

# Principles of Programming Languages

## Operational Semantics

---

The operational semantics of a programming language is specified by a set of formal evaluation rules that operate on the AST of an expression. The evaluation process can be specified as an algorithm `eval(exp)` which maps an `AST` to a `Value`.

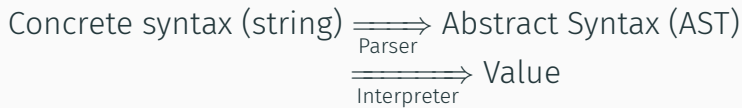
In this lecture, we go back to the definition of languages  $L1$  to  $L3$  (which are all subsets of Scheme) and present their operational semantics in a more formal manner, based on the definition of the AST of the languages and the method of structural induction.

For each language, we also specify the set of Values that can be computed by the language and review different implementation options for the domain of Values.

In the previous lectures, we studied the formal description of the **syntax** of programming languages. In subsequent lectures, we will study the formal description of their **semantics** - using the **operational semantics** approach.

The tool we use is to describe the implementation of interpreters for these languages using a functional subset of TypeScript as a meta-language.

The pipeline of operations we describe is:



# L1 Operational Semantics

---

L1 is a language in which primitive operators and primitive values can be combined recursively. In addition, composite expressions can be given a name and bound to variables using the **define** special form.

For example, the following is a program in  $L1$ :

```
(L1
  (define x (+ (* 2 3) (* 4 5)))
  (+ x (* 2 2)))
```

Let us summarize the syntax of *L1* using the BNF + Abstract Syntax specification we have developed in the previous lectures:

# L1 Syntax

```
<program> ::= (L1 <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)
          | (<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
```

Looking at the set of values, we decide to represent **Number** and **Boolean** using the corresponding value types in the meta-language (in our case, in TypeScript).

We must then decide how to represent primitive operators as computed values - so that we can decide which value to return when we compute the expression `+`.

In Scheme, when we compute this expression,  
we get:

```
> +
```

```
#<procedure:+>
```

That is, the value of the `+` expression (which is an expression of type `Symbol`) is a procedure in Scheme. If we implement primitive values as procedures, we rely on the fact that our meta-language (TypeScript) is a functional language which supports first-class procedures: a procedure value can be bound to a variable.

## Representing Primitive Operators

In JavaScript (and TypeScript), primitive operators are *not* variables bound to procedures. This can be verified by the following test:

```
const plus = (x, y) => x + y;  
console.log(plus);  
// => [Function: plus]  
console.log(+);  
// => SyntaxError: Unexpected token ')'
```

We observe that if we bind a function to a variable, then its value is returned as an object of type `[Function]`. But in contrast, the expression `+` (where `+` can be any primitive operator) is *not* a well formed expression in TypeScript.

## Representing Primitive Operators

We decide in our language  $L1$  to represent primitive operators as strings, and in the code of the interpreter, map each primitive operator to the underlying primitive operation in the meta-language. We will adopt this solution in our first model and in our implementation.

## Representing Primitive Operators

We represent primitive operators as a specific expression type in the AST

`PrimOp(op:PrimOpKeyword)`, where `PrimOpKeyword` is an enumerated type of all the defined primitive operators. The value of a `PrimOp` expression is itself when we evaluate the expression.

When we apply a primitive operator to arguments, we explicitly dispatch to each known primitive operator in the language and apply the corresponding primitive operation in the meta-language. The exhaustive enumeration of operators is terminated by a **never** case.

# Representing Primitive Operators

```
// There are type errors which we will address in L3
const applyPrimitive =
  (proc: PrimOp, args: Value[]): Result<Value> =>
    proc.op === "+" ? makeOk(reduce((x, y) => x + y, 0, args)) :
    proc.op === "-" ? makeOk(reduce((x, y) => x - y, 0, args)) :
    proc.op === "*" ? makeOk(reduce((x, y) => x * y, 1, args)) :
    proc.op === "/" ? makeOk(reduce((x, y) => x / y, 1, args)) :
    proc.op === ">" ? makeOk(args[0] > args[1]) :
    proc.op === "<" ? makeOk(args[0] < args[1]) :
    proc.op === "=" ? makeOk(args[0] === args[1]) :
    proc.op === "not" ? makeOk(!args[0]) :
    proc.op; // never
```

Note the fact that we adopt Scheme's model for the arithmetic operators: they are **variadic** - meaning that they can accept any number of arguments (from 0 and up). Since the AST of application forms (**AppExp**) supports any number of arguments, the syntax of *L1* also supports expressions of the form: `(+ 1 2)`, `(+ 1 2 3 4)` and even `(+ 1)` and `(+)`. In the meta-language, to apply a procedure to a list of arguments, we use **reduce**.

