AST: a *parser*.

We will use the parenthetical syntax of Racket:

- ▶ It strikes a reasonable balance between convenience and simplicity
- ▶ plait already has special support for it so parsing will be easier and we can focus on other things

# The Problem

We would like to be able to type the above examples using something like:

```
1
(+ 1 2)
(+ 1 (+ 2 3))
```

Racket lets us use `()`, `[]`, and `{}` interchangeably so we will adopt a convention: we will write programs to be represented using `{}`, e.g.:

```
1
{+ 1 2}
{+ 1 {+ 2 3}}
```

This will visually distinguish the programs we are representing from the host language.

# S-Expressions

There is a name for this syntax: these are called *s-expressions*. In plait we will write these expressions preceded by a *back-tick*. A back-tick followed by a Racket term is of type `S-Exp`:

```
`1  
`2.3  
`-40
```

These are all numeric s-expressions. We can also write

```
`{+ 1 2}  
`{+ 1 {+ 2 3}}
```

# S-Expressions

These are *list* s-expressions:

```
> (s-exp-list? `1)
- Boolean
#f
> (s-exp-list? `{+ 1 2})
- Boolean
#t
> (s-exp-list? `{+ 1 {+ 2 3}})
- Boolean
#t
```

So the first is not but the second two are; similarly:

```
> (s-exp-number? `1)
- Boolean
#t
> (s-exp-number? `{+ 1 {+ 2 3}})
- Boolean
#f
```

# S-Expressions

The S-Exp type is a container around the actual number of list, which we can extract:

```
> (s-exp->number `1)
- Number
1
> (s-exp->list `{+ 1 2})
- (Listof S-Exp)
(list `+ `1 `2)
```

What happens if you apply `s-exp->number` to a list s-exp or `s-exp->list` to a number s-expression? Or either to something that is not an s-expression at all? Try it!

Note the list above contains two numbers but also a symbol for `+`. A *symbol* s-expression is kind of like a string but treated atomically: the only operation on symbols is equality testing.

```
> (s-exp-symbol? `+)
- Boolean
#t
> (s-exp->symbol `+)
- Symbol
`+
```

## Writing the parser

Think about what type we want for our parser.

What does our parser need to produce? Whatever the calculator consumes, i.e., `Expr`. What does it consume? Program source expressions written in a “convenient” syntax, i.e., `S-Exp`. Hence, its type must be

```
(parse : (S-Exp -> Expr))
```

That is, it converts the surface syntax into an AST. First we need to check what kind of s-exp we were given:

```
(define (parse s)
  (cond
    [(s-exp-number? s) ...]
    [(s-exp-list? s) ...]))
```

If it's a numeric s-exp, then we need to extract the number and pass it to the `num` constructor:

```
(num (s-exp->number s))
```

## Writing the parser

Otherwise, we need to extract the list and check whether the first thing in the list is an addition symbol. If it is not, we signal an error:

```
(let ([l (s-exp->list s)])  
  (if (symbol=? '+  
                (s-exp->symbol (first l)))  
      ...  
      (error 'parse "list not an addition"))))
```

Otherwise, we create a plus term by recurring on the two sub-pieces.

```
(plus (parse (second l))  
      (parse (third l)))
```



## Writing the parser

Putting it all together:

```
(define (parse s)
  (cond
    [(s-exp-number? s)
     (num (s-exp->number s))]
    [(s-exp-list? s)
     (let ([l (s-exp->list s)])
       (if (symbol=? '+
                     (s-exp->symbol (first l)))
           (plus (parse (second l))
                 (parse (third l)))
           (error 'parse "list not an addition")))]))
```

It is a lot and somewhat tedious, but this is about as complicated as parsing will get for us. Future parsers will extend this pattern.

# Testing the parser

We should write tests:

```
(test (parse `1) (num 1))  
(test (parse `2.3) (num 2.3))  
(test (parse `{+ 1 2}) (plus (num 1) (num 2)))  
(test (parse `{+ 1  
              {+ {+ 2 3}  
                4}})  
      (plus (num 1)  
            (plus (plus (num 2)  
                        (num 3))  
                  (num 4)))))
```

## Testing the parser

We have only written *positive* tests. We can also write *negative* tests for situations where we expect errors:

```
(test/exn (parse `{1 + 2}) "")
```

`test/exn` takes a string that must be a substring of the error message. Why the empty string and not something like `"addition"`? Let's try it out and see. How could you improve the parser to address this?

Other situations we should check for include there being too few or too many sub-parts. Addition, for instance, is defined to take exactly two sub-expressions. What if a source program contains none, one, three, four, ...? This is the kind of pedantry that parsing calls for.

## Testing the parser

Now we can compose `parse` and `calc` because `parse` produces output that `calc` can consume. We can even write a helper function that does it for us:

```
(run : (S-Exp -> Number))  
(define (run s)  
  (calc (parse s)))
```

Now we can repeat our earlier tests

```
(test (run `1) 1)  
(test (run `2.3) 2.3)  
(test (run `{+ 1 2}) 3)  
(test (run `{+ {+ 1 2} 3})  
      6)  
(test (run `{+ 1 {+ 2 3}})  
      6)  
(test (run `{+ 1 {+ {+ 2 3} 4}})  
      10)
```

Compare this against the `calc` tests we had earlier!

