Ground-Up Computer Science

Chapter 7

(Feb 27, 2022)

Yin Wang

7 Interpreter

A: Today we will write an interpreter.

B: I often heard of interpreters, for example *JavaScript interpreter*, but I don't really know what they are.

A: An *interpreter* is what executes your programs. All programs in this course are executed by a *JavaScript interpreter*.

B: This looks like the console. I enter an expression and it gives me its value.

A: The console contains an interpreter inside, so it is basically an interpreter.

B: Is JavaScript an interpreter?

A: No, JavaScript is a programming language. You need to distinguish a *language* and its *interpreter*. Those are two different things.

B: So JavaScript is a language as English is a language. Can you give me an example of an interpreter?

A: For example, *node.js* is an interpreter. Node.js is a *JavaScript interpreter*. This means that it can execute programs written in the JavaScript language.

B: I see. JavaScript is the language and node.js is its interpreter. Are there other interpreters of JavaScript?

A: There are many JavaScript interpreters. Anybody can write a JavaScript interpreter. Every web browser has a built-in JavaScript interpreter, for example Chrome, Safari, Firefox, Internet Explorer.

B: Why do we have so many interpreters for one language?

A: This is like there are many singers of the same song. Each one is different and nobody is perfect. Every interpreter author tried to improve what they didn't like or doesn't fit their needs, but everybody makes mistakes, sometimes new mistakes, so this goes on forever.

B: That's funny.

A: Even big companies like Microsoft make mistakes, over and over. If we learn from old mistakes, we won't make the same mistakes again.

B: I should learn from mistakes, either made by myself or by others. They are valuable information.

A: I'm glad that you realized this. In case you know what is a CPU, CPUs are interpreters too. They are interpreters implemented in electronic circuits. They execute machine code.

B: I'm surprised. Interpreters can also be hardware?

A: Yes. Computer hardware circuits are not that different from software. Both hardware and software can implement the same logic. I hope you can see their connections from this course.

B: This idea is fascinating! I also heard of GPUs. Are GPUs interpreters too?

A: GPUs are not that different from CPUs, so they are interpreters too.

B: *Interpreter* seems to be a very broad concept.

A: Actually we humans are interpreters too. We are interpreters of natural languages (English, Japanese etc). We *make sense* out of words and sentences, in a similar way as JavaScript interpreters.

B: It seems to go far beyond computers!

A: If you understand interpreters, you may apply the idea to many things, including the design of CPU, GPU, network protocols etc. It is fair to say that interpreters are the essence of computer science, because "interpret" is a synonym of "compute".

B: That makes a lot of sense.

A: If you know how to write interpreters, you have the ability to *implement* programming languages. You may then design your own programming languages. You can also readily understand languages designed by others.

B: I'm glad that I'm getting there so soon.

Datatype definitions

A: Let's get down to earth. In this lesson you will implement an interpreter for a very simple but powerful language. It can execute most of programs you wrote in Lesson 1 and 2.

B: Good. So we already have lots of test cases for it.

A: Indeed, I made some tests using Lesson 1 functions, for example the `compose` function. You may open the exercise document and use it as a reference.

Exercise Set 7

B: Thank you. I opened it.

A: Don't read too much into it yet. Let me explain.

B: Okay.

A: You have already implemented an interpreter. The calculator (`calc`) that you wrote in Lesson 5 is also an interpreter. It interprets a very simple language of arithmetic expressions, such as `1 + 2 * 3`.

B: Yes. We wrote it as `binop("+", 1, binop("*", 2 3))`.

A: Yes. I wrote it in JavaScript syntax `1 + 2 * 3`, but you should understand this as `binop("+", 1, binop("*", 2 3))` in our own language. We will often use JavaScript's syntax to explain things in this lesson because otherwise it will be very verbose.

B: I see. Are we going to extend this language in this lesson?

A: That's the plan. Using the same ideas, you will arrive at a more powerful language. Arithmetic expressions (expressed by `binop`) will be part of the new language, and the calculator will be part of the new interpreter.

B: New language, new interpreter. Languages and interpreters seem to grow together.

A: Yes. It is often a good idea to *grow* a language instead of starting from scratch. You are very likely to make mistakes if you throw everything away.

B: What's new in the new language?