Think about what the value of `(+)` and `(*)` without parameters should be.

It turns out the handling of `-` and `/` is more complex than `+` and `*` because they are not associative operators. The code above does not reflect correctly the way `-` works in Scheme.

This is revised in the interpreter for *L3*.

Check in Scheme what is the value returned by `(-)` and `(/)`.

We have already pointed at the difference in behavior of primitives in Scheme and in JavaScript:

- Scheme has type-strict primitives. Invoking a primitive like `+` with non-number parameters throws an error.
- JavaScript has type-flexible primitives. Primitives do not fail in general when given unexpected data types as arguments.

In the implementation of the *L1* interpreter, we do not test the type of arguments at runtime. We also do not attempt to “do the right thing” when given unexpected arguments. Instead we silently compute “junk”.

In the implementation of *L3*, we attempt to add more type checking for primitives.

To determine which values can be computed by  $L1$  programs, we proceed inductively on the structure of  $L1$ -ASTs.

Atomic expressions can return as a value:

- A number (**NumExp**)
- A boolean (**BoolExp**)
- The value of a primitive operator (**PrimOp**)
- The value of a variable reference (**VarRef**) - which can be any value.

Define expressions return a `void` value.

Composite expressions return a value returned by the application of a primitive operator. This can be proved by induction - the value of a compound expression will always be either the value of a literal expression or the result of a primitive operator applied to values. We posit that primitive operators are delegated to the corresponding TypeScript primitives, which return either:

- A number (+, -, \*, /)
- A boolean (<, >, =, not)

Hence, composite expressions will return either a number, a boolean value or one of the literal values that atomic expressions can return.

Altogether, we conclude that the set of all possible values computed by *L1* programs is:

`Value = Number | Boolean | PrimOp | Void`

At this point, the type representing the values that can be computed by *L1* expressions is:

`Value = Number | Boolean | PrimOp`

We need to decide how to handle the case of `define` expressions.

In Scheme, a define expression does not return any value:

```
> (define x 1)  
>
```

This is unusual for a functional language - where all expressions are expected to return a value. The reason **define** does not return a usable value is that it is only used for its side-effect (which is to create a new binding for the variable with universal scope).

In non-functional languages, such expressions are called **statements** - as opposed to expressions which return a value.

To avoid the dichotomy expression/statement, we will adopt Scheme's model which is that expressions that have a side-effect return a special type of value - which is called **void**. The **void** type contains a single value (also called **void**).

In TypeScript, there are multiple *strange values* that can be used to encode the void value. We choose to use the **undefined** TypeScript value for this purpose.

We thus adopt the convention that the value of a **define** expression will be the **undefined** value.

Value = number | boolean | string | PrimOp | undefined

The operational semantics of  $L1$  is a function which maps inductively any Expression in  $L1$  to a Value:

$$eval : Expression \rightarrow Value$$

Let us define this algorithm in an inductive manner - by starting from the base cases - atomic expressions, and then moving up to composite expressions.

## Evaluation of Atomic Expressions

- `eval(NumExp(val)) => val`  
Number atomic literal expressions evaluate to number values.
- `eval(BoolExp(val)) => val`  
Boolean atomic literal expressions evaluate to boolean values `true` and `false`.

## Evaluation of Atomic Expressions

- `eval(PrimOp(op)) => PrimOp(op)`

Primitive procedures evaluate to the primitive operation.

- `eval(VarRef(var)) => applyEnv(env, var)`

Variables are evaluated by looking up their value in the global environment.

The evaluation of compound forms involves the recursive evaluation of parts of the compound forms, followed by a rule that determines how to combine the resulting values to obtain the value of the compound form.

For special forms, not all the parts of the compound form are always evaluated. The order in which the parts is evaluated is determined by the computation rule of the compound form type. In *L1* there is a single special form - **define**.

- `eval(DefineExp(var, val)) =>`  
 `;; var is of type VarDecl`  
 `;; val is type CExp`  
 `let val: Value = eval(val)`  
 add the binding `<VarDecl.var, val>`  
 to the global environment.  
 return undefined.

- `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)`

## Global Environment, Variable References and Define Expressions

In the specification of the `eval` algorithm above, we need to clarify two clauses:

- `eval(VarRef(var)) => applyEnv(env, var)`  
Variables are evaluated by looking up their value in the global environment.
- `eval(DefineExp(var, val)) =>`  
Add the binding to the global environment.

Both of these clauses rely on the object we called the **global environment**.

