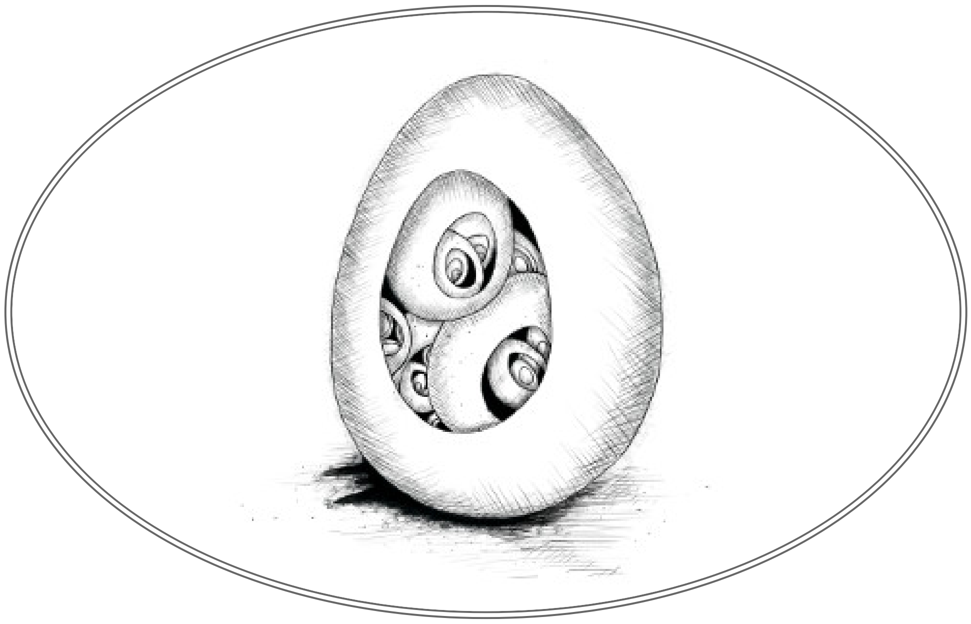
Project: A Programming Language

“The evaluator, which determines the meaning of expressions in a programming language, is just another program.”

— Hal Abelson and Gerald Sussman, *Structure and Interpretation of Computer Programs*



Building your own programming language is surprisingly easy (as long as you do not aim too high) and very enlightening.

The main thing I want to show in this chapter is that there’s no magic involved in building a programming language. I’ve often felt that some human inventions were so immensely clever and complicated that I’d never be able to understand them. But with a little reading and experimenting, they often turn out to be quite mundane.

We will build a programming language called Egg. It will be a tiny, simple language—but one that is powerful enough to express any computation you can think of. It will allow simple abstraction based on functions.

Parsing

The most immediately visible part of a programming language is its *syntax*, or notation. A *parser* is a program that reads a piece of text and produces a data structure that reflects the structure of the program contained in that text. If the text does not form a valid program, the parser should point out the error.

Our language will have a simple and uniform syntax. Everything in Egg is an expression. An expression can be the name of a binding, a number, a string, or an *application*. Applications are used for function calls but also for constructs such as `if` or `while`.

To keep the parser simple, strings in Egg do not support anything like backslash escapes. A string is simply a sequence of characters that are not double quotes, wrapped in double quotes. A number is a sequence of digits. Binding names can consist of any character that is not whitespace and that does not have a special meaning in the syntax.

Applications are written the way they are in JavaScript, by putting parentheses after an expression and having any number of arguments between those parentheses, separated by commas.

```
do(define(x, 10),
   if(>(x, 5),
      print("large"),
      print("small")))
```

The uniformity of the Egg language means that things that are operators in JavaScript (such as `>`) are normal bindings in this language, applied just like other functions. Since the syntax has no concept of a block, we need a `do` construct to represent doing multiple things in sequence.

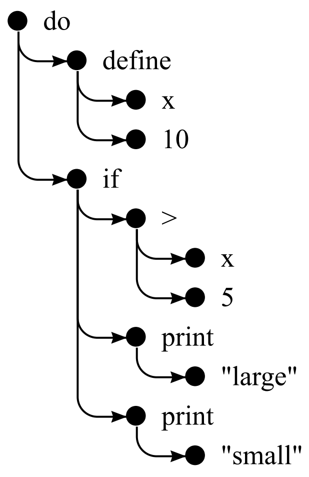
The data structure that the parser will use to describe a program consists of expression objects, each of which has a `type` property indicating the kind of expression it is and other properties to describe its content.

