

Principles of Programming Languages

Operational Semantics: Substitution Model for Procedure Application

We continue our exploration of operational semantics by addressing the case of procedure application. The *L2* language extends *L1* by introducing:

- User defined procedures
(`lambda` expressions)
- Conditional expressions
(`if` expressions)

These two constructs combine well - so that we can construct recursive functions (which require a condition between the base case and the recursive case).

The syntax of *L2* extends that of *L1* with two new expression types - **if-exp** and **proc-exp**:

L2 Syntax

```
<program> ::= (L2 <exp>+) // program(exps:List(exp))
<exp> ::= <define-exp> | <cexp>
<define-exp> ::= (define <var-decl> <cexp>)
                  // def-exp(var:var-decl, val:cexp)
<cexp> ::= <num-exp> // num-exp(val:Number)
          | <bool-exp> // bool-exp(val:Boolean)
          | <prim-op> // prim-op(op:string)
          | <var-ref> // var-ref(var:string)
          | (if <exp> <exp> <exp>)
              // if-exp(test,then,else) ##### L2
          | (lambda (<var-decl>*) <cexp>+)
              // proc-exp(params:List(var-decl), body:List(cexp)) ##### L2
          | (<cexp> <cexp>*)
              // app-exp(rator:cexp, rands:List(cexp))
<prim-op> ::= + | - | * | / | < | > | = | not
<num-exp> ::= a number token
<bool-exp> ::= #t | #f
<var-ref> ::= an identifier token
<var-decl> ::= an identifier token
```

A program in *L2* can now use user defined procedures such as:

```
(L2  
  (define square (lambda (x) (* x x)))  
  (+ (square 2) (square 3)))
```

To determine which values can be computed by L2 programs, we proceed inductively on the structure of L2-ASTs. The same values as those in L1 can be computed, and we must analyze the two new types of expressions:

- **IfExp** expressions return the value of either the then-branch or the else-branch, which can be any **CExp** expressions. Thus **IfExp** does not compute new types of values.
- **ProcExp** expressions return a new type of value - which we have called a **closure**.

We must then extend the definition of the Value type to include closure values:

```
Value = Number | Boolean | Prim-op  
      | Void    | Closure
```

We define the closure data type as a record with two fields:

- Params: a list of **VarDecl** values
- Body: a list of **CExp** values

```
Closure ::= (Closure (<var-decl>*) <cexp>+)  
// closure(params: List(var-decl), body: List(cexp))
```

Note that a **ProcExp**

(**lambda** (x) (* x x))

is an **expression** while a closure is a **value**.

They are of 2 different types - closures are the result of a computation.

The evaluation rules that define the operational semantics of $L2$ include the same rules as those of $L1$ for the $L1$ expression types. We must specify evaluation rules for the new expression types:

Evaluation of Conditional Expressions

- `eval(IfExp(test, then, alt), env) =>`
 ;; test, then, alt are of type CExp
 let c: Value = eval(test, env)
 If c is considered a true value:
 return eval(then, env)
 else
 return eval(alt, env)

Note that we must define what counts as a true value in the testing of the condition in an **IfExp**. This definition of what counts as true is a semantic decision - which is different in different languages.

Evaluation of Conditional Expressions

In Scheme, a true value is anything that is not `#f`. We implement this in this procedure in our interpreter code:

```
// Purpose: Define what is considered a true value  
// in an IfExp  
export const isTrueValue = (x: Value): boolean =>  
    ! (x === false);
```

Evaluation of Conditional Expressions

In JavaScript, the definition of what counts as true in an **IfExp** is more complicated - it is any value that is:

- not **false**
- not **undefined**
- not **null**
- not **+0**, **-0** or **NaN**
- not an empty string (**" "**)

- `eval(ProcExp(params, body), env) =>`
 `;; Construct a closure value`
 `return makeClosure(params, body)`

Observe that when we compute the value of a procedure expression, there is no actual computation going on besides the packaging of the parameters and the body into a closure record.

The body is not computed at this stage. It will only be computed when the procedure is actually applied to arguments.

This property is important: it means we can **delay** the computation of an expression by wrapping it inside a procedure, and invoking the procedure only later. We will exploit this property in Chapter 4.

We have covered the two new types of expressions with the evaluation rules above – but another place in the operational semantics must now be updated because of the presence of closures: procedure application.

In the $L1$ case, the only procedure value that could be applied to arguments was a primitive operator. This was covered in the evaluation rule for **AppExp** expressions:

- `eval(AppExp(rator, rands)) =>`
 `;; rator is of type CExp`
 `;; rands is of type CExp[]`
 `let proc = eval(rator)`
 `let args = map(eval, rands)`
 `return applyProc(proc, args)`

`applyProc` defines how a procedure value is applied to values. We must now define how a closure value is applied to argument values.

Observe that in order to evaluate an **AppExp** whose operator evaluates to a closure, we followed the same process as for primitive procedures: first evaluate the elements of the combination, then apply the closure (which is the value of the operator) to the arguments (which are the values of the operands).

This strategy is what is called **applicative order evaluation**. It is the standard evaluation strategy in most programming languages, but we will discuss an alternative strategy in the next section (*normal evaluation*).

To apply a closure to arguments, we define the **substitution model**: evaluate the body of the closure with each formal parameter replaced by the corresponding argument.