A: We need to add three constructs–variables, functions and calls.

B: The three pillars of programming languages, as we learned from Lesson 1.

A: It's good that you remembered that. Whenever you are lost in programming languages, look for those three things and think in terms of them. You will often find your way. Now let's start with their datatype definitions.

B: Okay.

A: Previously for the calculator, you have defined the `binop` datatype. It enables you to construct arithmetic expressions. You can write `2 * 3` as `binop("*", 2, 3)`.

B: I also can write `1 + 2 * 3` as `binop("+", 1, binop("*", 2, 3))` because the `binop` type can be nested.

A: This time we will create datatypes for variables, functions and calls. They are very much like the `binop` datatype. They can also be nested inside each other. For example, you can nest a `binop` inside a function. Maybe you can figure out what this expression means?

```
fun("x", binop("*", variable("x"), variable("x")))
```

B: It seems equivalent to the JavaScript function `x => x * x`, but it is in our own language constructs. I guess `fun` is the constructor of functions?

A: Right. `fun` is the constructor of functions. We use `fun` because *function* is a JavaScript keyword so we can't use it.

B: This is good too. Functions are `fun`.

A: Can you see that there are variables nested inside the `binop` structure?

B: Yes, I see two variables inside `binop("*", variable("x"), variable("x"))`, which means `x * x`.

A: Notice that we can't write this in the calculator language because it doesn't have variables.

B: I can see how the language has grown.

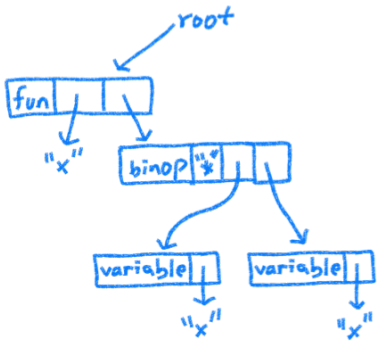
A: Once you have defined the datatypes `variable`, `fun` and `call`, you can construct programs like this. We will reuse the `binop` datatype definition from Lesson 5 so you may just copy it.

B: Since `binop` creates trees, after we have `variable`, `fun` and `call`, are we also creating trees?

A: Yes, programs are trees. We call them *abstract syntax trees* (*AST*). It may be helpful if you draw an AST for this example.

```
fun("x", binop("*", variable("x"), variable("x")))
```

B: I'll try.



A: Very good. Because trees are recursive data structures, you can construct programs of any size.

B: Indeed, trees are simple and powerful structures.

A: Now you may start writing datatype definitions for `variable`, `fun` and `call`. They are similar to `binop`.

B: Okay. I will write `variable`'s constructor first. It takes one parameter `name` which is a string.

```
function variable(name)
{
  return name;
}
```

A: That's not right. You can't just use the string as the variable.

B: Oh, I thought a variable is just a string.

A: If you just return the string, you won't be able to tell our variables from JavaScript strings.

B: It seems even if there is only one member, I still need to return a structure. How about this?

```
function variable(name)
{
  return pair("variable", pair(name, null));
}
```

A: Correct. The type tag will make it recognizable as a `variable`. Now you may finish the definitions of the type predicate and visitors of `variable`.

B: *(Write the rest of the datatype definition of* `variable`*, consult the exercise set, and send it to the teacher for a check.)*

A: Now write the datatype definition of `fun`. Do you still remember how many parts are in a function?

B: A function has two parts–parameters and function body.

A: Right, but in our language a function has only one parameter. This is to make things simple.

B: Just one parameter. Wouldn't that be limited?

A: Not really. You may still simulate multiple parameters by nested functions, such as `x => y => x + y`. For example, the `compose` function in Lesson 1 was written as `(f, g) => x => f(g(x))`. In our language, you may write it as `f => g => x => f(g(x))`. This is more complex, but you can still express it.

B: Got it. Whenever we have a multi-parameter function `(x, y, ...) => body`, we write it as `x => y => ... => body`

A: Yes, remember that you need to change the calls too. Where you normally write `f(2, 3)`, now you have to write `f(2)(3)`.

B: This style of calls often appear in Lesson 1's exercises. They puzzled me a lot, but now I'm used to it.

A: After you understand the interpreter, multi-parameter functions are fairly easy to add. Now you may write the datatype definition for `fun`. Its two members should be named "param" and "body".