Expressions of type `"value"` represent literal strings or numbers. Their `value` property contains the string or number value that they represent. Expressions of type `"word"` are used for identifiers (names). Such objects have a `name` property that holds the identifier's name as a string. Finally, `"apply"` expressions represent applications. They have an `operator` property that refers to the expression that is being applied, as well as an `args` property that holds an array of argument expressions.

The `>(x, 5)` part of the previous program would be represented like this:

```
{
  type: "apply",
  operator: {type: "word", name: ">"},
  args: [
    {type: "word", name: "x"},
    {type: "value", value: 5}
  ]
}
```

Such a data structure is called a *syntax tree*. If you imagine the objects as dots and the links between them as lines between those dots, as shown in the following diagram, the structure has a treelike shape. The fact that expressions contain other expressions, which in turn might contain more expressions, is similar to the way tree branches split and split again.



Contrast this to the parser we wrote for the configuration file format in Chapter 9, which had a simple structure: it split the input into lines and handled those lines one at a time. There were only a few simple forms that a line was allowed to have.

Here we must find a different approach. Expressions are not separated into lines, and they have a recursive structure. Application expressions *contain* other expressions.

Fortunately, this problem can be solved very well by writing a parser function that is recursive in a way that reflects the recursive nature of the language.

We define a function `parseExpression` that takes a string as input. It returns an object containing the data structure for the expression at the start of the string, along with the part of the string left after parsing this expression. When parsing subexpressions (the argument to an application, for example), this function can be called again, yielding the argument expression as well as the text that remains. This text may in turn contain more arguments or may be the closing parenthesis that ends the list of arguments.

This is the first part of the parser:

```
function parseExpression(program) {
  program = skipSpace(program);
  let match, expr;
  if (match = /^"([^"]*)"/.exec(program)) {
    expr = {type: "value", value: match[1]};
  } else if (match = /^\d+\b/.exec(program)) {
    expr = {type: "value", value: Number(match[0])};
  } else if (match = /^[^\s(),#"]+/.exec(program)) {
    expr = {type: "word", name: match[0]};
  } else {
    throw new SyntaxError("Unexpected syntax: " + program);
  }

  return parseApply(expr, program.slice(match[0].length));
}

function skipSpace(string) {
  let first = string.search(/\S/);
  if (first == -1) return "";
  return string.slice(first);
}
```

Because Egg, like JavaScript, allows any amount of whitespace between its elements, we have to repeatedly cut the whitespace off the start of the program string. The `skipSpace` function helps with this.

After skipping any leading space, `parseExpression` uses three regular expressions to spot the three atomic elements that Egg supports: strings, numbers, and words. The parser constructs a different kind of data structure depending on which expression matches. If the input does not match one of these three forms, it is not a valid expression, and the parser throws an error. We use the `SyntaxError` constructor here. This is an exception class defined by the standard, like `Error`, but more specific.

We then cut off the part that was matched from the program string and pass that, along with the object for the expression, to `parseApply`, which checks whether the expression is an application. If so, it parses a parenthesized list of arguments.

```
function parseApply(expr, program) {
  program = skipSpace(program);
  if (program[0] != "(") {
    return {expr: expr, rest: program};
  }

  program = skipSpace(program.slice(1));
  expr = {type: "apply", operator: expr, args: []};
  while (program[0] != ")") {
    let arg = parseExpression(program);
    expr.args.push(arg.expr);
    program = skipSpace(arg.rest);
    if (program[0] == ",") {
      program = skipSpace(program.slice(1));
    } else if (program[0] != ")") {
      throw new SyntaxError("Expected ',' or ')'");
    }
  }
  return parseApply(expr, program.slice(1));
}
```

If the next character in the program is not an opening parenthesis, this is not an application, and `parseApply` returns the expression it was given. Otherwise, it skips the opening parenthesis and creates the syntax tree object for this application expression. It then recursively calls `parseExpression` to parse each argument until a closing parenthesis is found. The recursion is indirect, through `parseApply` and `parseExpression` calling each other.

Because an application expression can itself be applied (such as in `multiplier(2)(1)`), `parseApply` must, after it has parsed an application, call itself again to check whether another pair of parentheses follows.