We model an **environment** as a **partial function** which maps variable references to values. A function is **partial** when it is defined on a restricted domain - in our case, not all variable references will be defined in a given environment, and the function is actually a **finite function** (defined on a finite set of values).

We model environments as an inductive data type, to reflect the fact that environments can be extended (this is what happens when we define a new variable and bind it to a value).

We adopt the method discussed in a previous lecture to model environments (which are mutable data structures) in a functional manner.

Recall that the strategy to implement mutable data structures in a functional manner consists of the following steps:

- Define a data type for the possible values of the data structure as a disjoint union - in particular, distinguish atomic value types and recursive value types.

- For each mutation operation, define a distinct value constructor for the data type which receives as a parameter the mutation parameters together with the original value of the object and returns a new value which represents the result of the mutation.

- Clients that perform mutation operations obtain a new value which represents the result of applying the mutation on the old version of the object.

## Global Environment, Variable References and Define Expressions

In the case of the environment data structure, we obtain the following inductive data type definition:

```
env ::= empty-env | extended-env
empty-env // empty-env()
extended-env(var, val, env)
// extended-env(var:string,
//               val:Value,
//               next-env:env)
```

That is, we define an environment as either:

- An empty environment
- Or an extended environment which maps variables (strings) to values on top of an existing environment.

On the basis of this inductive definition, we define a single value accessor for environment, which we call `applyEnv`:

## Global Environment, Variable References and Define Expressions

```
type Env = EmptyEnv | NonEmptyEnv;  
  
const isEnv = (x: any): x is Env =>  
  isEmptyEnv(x) || isNonEmptyEnv(x);
```

## Global Environment, Variable References and Define Expressions

```
interface EmptyEnv {  
  tag: "EmptyEnv"  
}
```

```
const makeEmptyEnv = (): EmptyEnv =>  
  ({tag: "EmptyEnv"});  
const isEmptyEnv = (x: any): x is EmptyEnv =>  
  x.tag === "EmptyEnv";
```

## Global Environment, Variable References and Define Expressions

```
export interface NonEmptyEnv {  
  tag: "Env";  
  var: string;  
  val: Value;  
  nextEnv: Env;  
};
```

```
const makeEnv =  
  (v: string, val: Value, env: Env): NonEmptyEnv =>  
    ({tag: "Env", var: v, val: val, nextEnv: env});  
const isEmptyEnv = (x: any): x is NonEmptyEnv =>  
  x.tag === "Env";
```

# Global Environment, Variable References and Define Expressions

```
const applyEnv =  
  (env: Env, v: string): Result<Value> =>  
    isEmptyEnv(env) ? makeFailure("var not found " + v)  
    env.var === v ? makeOk(env.val) :  
    applyEnv(env.nextEnv, v);
```

We lookup a variable `v` in an environment `env` recursively:

- No variable is defined in an empty environment - in this case, we return a `Failure`.

We lookup a variable `v` in an environment `env` recursively:

- Else, for an environment made up of the binding `var -> val` and a next environment `nextEnv`: If `var` is the same as `v`, return the corresponding `val`. Otherwise, continue searching in `nextEnv`.

As usual - this computation is an instance of the structural induction principle.

Here is an example of using this data structure:

## Global Environment, Variable References and Define Expressions

```
isFailure(applyEnv(makeEmptyEnv(), "x")); // => true
applyEnv(makeEnv("x", 1,
                makeEmptyEnv()), "x"); // => Ok(1)
isFailure(applyEnv(makeEnv("x", 1,
                makeEmptyEnv()), "y")); // => true
applyEnv(makeEnv("y", 2,
                makeEnv("x", 1,
                makeEmptyEnv()))), "x"); // => Ok(1)
applyEnv(makeEnv("x", 2,
                makeEnv("x", 1,
                makeEmptyEnv()))), "x"); // => Ok(2)
```

Given this definition of the **environment** data type, let us review how to handle variable references and **define** expressions.

We first observe that all mutation to the environment are necessary when evaluating a program (that is, a sequence of expressions that can include **define** expressions), while the evaluation of **CExp** expressions does not require any mutation. Yet, the evaluation of variable references (**VarRef** expressions) requires access to an environment.

We split the implementation of the eval algorithm in two cases:

- `L1applicativeEval(exp, env)`: evaluate a `CExp` AST with reference to a given environment.
- `evalL1program(program)`: evaluate a program.

`L1applicativeEval` handles the case of evaluating a variable reference with respect to a given environment. The evaluation rule for `VarRef` expressions is now clarified:

```
L1appEval(VarRef(var)) => applyEnv(env, var)  
;; Variables are evaluated by looking up  
;; their value in the global environment.
```

Let us now address the issue of evaluating a program - which is a sequence of expressions, which can be either **define** expressions, which update the current environment (and return **void**), or **CExp** expressions, which return a value.