# Evaluating Conditionals

Our language will be more interesting with conditionals. We will start with a simple `if` with three parts: the condition, the then-branch, and the else-branch. Later we will see how to build more complex conditionals on top of this.

Our SImPI interpreter needs a few changes:

1. Extend the datatype representing expressions to include conditionals.
2. Extend the evaluator to handle (the representation of) these new expressions.

Optionally, if we have a parser we should also

3. Extend the parser to produce these new representations.

## Extending the AST

Because we have fixed our conditionals to have three parts, we just need to represent that in the AST. This is straightforward:

```
(define-type Exp
  [num (n : Number)]
  [plus (left : Exp) (right : Exp)]
  [cnd (test : Exp) (then : Exp) (else : Exp)])
```

The real work will happen in the evaluator.

## Extending the Calculator

Adding conditionals doesn't change what our calculator previously did, we can leave that intact, and just focus on the handling of `if`:

```
(define (calc e)
  (type-case Exp e
    [(num n) n]
    [(plus l r) (+ (calc l) (calc r))]
    [(cnd c t e) ...]))
```

## Extending the Calculator

We can start by recursively evaluating each term:

```
(define (calc e)
  (type-case Exp e
    [(num n) n]
    [(plus l r) (+ (calc l) (calc r))]
    [(cnd c t e) ... (calc c) ... (calc t) ... (calc e) ...]))
```

But not we have a few problems:

1. What is the result of calling `(calc c)`? We expect some kind of Boolean value, but we do not *have* Boolean values in the language
2. We have written both `(calc t)` and `(calc e)`, but the whole point of a conditional is that we do *not* want to evaluate both branches, only one.



# The Design Space of Conditionals

Even a simple conditional exposes many variations in language design. We normally expect to evaluate the condition and choose either the then-expression or the else-expression to evaluate next.

Even this simple description exposes three different, mostly independent design decisions:

1. What kind of values can the test-expression be? In some languages it must be a Boolean. In others, it can be about any value and some of them are considered *truthy* while the remaining ones are *falsy*.

# The Design Space of Conditionals

It seems convenient to define several truthy and falsy values. However:

Value	JavaScript	Perl	PHP	Python	Ruby
-1	truthy	truthy	truthy	truthy	truthy
0	falsy	falsy	falsy	falsy	truthy
""	falsy	falsy	falsy	falsy	truthy
"0"	truthy	falsy	falsy	truthy	truthy
NaN	falsy	truthy	truthy	truthy	truthy
nil, null, None, undefined	falsy	falsy	falsy	falsy	falsy
[]	truthy	truthy	falsy	falsy	truthy
empty map or object	truthy	falsy	falsy	falsy	truthy

# The Design Space of Conditionals

## 2. What kind of terms are the branches?

Some languages make a distinction between statements and expressions; in such languages, designers need to decide which of these are permitted.

In some languages, there are even two syntactic forms of conditional to reflect these two choices: e.g., in C, `if` uses statements (and does not return any value) while the “ternary operator” (`c?t:e`) permits expressions and returns a value.

## 3. If the branches are expressions and hence allowed to evaluate to values, how do the values relate? Many (but not all) languages with static type systems expect the two branches to have the same type. Languages without static type systems usually place no restrictions.

## Using Truthy-Falsy Values

Some languages use truthy-falsy values to handle partial functions. Instead of signaling an error, they return a falsy value when the argument cannot be handled.

For instance, it is common to return `#false` in Racket or `None` in Python as an error code, and a proper value for normal execution.

Consider:

```
(define (g s)
  (+ 1 (or (string->number s) 0)))
```

This function accepts a string that may or may not represent a number. If it does, it returns one bigger number; otherwise it returns 1:

```
(test (g "5") 6)
(test (g "hello") 1)
```

This works because `string->number` returns a number or, if the string is not legal, `#false`. In Racket, all values other than `#false` are truthy. A string that parses as a number short-circuits the `or`, while non-numeric strings result in 0.

# Implementing Conditionals

We have decisions to make! Since we only have numbers, we will say that 0 is falsy and everything else is truthy:

```
(define (calc e)
  (type-case Exp e
    [(num n) n]
    [(plus l r) (+ (calc l) (calc r))]
    [(cnd c t e) (if (zero? (calc c))
                      (calc t)
                      (calc e))]))
```

The semantics of the conditional are made explicit in the body of `calc`. If we want different semantics, this is where we would make the change.

# Implementing Conditionals

We have deferred a lot to the implementation of plait. Some thoughts:

1. This is true!
2. This is not entirely true. We have made some conscious decisions, like the handling of conditionals.
3. In fact, we have made even more decisions, whether or not we were conscious of them, such as the handling of numbers. We just happened to defer those to plait, but we could have made other decisions if we wanted.
4. This reuse is actually part of the power of an interpreter: it lets you exploit features that have already been built instead of having to re-implement all of them from scratch.
5. By reusing the host language (here, plait), we can zero in on the differences (like the handling of conditionals), which would otherwise be lost if we had to implement everything. Later we will see stronger departures from the semantics of plait.

## Adding Booleans

What if we want a proper Boolean type?