B: *(Write the definition of* `fun` *(constructor, type predicate and visitors) and send it to the teacher.)*

A: Next is the definition for `call`. Do you remember how many parts are in a call?

B: Two. The operator and the operand.

A: Good. You may write the datatype definition of `call` now. Its two members can be named "op" and "arg".

B: *(Write the definition of* `call` *and send it to the teacher.)*

A: Let's do a small exercise. Can you translate the JavaScript function `x => y => x + y` into our language?

B: *(Write your answer and send it to the teacher.)*

A: Now translate the call `(x => y => x + y)(2)(3)` into our language. You may copy the previous answer because it is part of the call.

B: *(Write your answer and send it to the teacher.)*

Structure of the interpreter

A: You are almost done with datatype definitions. Before we get into more details, I hope you can have a working interpreter really soon. This can motivate you. Let's look at the general structure of the interpreter.

B: Okay.

A: One thing that makes the interpreter different from the calculator is that the interpreter supports multiple language constructs (`variable`, `fun`, `call`, `binop`), whereas the calculator supports only `binop`. For this reason we need multiple branches in the interpreter. Its structure looks like this.

```
function interp(exp)
{
  if (typeof(exp) == "number")
  {
    ...
  }
  else if (isVariable(exp))
  {
    ...
  }
  else if (isFunction(exp))
  {
    ...
  }
  else if (isCall(exp))
  {
    ...
  }
  else if (isBinOp(exp))
  {
    ...
  }
  else
  {
    throw "Illegal expression: " + pairToString(exp);
  }
}
```

It is not complete code so don't copy it as yet.

B: So we have a branch for each language construct. The last branch will report unrecognized expressions.

A: Yes, the structure is quite regular. We also have a branch for *literal values*. Literals are such as `2`, `3`, `"hello"`, `true`, `false`. They each represent one specific value in the language. For now it's okay to have just numbers. We may want to add other literals later.

B: I see the first branch is for number literals.

A: Like the calculator, the interpreter is a recursive function on trees. The parameter `exp` is the input program (expression). The interpreter will compute its value.

B: This input-output pattern of interpreter seems to be exactly the same as the calculator.

A: Actually this is not the whole picture, but it helps. First, think about this question. What do you return for the first branch, when `exp` is a number?

B: This is similar to the calculator's base case. For that case I just return the number itself.

```
if (typeof(exp) == "number")
{
  return exp;
}
```

A: Correct. Since you noticed that this is also the base case of the calculator, maybe it is helpful for you to implement the `binop` branch as the next step.

B: That seems to make a very smooth transition. I can then use the interpreter as a calculator even though it is not complete yet.

A: Remember that we are *growing* from the calculator.

B: Yes.

A: The `binop` branch should be almost the same as the `calc` function, except that you need to recursively call `interp` instead of `calc`.

B: *(Finish the `binop` branch so your interpreter is equivalent to the calculator. Test it with simple arithmetic expressions. Send the interpreter to the teacher for a check.)*

Variables

A: Very good. Now we can proceed to add other constructs. Let's look at the branch for variables. What should we do for this branch?

```
else if (isVariable(exp))
{
  ...
}
```

B: We should return the variable's value. The value must be stored somewhere, but I have no idea where.

A: Actually, I lied about the interpreter's parameters. There is one more parameter for the `interp` function. Its name is `env`, meaning ***env****ironment*. `env` is a lookup table, but it may also be a BST. It contains mappings from variable names to values.

Here is the new framework of `interp`. You may now add the `env` parameter to your `interp` function.

```
function interp(exp, env)
{
  if (typeof(exp) == "number")
  {
    ...
  }
  else if (isVariable(exp))
  {
    ...
  }
    ...
}
```

B: I see. I can just lookup `env` for the variable's value.

```
  else if (isVariable(exp))
  {
    return lookupTable(exp, env);
  }
```

A: You are almost there, but `env`'s keys are strings, not variables, so you need to get the variable's name first.

B: That's easy to fix.

```
  else if (isVariable(exp))
  {
    return lookupTable(variableName(exp), env);
  }
```

A: Good. There is one more problem. If the programmer makes a mistake, we may have *undefined variables* in the input program. For example we may have `(x => x * y)(3)`. In this case you need to report the error.