Let's follow an example:

```
(define square (lambda (x) (* x x)))  
(square 5)
```

The evaluation process is the following:

Procedure Application

1. Evaluate defineExp:

1.1 Evaluate `(lambda (x) (* x x)) => (closure (x) (* x x))`

1.2 Bind square to the value `(closure (x) (* x x))`
in the global environment

2. Evaluate `(square 5)` (an AppExp) in the global environment:

2.1 Evaluate square (a VarRef expression) `=> (closure (x) (* x x))`

2.2 Evaluate 5 (a NumExp expression) `=> 5`

2.3 `ApplyProc[(closure (x) (* x x)) (5)]`

Procedure Application

```
;; To clarify the process let us use full AST for the closure elements
;; (closure [(VarDecl x)]
;;          [(AppExp (PrimOp *) [(VarRef x), (VarRef x)])])
2.3.1 Substitute the VarRef free occurrences of the VarDecl in body
    with the corresponding value
    Substituted-body = [(AppExp (PrimOp *) [5, 5])]
2.3.2 Evaluate the resulting substituted body:
    return eval(substituted-body)
    2.3.2.1 Eval (PrimOp *) => (PrimOp *)
    2.3.2.2 Eval 5 => 5
    2.3.2.3 Eval 5 => 5
    2.3.2.4 ApplyProc( (PrimOp *), [5, 5] )
        2.3.2.4.1 ApplyPrimitive( (PrimOp *), [5, 5] ) => 25
```

We must clarify three aspects of the applicative-eval substitution model:

- Type of the substituted elements into the AST of the body
- Substitute only free occurrences of the params in the body
- Need to rename bound variables inside the body to avoid variable capture

Let us analyze the types of the objects manipulated in the substitution operation above:

```
;; (closure [(VarDecl x)]  
;;          [(AppExp (PrimOp *) [(VarRef x), (VarRef x)])])  
2.3.1 Substitute the VarRef free occurrences of the VarDecl in body  
      with the corresponding value  
      Substituted-body = [(AppExp (PrimOp *) [5, 5])]
```

Substitute Expressions instead of Values

The `applyProc` procedure receives arguments which are all of type `Value` (`proc` is a `Value` which can be either a `PrimOp` or a `Closure` value, `rands` is a list of `Values`).

Substitute Expressions instead of Values

The body of the closure is a list of **CExp** expressions. Our objective is to replace all **VarRef** occurrences in the body with the corresponding values of the arguments (in our example, we want to replace (**VarRef** *x*) with 5).

Substitute Expressions instead of Values

There is a typing problem with this operation: `5` is a `Value`, while `(VarRef x)` is an expression. If we replace `(VarRef x)` with the value `5` (a number), the resulting body is not a valid AST.

To address this discrepancy, we must map the values of the arguments to corresponding expressions. This mapping is performed in our interpreter with the following function:

Substitute Expressions instead of Values

```
const valueToLitExp =  
  (v: Value): NumExp | BoolExp | StrExp | LitExp |  
    PrimOp | ProcExp =>  
    isNumber(v) ? makeNumExp(v) :  
    isBoolean(v) ? makeBoolExp(v) :  
    isString(v) ? makeStrExp(v) :  
    isPrimOp(v) ? v :  
    isClosure(v) ? makeProcExp(v.params, v.body) :  
    makeLitExp(v);
```

As a result, the closure application above is processed as follows:

```
;; (closure [(VarDecl x)]  
;;          [(AppExp (PrimOp *) [(VarRef x), (VarRef x)])])  
2.3.1 Substitute the VarRef free occurrences of the VarDecl in body  
      with the corresponding value  
      Substituted-body = [(AppExp (PrimOp *) [NumExp(5), NumExp(5)])]
```

and we confirm that the resulting substituted-body is a valid AST.

In summary, the substitution procedure has
type:

```
// @Pre: vars and exps have the same length  
const substitute = (body: CExp[],  
                   vars: string[],  
                   exps: CExp[]): CExp[]
```

Substitute Only Free Variable Occurrences in the Body

When we apply a closure to arguments, we consider the body and the params of the closure separately. If you look at

`(closure (x) (* x y))`

in the body of the closure - the variable `x` occurs bound (it is bound by the parameter of the closure) and the variable `y` occurs free.

Substitute Only Free Variable Occurrences in the Body

If we now look at the body *separately* -
 $(* \ x \ y)$ - then the variables which were
bound to the params now appear free in the
body.

These variable references are the occurrences
we must replace with the value of the
argument.

Substitute Only Free Variable Occurrences in the Body

Consider the case of another bound occurrence of the x variable in the body as in this example:

```
(closure (x) ; 1
  ((lambda (x) (* x x)) ; 2
    (+ x x))) ; 3
```

Substitute Only Free Variable Occurrences in the Body

```
(closure (x)                                ; 1
  ((lambda (x) (* x x))                     ; 2
   (+ x x)))                                ; 3
```

In this case, the **VarRef** occurrences in line 2 are bound in the body to the **VarDecl** in line 2, while the occurrences in line 3 are free.

When we apply this closure to the value 2, we must replace the free occurrences in line 3 but leave those in line 2 unchanged.

Substitute Only Free Variable Occurrences in the Body

The substitution algorithm is implemented in the following function - which is a typical syntax-driven function, which traverses a list of ASTs, and recursively transforms the nodes. The only expression type where an actual transformation is performed is **VarRef**.

Substitute Only Free Variable Occurrences in the Body

Observe how the code of the transformation is similar to the code of **applyEnv** we discussed in the previous lecture.

When traversing a **ProcExp** within the body (as in the example we just reviewed above), substitute removes from the list of variables to be substituted the variables which are now bound by the new **VarDecls** of the **ProcExp**. This is performed with the call to **filter**.