As before, we must alter the AST, the evaluator, and the parser. We can add a new AST type constructor similar to numbers:

```
(define-type Exp
  [num (n : Number)]
  [bool (b : Boolean)]
  [plus (left : Exp) (right : Exp)]
  [cnd (test : Exp) (then : Exp) (else : Exp)])
```

Important note: this is the *abstract syntax*: we are simply representing the *program that the user wrote*, not the result of its evaluation.

I.e., these constructors capture the syntactic *constants* in the source program: values like 3.14 and -1 for numbers and `#true` and `#false` for Booleans. They do *not* represent compound expressions that will *evaluate* to numbers or Booleans. We can only know what an expression will evaluate to by running it.

## Adding Booleans

For the evaluator:

```
(define (calc e)
  (type-case Exp e
    [(num n) n]
    [(bool b) b]
    [(plus l r) (+ (calc l) (calc r))]
    [(cnd c t e) (if (zero? (calc c))
                      (calc t)
                      (calc e))]))
```

Oops. It fails the type check now because `calc` is only supposed to return numbers, not Booleans.



# The Value Datatype

Instead of relying on the `Number` type from `plait` as our basic datatype, we need our own type that can reflect different kinds of values. First we will rename our `Exp` constructors so we can distinguish between constructors for simple value expressions and constructors for value types:

```
(define-type Exp
  [numE (n : Number)]
  [boolE (b : Boolean)]
  [plusE (left : Exp) (right : Exp)]
  [cndE (test : Exp) (then : Exp) (else : Exp)])
```

(nothing changed expect the *names* of the constructors)

Now we will introduce a `Value` type with the kinds of answers our evaluator can produce:

```
(define-type Value
  [numV (the-number : Number)]
  [boolV (the-boolean : Boolean)])
```

# The Value Datatype

Now we can update the type of our evaluator:

```
(calc : (Exp -> Value))
```

and the early parts are easy:

```
(define (calc e)
  (type-case Exp e
    [(numE n) (numV n)]
    [(boolE b) (boolV b)]
    ...))
```

# Updating the Evaluator

Now suppose we try to use our existing code:

```
[(plusE 1 r) (+ (calc 1) (calc r))]
```

This has two problems. The first is we can't return a number; we have to return a numV:

```
[(plusE 1 r) (numV (+ (calc 1) (calc r)))]
```

But now we run into a subtler problem. The type-checker is not happy with this program. Why?

## Updating the Evaluator

The result of `calc` is a `Value`, and `+` consumes only `Numbers`. We must make a decision here: what happens if one of the sides of `+` does not evaluate to a number?

We will build an abstraction to handle this so that we can keep the core of the interpreter relatively clean:

```
[(plusE l r) (add (calc l) (calc r))]
```

Now we can defer the logic of evaluating `+` to `add`. We must make a *semantic decision* here. Should we be allowed to “add” two `Boolean` values? What about a mix?

The least non-standard policy is to require both branches to evaluate to numbers:

```
(define (add v1 v2)
  (type-case Value v1
    [(numV n1)
     (type-case Value v2
       [(numV n2) (numV (+ n1 n2))]
       [else (error '+ "expects RHS to be a number")])])
    [else (error '+ "expects LHS to be a number")]))
```

## Updating the Evaluator

Let's turn back to the conditional:

```
[(cndE c t e) ...]
```

The core logic is similar to before: evaluate the condition and based on the result evaluate one of the other two clauses.

We have another semantic decision to make: should we strictly require Boolean values or make a truthy/falsy decision? We defer it to a helper function:

```
[(cndE c t e) (if (boolean-decision (calc c))  
                  (calc t)  
                  (calc e)))]
```

Again the least non-standard policy is to be strict about requiring a Boolean:

```
(define (boolean-decision v)  
  (type-case Value v  
    [(boolV b) b]  
    [else (error 'if "expects conditional to evaluate to a boolean")]))
```

If we change our mind the `else` clause makes it easy to see where to make a change.

# Evaluating Local Binding

Most programming languages have some notion of *local binding*:

- ▶ Binding means to associate names with values. For instance, when we call a function, the act of calling associates (“binds”) the formal parameters with the actual values.
- ▶ Local means they are limited to some region of the program, and not available outside that region.

In many languages we can write something like:

```
fun f(x):  
  y = 2  
  x + y
```

This seems clear enough.

# Evaluating Local Binding

Here is a more subtle program:

```
fun f(x):  
  for(y from 0 to 10):  
    print(x + y)  
  y
```

Is that legal? Is the `y` is still “alive” or “active” or “visible” or whatever? We would say that it depends on whether `y` is *in scope*. More specifically, we’d ask whether the last `y` is a *bound* instance of the *binding* that takes place in the `for`.

This is complicated! Many languages do rather odd, complicated, and certainly unintuitive things.

## A Syntax for Local Binding

Part of the problem is syntactic. In the example above, there is no clear beginning or ending of the scope of  $y$ .

The paranthetical syntax we are using *suggests* a clear region (between the parentheses). The implementation must also reflect this, which is not entirely trivial.