This is all we need to parse Egg. We wrap it in a convenient `parse` function that verifies that it has reached the end of the input string after parsing the expression (an Egg program is a single expression), and that gives us the program's data structure.

```
function parse(program) {
  let {expr, rest} = parseExpression(program);
  if (skipSpace(rest).length > 0) {
    throw new SyntaxError("Unexpected text after program");
  }
  return expr;
}

console.log(parse("+(a, 10)"));
// → {type: "apply",
//    operator: {type: "word", name: "+"},
//    args: [{type: "word", name: "a"},
//           {type: "value", value: 10}]}
```

It works! It doesn't give us very helpful information when it fails and doesn't store the line and column on which each expression starts, which might be helpful when reporting errors later, but it's good enough for our purposes.

The evaluator

What can we do with the syntax tree for a program? Run it, of course! And that is what the evaluator does. You give it a syntax tree and a scope object that associates names with values, and it will evaluate the expression that the tree represents and return the value that this produces.

```
const specialForms = Object.create(null);

function evaluate(expr, scope) {
  if (expr.type == "value") {
    return expr.value;
  } else if (expr.type == "word") {
    if (expr.name in scope) {
      return scope[expr.name];
    } else {
      throw new ReferenceError(
        `Undefined binding: ${expr.name}`);
    }
  } else if (expr.type == "apply") {
    let {operator, args} = expr;
    if (operator.type == "word" &&
        operator.name in specialForms) {
      return specialForms[operator.name](expr.args, scope);
    } else {
      let op = evaluate(operator, scope);
      if (typeof op == "function") {
        return op(...args.map(arg => evaluate(arg, scope)));
      } else {
        throw new TypeError("Applying a non-function.");
      }
    }
  }
}
```

The evaluator has code for each of the expression types. A literal value expression produces its value. (For example, the expression `100` evaluates to the number 100.) For a binding, we must check whether it is actually defined in the scope and, if it is, fetch the binding's value.

Applications are more involved. If they are a special form, like `if`, we do not evaluate anything—we just and pass the argument expressions, along with the scope, to the function that handles this form. If it is a normal call, we evaluate the operator, verify that it is a function, and call it with the evaluated arguments.

We use plain JavaScript function values to represent Egg's function values. We will come back to this later, when the special form `fun` is defined.

The recursive structure of `evaluate` resembles the structure of the parser, and both mirror the structure of the language itself. It would also be possible to combine the parser and the evaluator into one function and evaluate during parsing, but splitting them up this way makes the program clearer and more flexible.

This is really all that's needed to interpret Egg. It's that simple. But without defining a few special forms and adding some useful values to the environment, you can't do much with this language yet.

Special forms

The `specialForms` object is used to define special syntax in Egg. It associates words with functions that evaluate such forms. It is currently empty. Let's add `if`.

```
specialForms.if = (args, scope) => {
  if (args.length != 3) {
    throw new SyntaxError("Wrong number of args to if");
  } else if (evaluate(args[0], scope) !== false) {
    return evaluate(args[1], scope);
  } else {
    return evaluate(args[2], scope);
  }
};
```

Egg's `if` construct expects exactly three arguments. It will evaluate the first, and if the result isn't the value `false`, it will evaluate the second. Otherwise, the third gets evaluated. This `if` form is more similar to JavaScript's ternary `?:` operator than to JavaScript's `if`. It is an expression, not a statement, and it produces a value, namely, the result of the second or third argument.

Egg also differs from JavaScript in how it handles the condition value to `if`. It will treat only the value `false` as false, not things like zero or the empty string.

The reason we need to represent `if` as a special form rather than a regular function is that all arguments to functions are evaluated before the function is called, whereas `if` should evaluate only *either* its second or its third argument, depending on the value of the first.

The `while` form is similar.

```
specialForms.while = (args, scope) => {
  if (args.length != 2) {
    throw new SyntaxError("Wrong number of args to while");
  }
  while (evaluate(args[0], scope) !== false) {
    evaluate(args[1], scope);
  }

  // Since undefined does not exist in Egg, we return false,
  // for lack of a meaningful result.
  return false;
};
```

Another basic building block is `do`, which executes all its arguments from top to bottom. Its value is the value produced by the last argument.

```
  specialForms.do = (args, scope) => {
    let value = false;
    for (let arg of args) {
      value = evaluate(arg, scope);
    }
    return value;
  };
```

To be able to create bindings and give them new values, we also create a form called `define`. It expects a word as its first argument and an expression producing the value to assign to that word as its second argument. Since `define`, like everything, is an expression, it must return a value. We'll make it return the value that was assigned (just like JavaScript's `=` operator).

```
  specialForms.define = (args, scope) => {
    if (args.length != 2 || args[0].type != "word") {
      throw new SyntaxError("Incorrect use of define");
    }
    let value = evaluate(args[1], scope);
    scope[args[0].name] = value;
    return value;
  };
```

The environment

The scope accepted by `evaluate` is an object with properties whose names correspond to binding names and whose values correspond to the values those bindings are bound to. Let’s define an object to represent the global scope.