Substitute Only Free Variable Occurrences in the Body

```
// @Pre: vars and exps have the same length  
const substitute =  
  (body: CExp[],  
   vars: string[],  
   exps: CExp[]): CExp[] => {  
    // ...  
    return map(sub, body);  
  }
```

Substitute Only Free Variable Occurrences in the Body

```
const sub = (e: CExp): CExp =>
  isNumExp(e) ? e :
  isBoolExp(e) ? e :
  isPrimOp(e) ? e :
  isLitExp(e) ? e :
  isStrExp(e) ? e :
  isVarRef(e) ? subVarRef(e) :
  isIfExp(e) ? makeIfExp(sub(e.test),
                          sub(e.then),
                          sub(e.alt)) :
  isProcExp(e) ? subProcExp(e) :
  isAppExp(e) ? makeAppExp(sub(e.rator),
                           map(sub, e.rands)) :
  e;
```

Substitute Only Free Variable Occurrences in the Body

```
const subVarRef = (e: VarRef): CExp => {
  const pos = indexOf(e.var, vars);
  return ((pos > -1) ? exps[pos] : e);
};

const subProcExp = (e: ProcExp): ProcExp => {
  const argNames = map((x) => x.var, e.args);
  // zip creates a list of pairs (x_i, y_i)
  // given 2 lists (x_i) and (y_i)
  const subst = zip(vars, exps);
  // Do not substitute vars which are bound
  // by the args of the proc
  const freeSubst = filter((ve => indexOf(first(ve), argNames) === -1,
    subst);
  return makeProcExp(e.args,
    substitute(e.body,
      map(first, freeSubst),
      map(second, freeSubst)));
};
```

Avoid Capturing Free Variables During Substitution

Consider the following program:

```
(define z (lambda (x) (* x x)))  
  
(((lambda (x) (lambda (z) (x z))) ; 1  
  (lambda (w) (z w)))           ; 2  
 2)
```

If we apply the substitution model as presented so far when computing the 2nd expression - we replace `x` with the expression

`(lambda (w) (z w))` (which is the parameter passed as the `x` argument in line 1).

Avoid Capturing Free Variables During Substitution

The resulting substituted body is:

```
(lambda (z) ((lambda (w) (z w)) z))
```

Avoid Capturing Free Variables During Substitution

The problem in this substitution is that the inner VarRef `z` coming from the function `(lambda (w) (z w))` is now captured by the `(lambda (z) ...)` context in which we operated the substitution.

As a result, this `z` VarRef now refers to the `(lambda (z) ...)` VarDecl instead of referring to the global `(define z ...)` VarDecl as it should.

Avoid Capturing Free Variables During Substitution

This effect is called **free variable capture** and we must avoid it.

Avoid Capturing Free Variables During Substitution

The simplest solution to address this problem is to ensure that **before we perform substitution**, we **rename consistently** all the bound variables that occur in the body with fresh names.

This solution relies on the observation we mentioned in the lecture about syntactic operations that the actual name of variables does not modify the semantics of expressions as long as their lexical address remains consistent.

Avoid Capturing Free Variables During Substitution

The renaming algorithm is performed consistently through a syntax-driven traversal of the body AST. It is implemented in the following manner in the interpreter. Note that the only type of expressions which are transformed in this AST transformation are **ProcExp** constituents.

Avoid Capturing Free Variables During Substitution

```
/*
```

Purpose: create a generator of new strings of the form `v__n` with `n` is incremented at each call.

Note the typical usage of a closure with side effect of the closed variable.

Example:

```
const gen = makeVarGen();
```

```
console.log(gen("v")) => "v__1"
```

```
console.log(gen("v")) => "v__2"
```

```
*/
```

```
const makeVarGen = (): (v: string) => string => {
```

```
  let count: number = 0;
```

```
  return (v: string) => {
```

```
    count++;
```

```
    return `${v}__${count}`;
```

```
  };
```

```
};
```

Avoid Capturing Free Variables During Substitution

```
const renameExps = (exps: CExp[]): CExp[] => {  
  // ...  
  return map(replace, exps);  
};
```

Avoid Capturing Free Variables During Substitution

```
const replace = (e: CExp): CExp =>
  isIfExp(e) ? makeIfExp(replace(e.test),
                        replace(e.then),
                        replace(e.alt)) :
  isAppExp(e) ? makeAppExp(replace(e.rator),
                          map(replace, e.rands)) :
  isProcExp(e) ? replaceProc(e) :
  e;
```

Avoid Capturing Free Variables During Substitution

```
const varGen = makeVarGen();
const replaceProc = (e: ProcExp): ProcExp => {
  const oldArgs = map((arg: VarDecl): string => arg.var,
    e.args);
  const newArgs = map(varGen, oldArgs);
  const newBody = map(replace, e.body);
  return makeProcExp(map(makeVarDecl, newArgs),
    substitute(newBody,
      oldArgs,
      map(makeVarRef, newArgs)));
}
```

Putting all elements discussed above together,
the apply-proc procedure implements the
following algorithm:

The Apply Procedure Summarized

```
const L3applyProcedure =  
  (proc: Value, args: Value[], env: Env): Result<Value> =>  
    isPrimOp(proc) ? applyPrimitive(proc, args) :  
    isClosure(proc) ? applyClosure(proc, args, env) :  
    makeFailure("Bad procedure " + JSON.stringify(proc));  
  
const applyClosure =  
  (proc: Closure, args: Value[], env: Env): Result<Value> => {  
    let vars = map((v: VarDecl) => v.var, proc.params);  
    let body = renameExps(proc.body);  
    let litArgs = map(valueToLitExp, args);  
    return evalSequence(substitute(body, vars, litArgs), env);  
  }
```

Observe that:

1. we make sure the body is renamed so that we avoid capturing free variables
2. we map arguments to lit-exps
3. then we perform the substitution

When describing the apply procedure operation, we use the following terminology:

- Substitute formal parameters
- Reduce (evaluate the substituted body)

Let us review the properties of the two operations we defined over ASTs: Renaming and Substitution.

Bound variables in expressions can be consistently renamed by new variables (that do not occur in the expression) without changing the intended meaning of the expression. That is, expressions that differ only by consistent renaming of bound variables are considered equivalent.