`evalL1program(program)` receives a program, which includes an ordered sequence of expressions. It iterates over the expressions and depending on the type of each expression, it performs the following:

- DefineExp(var, val) =>  
 let value = L1applicativeEval(val, env)  
 if there are more expressions in the program:  
 let newEnv = extendEnv(var, val, env)  
 continue evaluating remaining  
 expressions in newEnv  
 else  
 return void

- CExp =>

```
let value = L1applicativeEval(cexp, env)
  if there are more expressions in the program:
    continue evaluating remaining
    expressions in env
  else
    return value
```

The function `L1applicativeEval` has the typical structure of a syntax-driven function with a conditional clause for each type of AST expression.

# Handling DefineExp and Evaluating Programs

```
const L1applicativeEval = (exp: CExp, env: Env): Result<Value> =>
  isNumExp(exp) ? makeOk(exp.val) :
  isBoolExp(exp) ? makeOk(exp.val) :
  isPrimOp(exp) ? makeOk(exp) :
  isVarRef(exp) ? applyEnv(env, exp.var) :
  isAppExp(exp) ? bind(
    mapResult((rand: CExp) =>
      L1applicativeEval(rand, env),
      exp.rands),
    (rands: Value[]) =>
      L1applyProcedure(exp.rator, rands)) :
  exp; // never
```

`evalL1program` invokes `evalSequence` to evaluate the sequence of expressions that appear inside the program with an initially empty environment. The expressions can either be of type **DefineExp** (which modify the current environment) or **CExp** (which have no side-effect).

To model the updates of the current environment in a functional manner, we implement in the `evalDefineExps` the process that creates a new environment after each `define`, and passes the new environment to the evaluation of the next expressions.

## Handling DefineExp and Evaluating Programs

```
const evalL1program = (program: Program): Result<Value> =>  
    evalSequence(program.exps, makeEmptyEnv());
```

## Handling DefineExp and Evaluating Programs

```
const evalSequence = (seq: Exp[], env: Env): Result<Value> =>
  isEmpty(seq) ? makeFailure("Empty sequence") :
  evalSequenceFirst(first(seq), rest(seq), env);
```

```
const evalSequenceFirst =
  (first: Exp, rest: Exp[], env: Env): Result<Value> =>
    isDefineExp(first) ? evalDefineExps(first, rest, env) :
    isEmpty(rest) ? L1applicativeEval(first, env) :
    bind(L1applicativeEval(first, env),
      _ => evalSequence(rest, env));
```

# Handling DefineExp and Evaluating Programs

```
const evalDefineExps =  
  (def: DefineExp, exps: Exp[], env: Env): Result<Value> =>  
    bind(L1applicativeEval(def.val, env),  
      (rhs: Value) => evalSequence(exps,  
                                    makeEnv(def.var.var, rhs, env)))
```

Observe that we implemented a form of mutation in *L1* (the evaluation of **define**) **without mutation** in the interpreter. This is obtained by using a functional implementation of the environment and *threading* the updated value of this environment (which is a new constructed value, obtained without mutation) at each step of the evaluation process.

The last aspect of the operational semantics that is left to clarify is how procedure calls are handled. In  $L1$ , the only procedures that can be applied are primitives, since we have not yet provided a way to define user procedures (we will do this next in  $L2$ ).

Consider for example the computation of this  
L1-expression:

$(+ (* 2 3) (- 3 2))$

The way this expression is evaluated according to the computation rule for **AppExp(rator, rands)** expressions is:

```
isAppExp(exp) ?  
  bind(mapResult((rand) => L1applicativeEval(rand, env),  
               exp.rands),  
        (rands) => L1applyProcedure(exp.rator, rands)) :
```

We first evaluate all the arguments, then we invoke the procedure on the computed values.

In the example above, it means that we compute the expression in this order:

- Compute  $( * \ 2 \ 3 )$  and  $( - \ 3 \ 2 )$
- Compute  $( + \ 6 \ 1 )$

Recursively, if we have an expression that is nested deeper, we start by computing the inner-most sub-expression, and then move up towards the root of the AST.

The operational semantics does **not** specify the order of execution among the arguments - we could compute  $( * \ 2 \ 3 )$  first and  $( - \ 3 \ 2 )$  next, or in reverse, or even together (in parallel).

The procedure `L1applyProcedure` does **not** need an `env` parameter because it receives only values, not expressions, and in particular, it does not receive any variable reference or any object that may contain a variable reference.