To be able to use the `if` construct we just defined, we must have access to Boolean values. Since there are only two Boolean values, we do not need special syntax for them. We simply bind two names to the values `true` and `false` and use them.

```
const topScope = Object.create(null);

topScope.true = true;
topScope.false = false;
```

We can now evaluate a simple expression that negates a Boolean value.

```
let prog = parse(`if(true, false, true)`);
console.log(evaluate(prog, topScope));
// → false
```

To supply basic arithmetic and comparison operators, we will also add some function values to the scope. In the interest of keeping the code short, we’ll use `Function` to synthesize a bunch of operator functions in a loop instead of defining them individually.

```
for (let op of ["+", "-", "*", "/", "==", "<", ">"]) {
  topScope[op] = Function("a, b", `return a ${op} b;`);
}
```

It is also useful to have a way to output values, so we’ll wrap `console.log` in a function and call it `print`.

```
topScope.print = value => {
  console.log(value);
  return value;
};
```

That gives us enough elementary tools to write simple programs. The following function provides a convenient way to parse a program and run it in a fresh scope:

```
function run(program) {
  return evaluate(parse(program), Object.create(topScope));
}
```

We'll use object prototype chains to represent nested scopes so that the program can add bindings to its local scope without changing the top-level scope.

```
run(`
do(define(total, 0),
   define(count, 1),
   while(<(count, 11),
         do(define(total, +(total, count)),
            define(count, +(count, 1)))),
   print(total))
`);
// → 55
```

This is the program we've seen several times before that computes the sum of the numbers 1 to 10, expressed in Egg. It is clearly uglier than the equivalent JavaScript program—but not bad for a language implemented in less than 150 lines of code.

Functions

A programming language without functions is a poor programming language indeed. Fortunately, it isn't hard to add a `fun` construct, which treats its last argument as the function's body and uses all arguments before that as the names of the function's parameters.

```
specialForms.fun = (args, scope) => {
  if (!args.length) {
    throw new SyntaxError("Functions need a body");
  }
  let body = args[args.length - 1];
  let params = args.slice(0, args.length - 1).map(expr => {
    if (expr.type != "word") {
      throw new SyntaxError("Parameter names must be words");
    }
    return expr.name;
  });
```

```
    return function(...args) {
      if (args.length != params.length) {
        throw new TypeError("Wrong number of arguments");
      }
      let localScope = Object.create(scope);
      for (let i = 0; i < args.length; i++) {
        localScope[params[i]] = args[i];
      }
      return evaluate(body, localScope);
    };
  };
```

Functions in Egg get their own local scope. The function produced by the `fun` form creates this local scope and adds the argument bindings to it. It then evaluates the function body in this scope and returns the result.

```
run(`
do(define(plusOne, fun(a, +(a, 1))),
   print(plusOne(10)))
`);
// → 11

run(`
do(define(pow, fun(base, exp,
     if(==(exp, 0),
        1,
        *(base, pow(base, -(exp, 1)))))),
   print(pow(2, 10)))
`);
// → 1024
```

Compilation

What we have built is an interpreter. During evaluation, it acts directly on the representation of the program produced by the parser.

Compilation is the process of adding another step between the parsing and the running of a program, which transforms the program into something that can be evaluated more efficiently by doing as much work as possible in advance. For example, in well-designed languages it is obvious, for each use of a binding, which binding is being referred to, without actually running the program. This can be used to avoid looking up the binding by name every time it is accessed, instead directly fetching it from some predetermined memory location.

Traditionally, compilation involves converting the program to machine code, the raw format that a computer's processor can execute. But any process that converts a program to a different representation can be thought of as compilation.

It would be possible to write an alternative evaluation strategy for Egg, one that first converts the program to a JavaScript program, uses `Function` to invoke the JavaScript compiler on it, and runs the result. When done right, this would make Egg run very fast while still being quite simple to implement.