Renaming

For example, the following are equivalent pairs:

`(lambda (x) x)`

`(lambda (x1) x1)`

`(+ x ((lambda (x) (+ x y)) 4))`

`(+ x ((lambda (x1) (+ x1 y)) 4))`

An example of incorrect renaming:

```
(+ x ((lambda (y) (+ y y)) 4))  
(+ x1 ((lambda (x1) (+ x1 y)) 4))
```

Compare this operation of renaming with the Lexical Address transformation we defined in a previous lecture. If we transform an expression **E1** and its renamed version **rename(E1)** into lexical address notation and remove the names of the variables - we will obtain identical expressions.

Substitute is an operation which replaces free occurrences of variable references in an expression by other expressions.

Definition

A substitution s is a mapping from a finite set of variables to a finite set of expressions.

Substitutions are denoted using set notation.

For example:

$$\{ \begin{array}{l} x = 3, \\ y = a, \\ z = \#t, \\ w = (\text{lambda } (x) (+ x 1)) \end{array} \}$$

is a substitution.

```
{ x = 3,  
  x = a,  
  z = #t,  
  w = (lambda (x) (+ x 1)) }
```

is not a substitution because the variable x is mapped to 2 distinct values.

NOTE: in these expressions, we denote expressions in their unparsed form instead of the more verbose AST form - but remember that substitutions map variable names to expressions.

For example,

`{ x = 3, y = a, z = #t }`

really denotes:

`{ x = (num-exp 3),
 y = (lit-exp a),
 z = (bool-exp #t) }`

Composition (combination) of Substitutions

Definition

The composition of substitutions s and s' , denoted $s \circ s'$, is a substitution s'' that extends s with a binding $\langle x; s'(x) \rangle$ for every variable x for which $s(x)$ is not defined.

Composition (combination) of Substitutions

For example:

$$\{ x = 3, y = a \} \circ \{ z = \#t, w = (\text{lambda } (x) (+ x 1)) \}$$
$$\{ x = 3, y = a, z = \#t, w = (\text{lambda } (x) (+ x 1)) \}$$

Composition (combination) of Substitutions

The empty substitution $\{\}$ is the neutral element of the substitution-composition operation:

For every substitution s , $\{\} \circ s = s \circ \{\} = s$.

By definition, The substitute operation consists of applying a substitution s to an expression E . This operation is denoted $E \circ s$ (or just Es if no confusion arises), and involves replacing free variable occurrences in E by expressions.

Substitution is performed in two steps:

- Consistent renaming of the expression E and the expressions in s .
- Simultaneous replacement of all free occurrences of the variables of s in the renamed E by the corresponding renamed expressions of s .

Substitution Examples

- $10 \circ \{x = 5\} = 10$: No renaming; no replacement.
- $(+ \ x \ y) \circ \{x = 5\} = (+ \ 5 \ y)$: No renaming; just replacement.
- $(+ \ x \ y) \circ \{x = 5, y = 'x'\} = (+ \ 5 \ 'x)$: No renaming; just replacement.

Substitution Examples

• $((+ x ((\text{lambda } (x) (+ x 3)) 4)))$
o $\{x = 5\} =$

1. Renaming: E turns into

$((+ x ((\text{lambda } (x1) (+ x1 3)) 4)))$

2. Substitute: E turns into:

$((+ 5 ((\text{lambda } (x1) (+ x1 3)) 4)))$

Substitution Examples

- $(\lambda y. ((\lambda x. x) y) x)$
 - o $\{x = (\lambda x. (y x))\} =$

Variable y in the substitution is free. It should stay free after the substitution application.

1. Renaming: The substitution turns into

$\{x = (\lambda x_1. (y x_1))\},$

E turns into

$(\lambda y_2. (((\lambda x_3. x_3) y_2) x))$

2. Substitute: E turns into

$(\lambda y_2. (((\lambda x_3. x_3) y_2) (\lambda x_1. (y x_1))))$

Observe: What would be the result without renaming? Note the difference in the binding status of the variable y .

Applicative Eval Examples

Let us trace the evaluation of the applicative eval algorithm on the following *L2* program:

```
(L2
  (define square (lambda (x) (* x x)))
  (define sum-of-squares (lambda (x y)
                           (+ (square x) (square y))))
  (define f (lambda (a) (sum-of-squares (+ a 1) (* a 2))))
  (f 5)) ;; 136
```

We skip the evaluation of the three define expressions, which bind the variables to closures - and trace the evaluation of `(f 5)`:

Applicative Eval Examples

```
applicative-eval[ (f 5) ] ==>  
  applicative-eval[ f ] ==>  
    <Closure (a) (sum-of-squares (+ a 1) (* a 2) )>  
  applicative-eval[ 5 ] ==> 5
```

Applicative Eval Examples

```
applicative-eval[ (sum-of-squares (+ 5 1) (* 5 2)) ] ==>
  applicative-eval[sum-of-squares] ==>
    <Closure (x y) (+ (square x) (square y))>
  applicative-eval[ (+ 5 1) ] ==>
    applicative-eval[ + ] ==> <prim-op +>
    applicative-eval[ 5 ] ==> 5
    applicative-eval[ 1 ] ==> 1
  ==> 6
applicative-eval[ (* 5 2) ] ==>
  applicative-eval[ * ] ==> <prim-op *>
  applicative-eval[ 5 ] ==> 5
  applicative-eval[ 2 ] ==> 2
  ==> 10
```

Applicative Eval Examples

```
applicative-eval[ (+ (square 6) (square 10)) ] ==>
  applicative-eval[ + ] ==> <prim-op +>
  applicative-eval[ (square 6) ] ==>
    applicative-eval[ square ] ==>
      <Closure (x) (* x x)>
      applicative-eval[ 6 ] ==> 6
  ==>
  applicative-eval[ (* 6 6) ] ==>
    applicative-eval[ * ] ==> <prim-op *>
    applicative-eval[ 6 ] ==> 6
    applicative-eval[ 6 ] ==> 6
  ==> 36
```