## A Syntax for Local Binding

Let us notate our syntax using BNF (Backus-Naur Form):

```
<expr> ::= <num>  
        | {+ <expr> <expr>}
```

Note that this is not quite the same as our AST. The left and right arguments to + are both just listed as `<expr>` whereas in the AST we must give each a unique name.

- ▶ BNF is divided into *terminals* and *non-terminals*. Non-terminals are placeholders that must be expanded (see `<expr>` and `<num>`).
- ▶ The *terminals* are symbols that actually appear in the language—they represent themselves, not some additional set of steps.
- ▶ There is also syntax like `::=` and `|` that are part of the BNF itself.

Now we can define an extended language:

```
<expr> ::= <num>  
        | {+ <expr> <expr>}  
        | {let1 {<var> <expr>} <expr>}
```

We are adding `let1`, which has three parts: a variable (`var`) and two expressions.

# The Meaning of Local Bindings

Here are some examples of this new construct. What do you *expect* each one to produce?

```
{let1 {x 1}  
  {+ x x}}
```

# The Meaning of Local Bindings

```
{let1 {x 1}  
  {let1 {y 2}  
    {+ x y}}}
```

# The Meaning of Local Bindings

```
{let1 {x 1}  
  {let1 {y 2}  
    {let1 {x 3}  
      {+ x y}}}}}
```

# The Meaning of Local Bindings

```
{let1 {x 1}  
  {+ x  
    {let1 {x 2} x}}}}
```

# The Meaning of Local Bindings

```
{let1 {x 1}  
  {+ {let1 {x 2} x}  
    x}}
```

# The Meaning of Local Bindings

x

# The Meaning of Local Bindings

Did you notice something? None of these programs is syntactically legal! Why?



# The Meaning of Local Bindings

We have no syntax for variables. We can *bind* them but not *use* them. Let us fix that:

```
<expr> ::= <num>
         | {+ <expr> <expr>}
         | {let1 {<var> <expr>} <expr>}
         | <var>
```

Now that they are all syntactically valid, let us return to the question of what they should evaluate to.

# The Meaning of Local Bindings

The first two programs are pretty obvious:

```
{let1 {x 1}  
  {+ x x}}
```

should evaluate to 2, and

```
{let1 {x 1}  
  {let1 {y 2}  
    {+ x y}}}
```

should evaluate to 3.

# The Meaning of Local Bindings

How about this program?

```
{let1 {x 1}
  {let1 {y 2}
    {let1 {x 3}
      {+ x y}}}}
```

Here we see the advantage of the parenthetical notation. In a more conventional syntax, this might correspond to

```
x = 1
y = 2
x = 3
x + y
```

In this syntax, do we have two different  $x$ 's? Is there one  $x$  that is bound and then modified?

The parenthetical syntax gives a clearer picture of what we want. Using substitution, we see that the inner  $x$  *shadows* the outer one, hence the result should be 5.

This example is uninteresting in that the outer  $x$  is never used. What is an example of a program with two `let` bindings of  $x$  that lets us clearly see that there are two  $x$ 's?

# The Meaning of Local Bindings

Here is one:

```
{let1 {x 1}  
  {+ x  
    {let1 {x 2} x}}}}
```

Here the syntax suggests that the left `x` in the addition should be 1, while `x` in the right expression should be shadowed and should evaluate to 2. The sum should therefore be 3.

(demo) If we write this in DrRacket using `#lang racket` we can hover over each `x` and see where it is bound.

# The Meaning of Local Bindings

Now for a more complex example:

```
{let1 {x 1}  
  {+ {let1 {x 2} x}  
    x}}
```

Substitution is a useful tool to answer this one as well. The  $x$  in the left expression is shadowed and hence should be 2.

What about the  $x$  on the right hand side of the addition, i.e., on the last line?

Consider this in a more conventional syntax:

```
x = 2
```

This is pretty ambiguous: is it a new *binding* or a *modification* of the outer  $x$ ? Those are two very different things.

## The Meaning of Local Bindings

With our syntax it's much clearer that it *should* be a new binding. Thus by substitution the outer  $x$  is replaced by 1 giving

```
{+ {let1 {x 2} x}
  1}
```

with one more substitution we get

```
{+ 2
  1}
```

(demo) Again let us try this in DrRacket:

```
(let ([x 1])
  (+ (let ([x 2])
      x)
     x))
```

This only leaves our last example program:

```
x
```

Because  $x$  is not bound anywhere, this is a syntax error.

# Static Scoping

The program

```
{let1 {x 1}  
  {+ {let1 {x 2} x}  
    x}}
```

introduces us to a very important concept: a variable's binding is determined by *its position in the source code program*, and **not** by *the order of the program's execution*.

The `x` on the last line is bound by the same place—and hence obtains the same value—irrespective of other bindings that took place before it was evaluated.

# Static Scoping

Let us see a progression of programs:

```
{let1 {x 1}  
  {+ {let1 {x 2} x}  
    x}}
```

It might seem okay if it produces either 3 or 4. How about this?

```
{let1 {x 1}  
  {+ {if true  
      {let1 {x 2} x}  
      4}  
    x}}
```

The conditional is always true, so clearly we are always going to evaluate the inner binding so the answer should be the same as for the previous program (regardless of whether that was 3 or 4).