B: I will change it this way then.

```
  else if (isVariable(exp))
  {
    var value = lookupTable(variableName(exp), env);

    if (value == null)
    {
      throw "undefined variable: " + variableName(exp);
    }
    else
    {
      return value;
    }
  }
```

A: Good. Reporting the variable's name is a good idea. This will help the programmer locate the problem.

B: But I'm a little sad because I didn't think of the problem of undefined variables. Is there a systematic way of thinking so that I won't miss a case like this?

A: The wisdom is, always think about *all possibilities* of a variables or a function's return value. If there are possibilities which you haven't checked, the program will go wrong there.

B: That seems to be a good way of thinking. I'll keep that in mind.

A: There is one more thing to change here. We may want to change `env`'s data structure later (for example to a BST), so it is better not to hardcode `lookupTable` here. We may abstract this out using *abstract interfaces*.

B: I have used abstracted interfaces, but I'm not sure how to use it for this case.

A: Names are the essence of abstraction. You may just define three variables like the following. If we ever want to switch to BST, we change just three lines. No other code needs to be changed even if you used them a thousand times.

```
var emptyEnv = emptyTable;
var extEnv = addTable;
var lookupEnv = lookupTable;
```

B: The idea of abstraction is so profound. Now the code of the variable branch looks like this.

```
  else if (isVariable(exp))
  {
    var value = lookupEnv(variableName(exp), env);

    if (value == null)
    {
      throw "Undefined variable: " + variableName(exp);
    }
    else
    {
      return value;
    }
  }
```

A: Remember that when you wrote the `binop` branch, the `interp` function had only one parameter. Now we have two parameters, so you should add `env` to the recursive calls in the `binop` branch, otherwise they will go wrong.

B: I will do that.

(Send your extended interpreter code to the teacher for a quick check.)

Functions

A: The variable branch is all good now. You may start thinking about the function branch. What is the value of a function?

B: From Lesson 1, we learned that the value of a function is the function itself, plus some extra information.

A: What's the extra information?

B: When we have a call `(x => y => x + y)(2)`, Chrome's console gives us `y => x + y`, but there is extra information that "`x` is 2 inside `y => x + y`".

A: It's good that you remembered this, but we will first pretend that the value of a function is just the function itself. This can make the transition smoother. Just keep a note that we have something missing here.

B: Okay. I just return the function itself for now.

```
else if (isFunction(exp))
{
  return exp;  // something is missing here
}
```

A: Let's move on to the `call` branch. After that everything will be connected together.

B: Okay.

Calls

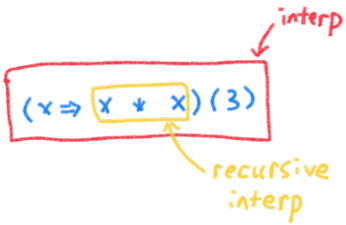
A: Do you remember how calls happen?

B: I remember *substitution*. In the function body, we replace every occurrence of the parameter with the operand. The value of the call is then the value of the substituted function body. I remember this example from Lesson 1.

A: Good, but substitution takes significant computing time and is complex to implement, so our interpreter will use a more practical strategy. Instead of substitution, we just recursively call the `interp` function on the function body.

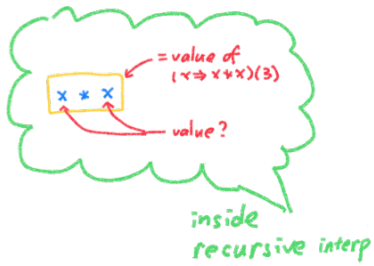
B: You mean, for this example `(x => x * x)(3)`, we recursively call `interp` on the function body `x * x`?

A: Right. If you draw a picture, it may look like this.



B: That's clear. If we call `interp` recursively on the function body, and we know the value of `x`, then we can compute the value of the call `(x => x * x)(3)`.

A: That's the idea. The question is how can we find the value of `x` while inside the recursive `interp`.



B: From the variable branch, I know we get the value of `x` from `env`.

A: But how did `env` contain the value of `x`? It didn't yet contain the value of `x` when `interp` sees `(x => x * x)(3)`.