Applicative Eval Examples

```
applicative-eval[ (+ (square 6) (square 10)) ] ==>
...
applicative-eval[ (square 10) ]
  applicative-eval[ square ] ==>
    <Closure (x) (* x x)>
  applicative-eval[ 10 ] ==> 10
==>
applicative-eval[ (* 10 10) ] ==>
  applicative-eval[ * ] ==> <prim-op *>
  applicative-eval[ 10 ] ==> 10
  applicative-eval[ 10 ] ==> 10
==> 100
==> 136
```

Example with Renaming

Let us trace the evaluation of the following program:

(L2

```
(define y 4)
(define f (lambda (g) (lambda (y) (+ y (g y)))))
(define h (lambda (x) (+ x y)))
(f h) ;; => <Closure (y1) (+ y1 ((lambda (x) (+ x y)) y1))>
```

Trace of the algorithm:

```
applicative-eval[ (f h) ] ==>  
  applicative-eval[ f ] ==>  
    <Closure (g) (lambda (y) (+ y (g y)))>  
  applicative-eval[ h ] ==>  
    <Closure (x) (+ x y)>
```

Example with Renaming

Substitute – rename both expressions and replace:
Map the closure value of *f* to the
corresponding lambda expression

```
(lambda (y2) (+ y2 (g y2)))  
o {g = (lambda (x1) (+ x1 y))}
```

Reduce –

```
applicative-eval[ (lambda (y2)  
                  (+ y2 ((lambda (x1)  
                          (+ x1 y)) y2 ) )) ]  
=> <Closure (y2) (+ y2 ((lambda (x1) (+ x1 y)) y2))>
```

Example with Renaming

Renaming plays here an essential role. Without it, the application `((f h) 3)` would replace all free occurrences of `y` by 3, yielding 9 as the result, instead of 10 - the correct value when the inner `y` variable reference remains bound to the global VarDecl of `y` to 4.

Parameter Passing Mode: By Value

The substitution model – applicative order uses the call-by-value method for parameter passing.

This is the standard evaluation model in Scheme, and the most frequent method in other languages as well (JavaScript, C++, Java).

Let us extend *L2* with support for compound values, leading to the definition of the *L3* language.

L3: Compound Values and Quoted Literal Expressions

In this language, the AST is extended with new primitives to support compound values (lists), we also introduce a special value for the empty list (' ()) which the parser must recognize and support for compound literal expressions: up to this point, the only literal expressions we supported were numbers, strings and booleans.

The expanded AST for *L3* is:

L3: Compound Values and Quoted Literal Expressions

```
<program> ::= (L3 <exp>+) // program(exps:List(exp))
<exp> ::= <define-exp> | <cexp>
<define-exp> ::= (define <var-decl> <cexp>) // def-exp(var:var-decl, val:cexp)
<cexp> ::= <num-exp> // num-exp(val:Number)
           | <bool-exp> // bool-exp(val:Boolean)
           | <prim-op> // prim-op(op:string)
           | <var-ref> // var-ref(var:string)
           | (if <exp> <exp> <exp>) // if-exp(test,then,else) ## L2
           | (lambda (<var-decl*>) <cexp>+) // proc-exp(params:List(var-decl), body:List(cexp)) ## L2
           | (quote <sexp>) // lit-exp(val:Sexp) ##### L3
           | (<cexp> <cexp>*) // app-exp(rator:cexp, rands:List(cexp))
<prim-op> ::= + | - | * | / | < | > | = | not | and | or | eq?
           | cons | car | cdr | pair? | list? | number? | boolean? | symbol?
           | display | newline ##### L3
<num-exp> ::= a number token
<bool-exp> ::= #t | #f
<var-ref> ::= an identifier token
<var-decl> ::= an identifier token
<sexp> ::= a symbol token | ( <sexp>* ) | ( <sexp> . <sexp> ) ##### L3
```

L3: Compound Values and Quoted Literal Expressions

The main additions in L3 is the fact that the set of computed values now includes complex composite values (lists). The set of computed values is now:

```
Value = Number | Boolean | Prim-op | Closure  
       | Void      | SExp  
SExp = Symbol      | Number | Boolean  
       | EmptySExp | Pair(SExp, SExp)
```

L3: Compound Values and Quoted Literal Expressions

To support these composite datatypes, we introduce value constructors and accessors as primitives (**cons**, **car**, **cdr**) and the corresponding type predicates as primitives as well (**pair?**, **list?**, **symbol?**) and equality predicate (**eq?**) which must be capable to recognize the empty-list value. We also introduce side effect primitives (**display** and **newline**).

The empty list special value (which is a value which is not a number, not a boolean and not a symbol) must be supported in the syntax.

L3: Compound Values and Quoted Literal Expressions

The last modification we introduce is to support literal expressions for the new compound values. We use the Scheme quote special operator to support these. An expression:

`(quote <sexp>)`

is a special expression which is computed according to the following computation rule:

`eval((quote <sexp>)) => <sexp>`

L3: Compound Values and Quoted Literal Expressions

For example:

```
(quote a)      ;; => 'a  
(quote (a b)) ;; => '(a b)
```

The special form (**quote** <sexp>) is written in Scheme in a shorthand notation '<sexp>' - for example, 'a for a symbol or '(a b) for a list.