# Static Scoping

How about this?

```
{let1 {x 1}
  {+ {if true
      4
      {let1 {x 2} x}}
    x}}
```

Since the conditional is never taken, you probably don't want the inner binding to have an influence. Again the outcome here is reasonable assuming you want to *let the program's control flow influence the bindings*.

# Static Scoping

What about this program?

```
{let1 {x 1}  
  {+ {if {random}  
      4  
      {let1 {x 2} x}}  
  x}}
```

or

```
{let1 {x 1}  
  {+ {if {moon-is-currently-full}  
      4  
      {let1 {x 2} x}}  
  x}}
```

Are you okay with the variable binding decision changing every two weeks?

# Static Scoping

What about this version?

```
{let1 {x 1}
  {+ {if {moon-is-currently-full}
        4
      {let1 {y 2} x}}
    y}}
```

Now, depending on the phase of the moon, the program either produces an answer or results in an unbound variable error.

- ▶ The decision to let control flow determine binding is called *dynamic scope*.
- ▶ It is the one unambiguously wrong design decision in programming languages.
- ▶ It has a long and sordid history: the original Lisp had it, and it was not until over a decade later that Scheme fixed it.
- ▶ Unfortunately, those who don't know history are doomed to repeat it: early versions of Python and JavaScript also had dynamic scope. Taking it back out has been a herculean effort.

# Static Scoping

Dynamic scope means:

- ▶ We can't be sure about the binding structure of our programs
- ▶ The evaluator can't be sure, either
- ▶ Nor can programmer tools

For instance, a program refactoring tool needs to know binding structure: even a simple “variable renaming” tool needs to know which variables to rename.

The opposite of dynamic scope—where we can determine the binding by following the structure of the AST—is called *static scope* or *lexical scope*. Static scope is a defining characteristic of SMoL.

Dynamic scope occurred in early implementations because it was easy to obtain: it was the default behavior. We have to work a bit harder to obtain static scope, as we will see.

# An Evaluator for Local Binding

Now that we know the behavior we want, let us implement it. We will also start calling our calculator an *interpreter* to reflect its growing sophistication.

Let us start with the AST. For simplicity we will ignore conditionals, which are orthogonal to local binding:

```
(define-type Exp
  [numE (n : Number)]
  [plusE (left : Exp) (right : Exp)]
  [varE (name : Symbol)]
  [let1E (var : Symbol)
        (value : Exp)
        (body : Exp)])
```

# An Evaluator for Local Binding

We can copy over our previous calculator, but we pretty quickly run into trouble:

```
(define (interp e)
  (type-case (Exp) e
    [(numE n) n]
    [(varE s) ...]
    [(plusE l r) (+ (interp l) (interp r))]
    [(let1E var val body) ...]))
```

What do we do when we encounter a `let1E`? What about when we encounter a variable?

# Caching Substitution

Substitution is useful method for understanding program semantics, but it is messy and slow as an *evaluation* technique. It requires us to keep rewriting the program text for *every* variable binding.

Instead, we add a *cache* of substitutions in a data structure called the *environment*, which records names and their corresponding values: that is, it's a collection of key-value pairs. Note: this changes *how* we produce an answer, but not *what* answer to produce: we should match the outcome we would get from substitution.

We will use a hash table to represent the environment:

```
(define-type-alias Env (Hashof Symbol Value))  
(define mt-env (hash empty)) ;; "empty environment"
```

We will need the interpreter to take an environment as a formal parameter:

```
(interp : (Exp Env -> Value))  
(define (interp e nv) ...)
```

## Caching Substitution

Now what happens when we encounter a variable? We try to look it up in the environment. A helper function is a good place to decide what happens if the lookup fails:

```
(define (lookup (s : Symbol) (n : Env))  
  (type-case (Optionof Value) (hash-ref n s)  
    [(none) (error s "not bound")]  
    [(some v) v]))
```

In the event the lookup succeeds, then we want the value found, which is wrapped in some. This function enables our interpreter to stay very clean and readable:

```
[(varE s) (lookup s nv)]
```



## Caching Substitution

Finally, we are ready to tackle `let1`. What happens here? We must

- ▶ evaluate the body of the expression, in
- ▶ an environment that has been extended, with
- ▶ the new name
- ▶ bound to its value.

Fortunately, this isn't as bad as it sounds. Again, a function will help a lot:

```
(extend : (Env Symbol Value -> Env))  
(define (extend old-env new-name value)  
  (hash-set old-env new-name value))
```

With this, we can see the structure clearly:

```
[(let1E var val body)  
 (let ([new-env (extend nv  
                        var  
                        (interp val nv))])
```

(observe that we used `let` in `plait` to define `let1`)

# Caching Substitution

In sum, our core interpreter is now:

```
(define (interp e nv)
  (type-case Exp e
    [(numE n) n]
    [(varE s) (lookup s nv)]
    [(plusE l r) (+ (interp l nv) (interp r nv))]
    [(let1E var val body)
     (let ([new-env (extend nv
                             var
                             (interp val nv))])
       (interp body new-env))]))
```

1. What if we had not called `(interp val nv)` above?
2. What if we'd used `nv` instead of `new-env` in the call to `interp`?
3. Are there any other errors in the interpreter based on copying what we had before?
4. We seem to extend the environment but never remove anything from it. Is that okay? If not, it should cause an error. What program would demonstrate this error, and does it actually do so? (If not, why not?)