If you are interested in this topic and willing to spend some time on it, I encourage you to try to implement such a compiler as an exercise.

Cheating

When we defined `if` and `while`, you probably noticed that they were more or less trivial wrappers around JavaScript's own `if` and `while`. Similarly, the values in Egg are just regular old JavaScript values. Bridging the gap to a more primitive system, such as the machine code the processor understands, takes more effort—but the way it works resembles what we are doing here.

Though the toy language in this chapter doesn't do anything that couldn't be done better in JavaScript, there *are* situations where writing small languages helps get real work done.

Such a language does not have to resemble a typical programming language. If JavaScript didn't come equipped with regular expressions, for example, you could write your own parser and evaluator for regular expressions.

Or imagine you are building a program that makes it possible to quickly create parsers by providing a logical description of the language they need to parse. You could define a specific notation for that, and a compiler that compiles it to a parser program.

```
expr = number | string | name | application

number = digit+

name = letter+

string = '"' (! '"')* '"'

application = expr '(' (expr (',' expr)*)? ')'
```

This is what is usually called a *domain-specific language*, a language tailored to express a narrow domain of knowledge. Such a language can be more expressive than a general-purpose language because it is designed to describe exactly the things that need to be described in its domain and nothing else.

EXERCISES

ARRAYS

Add support for arrays to Egg by adding the following three functions to the top scope: `array(...values)` to construct an array containing the argument values, `length(array)` to get an array's length, and `element(array, n)` to fetch the *n*th element from an array.

```
// Modify these definitions...

topScope.array = "...";

topScope.length = "...";

topScope.element = "...";

run(`
do(define(sum, fun(array,
     do(define(i, 0),
        define(sum, 0),
        while(<(i, length(array)),
          do(define(sum, +(sum, element(array, i))),
             define(i, +(i, 1)))),
        sum))),
   print(sum(array(1, 2, 3))))
`);
// → 6
```

▶ Display hints...

Closure

The way we have defined `fun` allows functions in Egg to reference the surrounding scope, allowing the function's body to use local values that were visible at the time the function was defined, just like JavaScript functions do.

The following program illustrates this: function `f` returns a function that adds its argument to `f`'s argument, meaning that it needs access to the local scope inside `f` to be able to use binding `a`.

```
run(`
do(define(f, fun(a, fun(b, +(a, b)))),
   print(f(4)(5)))
`);
// → 9
```

Go back to the definition of the `fun` form and explain which mechanism causes this to work.

▶ Display hints...

Comments

It would be nice if we could write comments in Egg. For example, whenever we find a hash sign (`#`), we could treat the rest of the line as a comment and ignore it, similar to `//` in JavaScript.

We do not have to make any big changes to the parser to support this. We can simply change `skipSpace` to skip comments as if they are whitespace so that all the points where `skipSpace` is called will now also skip comments. Make this change.

```
// This is the old skipSpace. Modify it...
function skipSpace(string) {
  let first = string.search(/\S/);
  if (first == -1) return "";
```

```
    return string.slice(first);
  }

  console.log(parse("# hello\nx"));
  // → {type: "word", name: "x"}

  console.log(parse("a # one\n   # two\n()"));
  // → {type: "apply",
  //    operator: {type: "word", name: "a"},
  //    args: []}
```

▶ Display hints...

Fixing scope

Currently, the only way to assign a binding a value is `define`. This construct acts as a way both to define new bindings and to give existing ones a new value.

This ambiguity causes a problem. When you try to give a nonlocal binding a new value, you will end up defining a local one with the same name instead. Some languages work like this by design, but I've always found it an awkward way to handle scope.

Add a special form `set`, similar to `define`, which gives a binding a new value, updating the binding in an outer scope if it doesn't already exist in the inner scope. If the binding is not defined at all, throw a `ReferenceError` (another standard error type).

The technique of representing scopes as simple objects, which has made things convenient so far, will get in your way a little at this point. You might want to use the `Object.getPrototypeOf` function, which returns the prototype of an object. Also remember that you can use `Object.hasOwn` to find out if a given object has a property.

```
specialForms.set = (args, scope) => {
  // Your code here.
};

run(`
do(define(x, 4),
    define(setx, fun(val, set(x, val))),
    setx(50),
    print(x))
`);
// → 50
run(`set(quux, true)`);
// → Some kind of ReferenceError
```

▶ Display hints...

◂ ● ▸ ?