To support the apply-procedure and substitution of values back as expressions, we took care in the procedure to turn values into expressions with a special case for Literal Expressions that wrap SExp values:

L3: Compound Values and Quoted Literal Expressions

```
const valueToLitExp =  
  (v: Value): NumExp | BoolExp | StrExp | LitExp |  
    PrimOp | ProcExp =>  
    isNumber(v) ? makeNumExp(v) :  
    isBoolean(v) ? makeBoolExp(v) :  
    isString(v) ? makeStrExp(v) :  
    isPrimOp(v) ? v :  
    isClosure(v) ? makeProcExp(v.params, v.body) :  
    makeLitExp(v);
```

Since SExps are not a type that exists in TypeScript, we must implement this type as part of the possible values computed by L3. This is implemented in the definition of the L3-value module:

L3: Compound Values and Quoted Literal Expressions

```
interface CompoundSExp {  
    tag: "CompoundSExp";  
    val1: SExpValue;  
    val2: SExpValue;  
}
```

```
interface EmptySExp {  
    tag: "EmptySExp";  
}
```

```
interface SymbolSExp {  
    tag: "SymbolSExp";  
    val: string;  
}
```

L3: Compound Values and Quoted Literal Expressions

```
type SExpValue = number | boolean | string | PrimOp  
               | Closure | SymbolSExp | EmptySExp  
               | CompoundSExp;
```

L3: Compound Values and Quoted Literal Expressions

```
const isSExp = (x: any): x is SExpValue =>  
  typeof(x) === 'string' ||  
  typeof(x) === 'boolean' ||  
  typeof(x) === 'number' ||  
  isSymbolSExp(x) ||  
  isCompoundSExp(x) ||  
  isEmptySExp(x) ||  
  isPrimOp(x) ||  
  isClosure(x);
```

L3: Compound Values and Quoted Literal Expressions

```
const makeCompoundSExp =  
  (val1: SExpValue, val2: SExpValue): CompoundSExp =>  
    ({tag: "CompoundSexp", val1: val1, val2 : val2});  
  
const isCompoundSExp =  
  (x: any): x is CompoundSExp =>  
    x.tag === "CompoundSexp";
```

L3: Compound Values and Quoted Literal Expressions

```
const makeEmptySExp = (): EmptySExp =>  
  ({tag: "EmptySExp"});
```

```
const isEmptySExp = (x: any): x is EmptySExp =>  
  x.tag === "EmptySExp";
```

L3: Compound Values and Quoted Literal Expressions

```
const makeSymbolExp = (val: string): SymbolExp =>  
  ({tag: "SymbolExp", val: val});
```

```
const isSymbolExp = (x: any): x is SymbolExp =>  
  x.tag === "SymbolExp";
```

The **SExp** datatype must also be supported in the parser for *L3* - so that we expand the *L3*-AST module with support for parsing SExp values and returning quoted **SExp** values accordingly.

This is supported in the following function:

L3: Compound Values and Quoted Literal Expressions

```
// param is the parameter given to quote  
const parseLitExp = (param: Sexp): Result<LitExp> =>  
    bind(parseSExp(param),  
        (sexp: SExpValue) => makeOk(makeLitExp(sexp)));
```

L3: Compound Values and Quoted Literal Expressions

```
const isDottedPair = (sexps: Sexp[]): boolean =>  
  sexps.length === 3 &&  
  sexps[1] === "."
```

```
const makeDottedPair = (sexps : Sexp[]): Result<SExpValue> =>  
  safe2((val1: SExpValue, val2: SExpValue) =>  
    makeOk(makeCompoundSExp(val1, val2)))  
    (parseSExp(sexps[0]), parseSExp(sexps[2]));
```

L3: Compound Values and Quoted Literal Expressions

```
// sexp is the output of p (sexp parser)
export const parseSExp = (sexp: SExp): Result<SExpValue> =>
  sexp === "#t" ? makeOk(true) :
  sexp === "#f" ? makeOk(false) :
  isString(sexp) && isNumericString(sexp) ? makeOk(+sexp) :
  isSexpString(sexp) ? makeOk(sexp.toString()) :
  isString(sexp) ? makeOk(makeSymbolSExp(sexp)) :
  sexp.length === 0 ? makeOk(makeEmptySExp()) :
  isDottedPair(sexp) ? makeDottedPair(sexp) :
  isArray(sexp) ? (
    // fail on (x . y z)
    sexp[0] === '.' ? makeFailure("Bad dotted sexp: " + sexp) :
    safe2((val1: SExpValue, val2: SExpValue) =>
      makeOk(makeCompoundSExp(val1, val2)))
    (parseSExp(first(sexp)), parseSExp(rest(sexp)))) :
  makeFailure(`Bad literal expression: ${sexp}`);
```

L3: Compound Values and Quoted Literal Expressions

With this interpreter, we can write programs such as the following - many examples are shown in the tests:

```
(L3 (define empty? (lambda (x) (eq? x '())))  
  (define filter  
    (lambda (pred l)  
      (if (empty? l)  
          l  
          (if (pred (car l))  
              (cons (car l) (filter pred (cdr l)))  
              (filter pred (cdr l))))))  
  (filter (lambda (x) (not (= x 2))) '(1 2 3 2)))
```

Both the parser and the interpreter of *L3* handle possible errors - either in the syntax of the program to be parsed or in the semantics of the program, that is, the interpreter must be capable of detecting an error at runtime and to report this error properly.

We implement error handler in the parser using the Result monad approach:

```
const parseL3 = (x: string): Result<Program>
```

That is, the parser either returns a valid **Program** result wrapped in an **Ok** value or a **Failure**. Similarly, we define the interpreter with the following signature:

```
const L3applicativeEval =  
  (exp: CExp, env: Env): Result<Value>
```

That is, the interpreter can obtain as parameter a valid **CExp**, and it returns either an **Ok Value** or a **Failure**. We must make sure the interpreter is never passed a **Failure** value in one of the recursive calls.

This is ensured by consistently adopting the pattern of

```
bind(<call returning a Result<Value>>,  
     (value: Value) => continuation)
```

This means that we do **not** change the definition of the AST data type or the Value data type to include **Error** as a possible values. There are two reasons for this decision:

- Errors are not *real* ASTs or Values - they are semantically outside the domains of the syntax and the interpreter.
- Errors are **absorbing** elements of the interpreter.