# Functions in the Language

There are many ways to think about adding functions to the language. Many languages, for instance, have top-level functions; e.g.:

```
fun f(x):  
  x + x
```

Indeed, some languages (such as C) only have top-level functions. Most modern languages, however, have the ability to write functions outside the top-level: e.g.,

```
fun f(x):  
  fun sq(y):  
    y * y  
  sq(x) + sq(x)
```

# Functions in the Language

Most modern languages even have the ability to *return* functions and to allow them to be written *anonymously*:

```
fun f(x):  
  sq = lam(y): y * y  
  sq(x) + sq(x)
```

This is such a common feature of modern languages that we will think of it as a component of SMoL.

We could have used the same construct to define `f` as well: a name-binding with a `lam`.

## Extending the Representation

What do we need to evaluate functions-as-values in SMoL?

- ▶ We can use `let1`, so functions do not inherently need a name
- ▶ For simplicity we will assume functions take only one argument

**Exercise:** What issues might we have to deal with when we extend functions from having one argument only to having multiple arguments?

## Extending the Representation

First we need to extend our abstract syntax.

**Do Now:** How many new constructs do we need to add to the abstract syntax?

When we added `let`, you may recall that it didn't suffice to add one construct; we needed two:

- ▶ One for *binding*
- ▶ One for *use*

This is a common pattern: when adding values to the language we can expect to need a way to *make* them and a way to *use* them.

So we need a way to represent both

`lam(x): x * x`

for defining new functions, and

`sq(3)`

to use them.

## Extending the Representation

We therefore add

```
[lamE (var : Symbol) (body : Exp)]  
[appE (fun : Exp) (arg : Exp)]
```

to our AST. Assuming we have already extended the parser:

These two programs:

```
{let1 {f {lam x {+ x x}}}  
      {f 3}}
```

```
{let1 {x 3}  
      {let1 {f {lam y {+ x y}}}  
            {f 3}}}
```

and should both evaluate to 6.

parse, respectively, into

```
(let1E 'f (lamE 'x (plusE (varE 'x) (varE 'x)))  
      (appE (varE 'f) (numE 3)))
```

```
(let1E 'x (numE 3)  
      (let1E 'f (lamE 'y (plusE (varE 'x) (varE 'y))  
                    (appE (varE 'f) (numE 3)))))
```

# Evaluating Functions

Now let's think about the interpreter, starting with a simple example:

```
{lam x {+ x x}}
```

which is represented as

```
(lamE 'x (plusE (varE 'x) (varE 'x)))
```

**Do Now:** What do we want this program to evaluate to? Think in terms of types!

Remember that `interp` produces a `Value`, which can be a *number* or a *Boolean*. What kind of value does the above expression evaluate to?



# Evaluating Functions

Look at what some other languages do:

```
> (lambda (x) (+ x x))  
#<procedure>  
> (number? (lambda (x) (+ x x)))  
#f
```

```
>>> lambda x: x + x  
<function <lambda> at 0x108fd16a8>  
>>> isinstance(lambda x: x + x, numbers.Number)  
False
```

Both Racket and Python agree: the result of creating an anonymous function is a function-kind of value, not a number. What this says is that we have to broaden the kinds of values that `interp` can produce.

# Evaluating Functions

Some terminology:

- ▶ A *side-effect* is a change to the system that is visible from outside the body of a function. Typical side-effects are modifications to variables that are defined outside the function, communication with a network, changes to files, and so on.
- ▶ A function is *pure* if, for a given input, it always produces the same output, and has no side-effects. In reality, a computation always has *some* side-effects, such as the consumption of energy and production of heat, but we usually overlook these because they are universal. In a few settings, however, they can matter: e.g., if a cryptographic key can be stolen by measuring these side-effects.
- ▶ Traditionally, some languages have used the terms *procedure* and *function* for similar but not identical concepts. Both are function-like entities that encapsulate a body of code and can be applied (or “called”). A procedure is an encapsulation that does not produce a value; therefore, it must have side-effects to be of any use. In contrast, a function always produces a value (and may be expected to not have any side-effects). This terminology has gotten completely scrambled over the years and people now use the terms interchangeably, but if someone seems to be making a distinction between the two, they probably mean something like the above.

## Extending Values

What happens when evaluating a function? Both Racket and Python seem to suggest that we return a function.

We can start by storing no additional information about the function:

```
(define-type Value
  [numV (the-number : Number)]
  [boolV (the-boolean : Boolean)]
  [funV])
```

(This syntax means `funV` is a constructor of no parameters. It is a kind of `Value` but conveys no additional information). But now think about a program like this (assuming `x` is bound):

```
{{if0 x
  {lam x {+ x 1}}
  {lam x {- x 2}}}}
5}
```

In both cases we're going to get a `funV` value with no additional information, so when we try to perform the application, we... can't.

## Extending Values

So it is clear that the function value needs to tell us about the function. We need:

- ▶ the body expression, because that is what we need to evaluate
- ▶ the name of the formal parameter, since the body probably references it

We need something like

```
(define-type Value
  [numV (the-number : Number)]
  [boolV (the-boolean : Boolean)]
  [funV (var : Symbol) (body : Exp)])
```

At this point it may seem like we are overcomplicating this question. We take numeric and Boolean values and simply re-wrap them in new constructors, and now we are doing the same thing for functions.

Patience.

## Extending Values

With what we have, we can already write a functioning interpreter. The `lam` case is simple:

```
[(lamE v b) (funV v b)]
```

The application case is a bit more detailed. We need to:

1. Evaluate the function position, to figure out what kind of value it is.
2. Evaluate the argument position, since we've agreed that's what happens in SMoL.
3. Check that the function position really does evaluate to a function. If it does not, raise an error.
4. Evaluate the body of the function. But because the body can refer to the formal parameter. . .
5. . . first make sure the formal is bound to the actual value of the argument.

## Extending Values

Codifying the application case in stages:

```
[(appE f a) (let ([fv (interp f nv)]  
                  [av (interp a nv)])  
              ...)]
```

```
[(appE f a) (let ([fv (interp f nv)]  
                  [av (interp a nv)])  
              (type-case Value fv  
                [(funV v b) ...]  
                [else (error 'app "didn't get a function")])))]
```

```
[(appE f a) (let ([fv (interp f nv)]  
                  [av (interp a nv)])  
              (type-case Value fv  
                [(funV v b)  
                 (interp b ...)]  
                [else (error 'app "didn't get a function")])))]
```

## Extending Values

```
[(appE f a) (let ([fv (interp f nv)]  
                  [av (interp a nv)])  
  (type-case Value fv  
    [(funV v b)  
     (interp b (extend nv v av))]  
    [else (error 'app "didn't get a function")])))]
```

## Stepping Back

Putting it all together, we get the following interpreter:

```
(interp : (Exp Env -> Value))

(define (interp e nv)
  (type-case Exp e
    [(numE n) (numV n)]
    [(varE s) (lookup s nv)]
    [(plusE l r) (add (interp l nv) (interp r nv))]
    [(lamE v b) (funV v b)]
    [(appE f a) (let ([fv (interp f nv)]
                      [av (interp a nv)])
                  (type-case Value fv
                    [(funV v b)
                     (interp b (extend nv v av))]
                    [else (error 'app "didn't get a function")]))])
  [(let1E var val body)
   (let ([new-env (extend nv
                          var
                          (interp val nv))])
     (interp body new-env))]))
```



## Stepping Back

**Exercise:** We wrote down a particular ordering above, which we put into practice in the code. But is that the same ordering that actual languages use? In particular, are non-function errors reported after or before evaluating the argument? Experiment and find out!

## Stepping Back

Since we've taken several steps to get here, it's easy to lose sight of what we've just done. In just 20 lines of code (with a few helper functions), we have described the implementation of a full programming language. Not only that, a language that can express all computations. When Turing Award winner Alan Kay first saw the equivalent program, he says,

*Yes, that was the big revelation to me when I was in graduate school—when I finally understood that the half page of code on the bottom of page 13 of the Lisp 1.5 manual was Lisp in itself. These were “Maxwell’s Equations of Software!” This is the whole world of programming in a few lines that I can put my hand over.*

*I realized that anytime I want to know what I’m doing, I can just write down the kernel of this thing in a half page and it’s not going to lose any power. In fact, it’s going to gain power by being able to reenter itself much more readily than most systems done the other way can possibly do.*

We've just rediscovered this same beautiful, powerful idea!

# Stepping Back

evalquote is defined by using two main functions, called eval and apply. apply handles a function and its arguments, while eval handles forms. Each of these functions also has another argument that is used as an association list for storing the values of bound variables and function names.

```
evalquote[fn;x] = apply[fn;x;NIL]
```

where

```
apply[fn;x;a] =
  [atom[fn] → [eq[fn;CAR] → caar[x];
    eq[fn;CDR] → cdar[x];
    eq[fn;CONS] → cons[car[x];cadr[x]];
    eq[fn;ATOM] → atom[car[x]];
    eq[fn;EQ] → eq[car[x];cadr[x]];
    T → apply[eval[fn;a];x;a]];
  eq[car[fn];LAMBDA] → eval[caddr[fn];pairlis[cadr[fn];x;a]];
  eq[car[fn];LABEL] → apply[caddr[fn];x;cons[cons[cadr[fn];
    caddr[fn]];a]]]
```

```
eval[e;a] = [atom[e] → cdr[assoc[e;a]];
  atom[car[e]] →
    [eq[car[e];QUOTE] → cadr[e];
    eq[car[e];COND] → evcon[cdr[e];a];
    T → apply[car[e];evlis[cdr[e];a];a]];
  T → apply[car[e];evlis[cdr[e];a];a]]
```

pairlis and assoc have been previously defined.

```
evcon[c;a] = [eval[caar[c];a] → eval[cdadr[c];a];
  T → evcon[cdr[c];a]]
```

and

```
evlis[m;a] = [null[m] → NIL;
  T → cons[eval[car[m];a];evlis[cdr[m];a]]]
```

Figure 1: Page 13 of the Lisp 1.5 manual

## Extending Tests

Well, actually, we shouldn't be too happy. Consider the following:

```
(let1E 'x (numE 1)
  (let1E 'f (lamE 'y (varE 'x))
    (let1E 'x (numE 2)
      (appE (varE 'f) (numE 10))))))
```

What do we expect it to produce? If in doubt, we can write the same thing as a Racket program:

```
(let ([x 1])
  (let ([f (lambda (y) x)])
    (let ([x 2])
      (f 10))))
```

In Racket, the inner binding of `x` does *not* override the outer one, the one that was present at the time the function bound to `f` was defined. Therefore, this produces 1 in Racket.