Absorbing means that if we *compose* an **Error** with any other value when interpreting a compound expression, then the whole value must become an **Error** - regardless of the rules of evaluation of the expression (whether it is a special form or an application).

For example, consider the rule of evaluation of an `IfExp` expression: it is a *special* form - that is, not all sub-expressions are evaluated before the value of the compound **IfExp** is computed. The semantic definition is:

```
To eval: IfExp(test, then, alt) in env:  
    if isTrueValue(eval(test, env)) then  
        eval(then, env)  
    else  
        eval(alt, env)
```

With error processing, the definition is implemented using the **bind** pattern:

```
const evalIf = (exp: IfExp, env: Env): Result<Value> =>  
  bind(L3applicativeEval(exp.test, env),  
    (test: Value) =>  
      isTrueValue(test) ?  
        L3applicativeEval(exp.then, env) :  
        L3applicativeEval(exp.alt, env));
```

We first test whether the evaluation of the `exp.test` sub-expression returns an error. If it is the case, the whole expression is reduced to this error - without evaluating any of the sub-expressions `exp.then` or `exp.alt`.

Similarly, when evaluating an application expression (**AppExp**) - the semantic rule is:

```
To eval: AppExp(rator, rands) in env:  
  let proc = eval(rator, env)  
      args = map(eval, rands)  
      applyProc(proc, args)
```

With error processing, the processing is expanded into:

```
isAppExp(exp) ? safe2((rator: Value, rands: Value[]) =>
  L3applyProcedure(rator, rands, env))
  (L3applicativeEval(exp.rator, env),
   mapResult(rand => L3applicativeEval(rand, env),
              exp.rands)) :
```

We adopt two distinct patterns (in addition to the general **bind** pattern) to handle possible errors:

First, the `safe2` higher-order procedure receives a procedure returning `Value` (not allowing Failures) and describes what to do with them in the happy path when no errors are met. It returns a safe version of the procedure which takes as arguments `Result<Value>` and guards the procedure with error checking.

If one of the arguments is a **Failure**, the safe procedure will return this **Failure**, else the **Ok** values are passed to the happy path procedure.

Second, the `mapResult` higher-order procedure applies a procedure `f` of type `[T1 => Result<T2>]` to an array `T1[]` and executes `f` on each of the elements of the array. If one of them returns a **Failure**, this failure is returned, else an **Ok** wrapper of the array of results is returned.

That is, `mapResult` is a safe version of `map` for functions that can return a `Failure`. Note that it does not return an array `Result<T2>[]` but instead a value of type `Result<T2[]>`.

Error Handling

```
const bind =  
  <T, U>(r: Result<T>, f: (x: T) => Result<U>): Result<U> =>  
    isOk(r) ? f(r.value) : r;  
  
const mapResult =  
  <T, U>(f: (x: T) => Result<U>, list: T[]): Result<U[]> =>  
    isEmpty(list) ? makeOk([]) :  
    bind(f(first(list)),  
      (fa: U) => bind(mapResult(f, rest(list)),  
        (fas: U[]) => makeOk(cons(fa, fas))));  
  
const safe2 = <T1, T2, T3>(f: (x: T1, y: T2) => Result<T3>):  
  (xr: Result<T1>, yr: Result<T2>) => Result<T3> =>  
    (xr: Result<T1>, yr: Result<T2>) =>  
      bind(xr, (x: T1) => bind(yr, (y: T2) => f(x, y)));
```

Note that the net result of guarding against errors at all stages of the parsing and evaluation process allows us to handle errors in a proper manner **without using exception** in the meta-language.

We do not throw exceptions. Instead, we return Result values, and properly guard against composing Error values against other types of values along the whole evaluation process.

This is an important methodological point: we can implement a process of evaluation which is in essence equivalent to throwing exceptions **without requiring exceptions** in the meta-language.

In a sense, we have *explained* what throwing an exception would mean in the semantic domain by enforcing the fact that **Failure** is an absorbing element for all semantic operations that operate over **Result<Value>**.

`applicative-eval` implements an **eager approach** in evaluation: arguments are evaluated immediately, before the closure is reduced.

An alternative algorithm implements a **lazy approach** in evaluation: it avoids evaluating arguments until the last moment it is necessary. When is a value necessary in the evaluation process?

- When we need to decide a computation branch.
- When we need to apply a primitive procedure.

Normal Order Evaluation Algorithm

The **normal evaluation algorithm** is similar to the applicative-eval we have just reviewed. The only difference, which leads to the lazy approach, moves the step of argument evaluation just before a primitive procedure is applied. Otherwise, the algorithm is unchanged, and the computation rules for the special operators are the same.

This is a remarkably small change in the algorithm with a deep change to the way the language behaves.

To describe the Normal Order evaluation algorithm, we only need to change a single evaluation rule - that for application expressions:

The applicative-eval form we first saw is:

- `appEval(AppExp(rator, rands)) =>`
 `;; rator is of type CExp`
 `;; rands is of type CExp[]`
 `let proc = appEval(rator, env)`
 `;; First evaluate parameters`
 `args = map(r => appEval(r, env), rands)`
 `;; Then invoke the procedure`
 `;; on the values`
 `return apply-proc(proc, args)`

To obtain the normal order lazy strategy, we change the evaluation rule to the following:

```
· normalEval(AppExp(rator, rands))  
=>  
;; rator is of type CExp  
;; rands is of type CExp[]  
let proc = normalEval(rator, env)  
    ;; Invoke the procedure on the  
    ;; arguments *without* evaluating  
    ;; them  
return normal-apply-proc(proc, args)
```

The apply procedure process must be adapted
to this slight change:

Normal Order Evaluation Algorithm

```
apply-proc(proc: Closure, rands: CExp): Value
  if proc is a primitive:
    ;; We must evaluate all the args to
    ;; apply a primitive
    let args = map(normal-eval, rands)
    return apply-primitive(proc, args)
  else ;; proc is a closure
    ;; Substitute the rands and reduce
    let subst-body = substitute params in body
                      of proc with rands
    return normal-eval(subst-body)
```

Note that we do not need to turn the values back into expression before we apply the substitution in the body, since the **rand**s are passed as non-evaluated expressions.

Normal Evaluation Example

Let us consider the normal evaluation of the same example as above:

```
(define square (lambda (x) (* x x)))  
(define sum-of-squares (lambda (x y)  
                          (+ (square x) (square y))))  
(define f (lambda (a) (sum-of-squares (+ a 1) (* a 2))))  
(f 5)
```

Normal Evaluation Example

```
normal-eval[ (f 5) ] ==>  
  normal-eval[f] ==>  
    <Closure (a) (sum-of-squares (+ a 1) (* a 2))>  
  ==>  
normal-eval[ (sum-of-squares (+ 5 1) (* 5 2)) ] ==>  
  normal-eval[ sum-of-squares ] ==>  
    <Closure (x y) (+ (square x) (square y))>  
  ==> ...
```

Normal Evaluation Example

```
normal-eval[ (+ (square (+ 5 1)) (square (* 5 2))) ] ==>
  normal-eval[ + ] ==> <prim-op +>
  normal-eval[ (square (+ 5 1)) ] ==>
    normal-eval[ square ] ==> <Closure (x) (* x x)>
    ==>
    normal-eval[ (* (+ 5 1) (+ 5 1)) ] ==>
    normal-eval[ * ] ==> <prim-op *>
    normal-eval[ (+ 5 1) ] ==>
      normal-eval[ + ] ==> <prim-op +>
      normal-eval[ 5 ] ==> 5
      normal-eval[ 1 ] ==> 1
    ==> 6
    normal-eval[ (+ 5 1) ] ==>
      normal-eval[ + ] ==> <prim-op +>
      normal-eval[ 5 ] ==> 5
      normal-eval[ 1 ] ==> 1
    ==> 6
  ==> 36
```

Normal Evaluation Example

```
normal-eval[ (square (* 5 2)) ] ==>
  normal-eval[ square ] ==> <Closure (x) (* x x)>
  ==>
  normal-eval[ (* (* 5 2) (* 5 2)) ] ==>
  normal-eval[ * ] ==> <prim-op *>
  normal-eval[ (* 5 2) ] ==>
    normal-eval[ * ] ==> <prim-op *>
    normal-eval[ 5 ] ==> 5
    normal-eval[ 2 ] ==> 2
  ==> 10
  normal-eval[ (* 5 2) ] ==>
    normal-eval[ * ] ==> <prim-op *>
    normal-eval[ 5 ] ==> 5
    normal-eval[ 2 ] ==> 2
  ==> 10
==> 100
==> 136
```

Normal Evaluation Example

Observe how the same computations are **repeated** in the normal evaluation algorithm, while they were processed only once in applicative order: for example `(* 5 2)` when it is passed to the function **square** is not computed before the substitution into the body of **square** - which leads to the computation of `(* (* 5 2) (* 5 2))` in normal order instead of `(* 10 10)` in applicative order.

Normal Order Parameter Passing Mode: Call by Name

The normal order strategy of passing arguments to procedures without pre-computing them is called **call by name** - as opposed to the **call by value** defined by applicative-eval.

Normal-order evaluation is also called **lazy evaluation** because it delays the evaluation of arguments to the last moment when it is needed.

Normal order and applicative order are different algorithms applied to expressions in order to compute their value. Do they compute the same values?

The Church Rosser Theorem is a fundamental result in lambda calculus which states that: when applying reduction rules to terms in the lambda calculus, the ordering in which the reductions are chosen does not make a difference to the eventual result.

More precisely, if there are two distinct reductions or sequences of reductions that can be applied to the same term, then there exists a term that is reachable from both results, by applying (possibly empty) sequences of additional reductions.

In the context of the substitution model of the operational semantics of our language which is a variant of Lambda Calculus rewriting, the Church Rosser theorem leads to the following statement:

If both applicative-eval and normal-eval terminate (compute a value without an infinite loop and without exceptions), then they compute the same value.

More precisely, the differences between applicative-eval and normal-eval are:

Comparison Applicative Order vs. Normal Order Evaluation

- If both orders terminate (no infinite loop and no exception): They compute the same value.
- Normal order evaluation may repeat computations which applicative-eval does not.
- Whenever applicative order evaluation terminates, normal order terminates as well.

Comparison Applicative Order vs. Normal Order Evaluation

- There are expressions where normal order evaluation terminates, while applicative order does not.

Comparison Applicative Order vs. Normal Order Evaluation

Side effects (like printing) are executed in different ways by applicative-eval and normal-eval - this fact can be used to identify the evaluation order of an interpreter.

Comparison Applicative Order vs. Normal Order Evaluation

In applicative order, side-effects included in parameters will be executed only once before the reduction step; in normal order, these side-effects can be executed 0 to many times - depending on the logic of the execution, and in different orders than what is executed in applicative-order.

Consider the example:

```
(L3  
  (define loop (lambda (x) (loop x)))  
  (define g (lambda (x) 5))  
  (g (loop 0)))
```

In normal order, the application `(loop 0)` is not evaluated. In applicative order: the call `(g (loop 0))` enters into an infinite loop.

Consider the example:

```
(L3
  (define try
    (lambda (a b)
      (if (= a 0)
          1
          b))))
(try 0 (/ 1 0)))
```

In normal order, this program returns 1. In applicative order, it throws a **divide by 0** exception.

Consider, for example:

```
(L3  
  (define f (lambda (x)  
              (display x)  
              (newline)  
              (+ x 1)))  
  (define g (lambda (x) 5))  
  (g (f 0)))
```

With applicative-eval, this program prints 0 then returns 5. In contrast, in normal-eval, this program returns 5 without printing anything.