We should want this!

## Extending Tests

Consider this program:

```
(let1E 'f (lamE 'y (varE 'x))  
  (let1E 'x (numE 1)  
    (appE (varE 'f) (numE 10))))
```

This corresponds to

```
(let ([f (lambda (y) x)])  
  (let ([x 1])  
    (f 10)))
```

which has an unbound identifier ( $x$ ) error. But our interpreter produces 1 instead of halting with an error, which leads us right back to **dynamic scope**!

## Return to Static Scope

So how do we fix this? The examples above actually give us a clue, but there is another source of inspiration as well. Do you remember that we started with *substitution*? We'll walk through these examples in Racket, so that you can run each of them directly and check that they produce the same answer. Consider again this program:

```
(let ([x 1])
  (let ([f (lambda (y) x)])
    (let ([x 2])
      (f 10))))
```

Substituting 1 for *x* produces:

```
(let ([f (lambda (y) 1)])
  (let ([x 2])
    (f 10)))
```

## Return to Static Scope

Substituting  $f$  produces:

```
(let ([x 2])  
  ((lambda (y) 1) 10))
```

Finally, substituting  $x$  with 2 produces (note that there are no  $x$ s left in the program!):

```
((lambda (y) 1) 10)
```

When you see it this way, it's clear *why* the later binding of  $x$  should have no impact: it's a different  $x$ , and the earlier  $x$  has effectively already been substituted. Since we have agreed that substitution is how we want our programs to work, our job now is to make sure that the environment actually implements that correctly.

## Return to Static Scope

The way to do it is to recognize that the environment represents the substitutions waiting to happen, and just *remembers* them. That is, our representation of a function needs to also keep track of the environment at the moment of function creation:

```
(define-type Value
  [numV (the-number : Number)]
  [boolV (the-boolean : Boolean)]
  [funV (var : Symbol) (body : Exp) (nv : Env)])
```

This new and richer kind of `funV` value has a special name: it's called a *closure*. That's because the expression is “closed” over the environment in which it was defined.



## Return to Static Scope

**Terminology:** A *closed* term is one that has no unbound variables. The body of a function may have unbound variables—like  $x$  above—but the closure makes sure that they aren't *really* unbound, because they can get their values from the stored environment.

**Quote:** “Save the environment! Create a closure today!” —Cormac Flanagan

**Quote:** “Lambdas are relegated to relative obscurity until Java makes them popular by not having them.” —James Iry, *A Brief, Incomplete, and Mostly Wrong History of Programming Languages*

## Return to Static Scope

That means, when we create a closure, we have to record the environment at the time of its creation:

```
[(lamE v b) (funV v b nv)]
```

Finally, when we use a function (represented by a closure), we have to make sure we use the *stored* environment, not the one present at the point of calling the function, which is the *dynamic* one:

```
[(appE f a) (let ([fv (interp f nv)]  
                  [av (interp a nv)])  
  (type-case Value fv  
    [(funV v b nv)  
     (interp b (extend nv v av))]  
    [else (error 'app "didn't get a function")])))]
```

Just to be clear: in the code above, the `nv` in the `funV` case *intentionally* shadows the `nv` bound at the top of the interpreter. Thus, the call to `extend` extends the environment from the closure, rather than the one present at the point of the call.

## Return to Static Scope

**Exercise:** Notice that the function and argument expressions ( $f$  and  $a$ , respectively) are evaluated in the environment given to the interpreter, not the one inside the closure. Is this correct? Or should they be using the closure's environment?

## A Subtle Test

In the examples above, we always use the closure in the scope in which it was defined. However, our language is actually more powerful than that: we can return a closure and use it outside the scope in which it was defined. Here's a sample Racket program:

```
((let ([x 3])  
  (lambda (y) (+ x y)))  
  4)
```

**Do Now:** Take a moment to read it carefully. What should it produce?

## A Subtle Test

First we bind the  $x$ , then we evaluate the lambda. This creates a closure that remembers the binding to  $x$ . This closure is the value returned by this expression:

```
((let ([x 3])
  (lambda (y) (+ x y)))
 4)
```

This value is now applied to 4. It's legal to do this, because the value returned is a function. When we apply it to 4, that evaluates the sum of 4 and 3, producing 7. Sure enough, translating this and sending it to our interpreter produces 7:

```
(test (interp (appE (let1E 'x (numE 3)
                      (lamE 'y (plusE (varE 'x) (varE 'y))))
                (numE '4))
      mt-env)
(numV 7))
```

## A Subtle Test

Here's another test to try out, written as a Racket program:

```
((let ([y 3])  
  (lambda (y) (+ y 1)))  
  5)
```

What does it produce in Racket? Translate it and try it in your interpreter.