B: Hmm, so we must put the key-value pair `x:3` into `env` somewhere, otherwise it just won't happen.

A: Actually, we extend `env` with the key-value pair `x:3` before the recursive call of `interp` on the function body `x * x`.

B: I see. First we do `addTable("x", 3, env)`. Oh, actually `extEnv("x", 3, env)` because we need abstraction.

A: Right. `extEnv("x", 3, env)` creates a new environment (call it `newEnv`). `newEnv` contains the value of `x`, so the recursive call `interp(funBody(exp), newEnv)` knows the value of `x`.

B: This is my current code.

```
  else if (isCall(exp))
  {
    var newEnv = extEnv(funParam(callOp(exp)), callArg(exp),
env);
    return interp(funBody(callOp(exp)), newEnv);
  }
```

A: This is not right yet. Can you really call `funBody` on the operator `callOp(exp)`? Is the operator a function?

B: Isn't the operator a function?

A: It may not be. For example `f(3)`, which is `call(variable("f"), 3)` in our language. The operator is `variable("f")`, which is a `variable`, not a `fun`!

B: This is interesting. We habitually say that `f` is a function, but actually `f` is just a variable!

A: Right. You are confused because of the imprecise way of describing things in natural languages and math. Now with programming languages, we have to be very accurate, otherwise things won't work.

B: Somebody said that programming is just another name for the lost art of thinking. Now I seems to understand it.

A: Very good. Actually for the call branch, you need to first evaluate both the operator and the operand. Otherwise you can't even ask if the operator is a function.

B: Got it. When `exp` is `f(3)`, we must first evaluate the operator `f` by a recursive call to `interp`. The recursive `interp` will lookup the value of the variable `f` in `env`. We need the same thing for the operand.

```
else if (isCall(exp))
{
  var op = interp(callOp(exp), env);
  var arg = interp(callArg(exp), env);
  var newEnv = extEnv(funParam(op), arg, env);
  return interp(funBody(op), newEnv);
}
```

A: You forgot to check whether `op` is a function. The programmer may make a mistake. He may have written `2(3)`. In this case you have to report the error "Calling non-function: 2".

B: Oh, old mistake again. I should always consider all possibilities.

```
else if (isCall(exp))
{
  var op = interp(callOp(exp), env);
  var arg = interp(callArg(exp), env);

  if (isFunction(op))
  {
    var newEnv = extEnv(funParam(op), arg, env);
    return interp(funBody(op), newEnv);
  }
  else
  {
    throw "Calling non-function: " + pairToString(op);
  }
}
```

A: This is much better. You are almost there, but there is still something wrong in the `call` and `fun` branch.

B: I guess we need to talk about the missing information in the function branch?

A: Yes. Let's talk about the example `(x => y => x + y)(2)` as you have seen in Lesson 1. Translate `(x => y => x + y)(2)(3)` into our language and run it with `interp`. Can you get the correct result `5`?

B: No, it complains that the variable `x` is undefined.

(Show that this happens to your teacher.)

A: The error message came from the variable branch. When `interp` sees the inner function's body `x + y`, is `x`'s value 2?

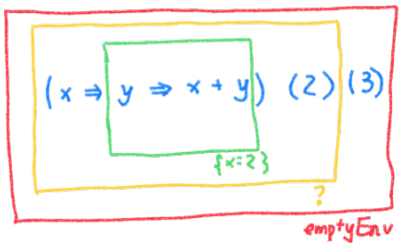
B: I think so, because when it sees `(x => y => x + y)(2)`, the call branch has put the key-value pair `x:2` into `env`, and it's called `newEnv`.

A: But `newEnv` is passed to the recursive call, so it is only visible inside the function body of `x => y => x + y`, which is `y => x + y`. When the recursive call returns, the original `env` doesn't contain `x`'s value.

B: You mean `env` doesn't contain x's value when we see `(x => y => x + y)(2)(3)`?

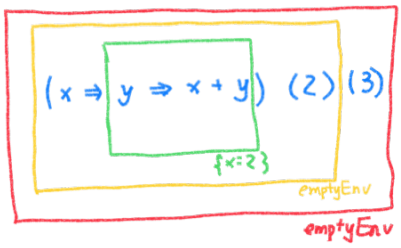
A: Right. You can draw a picture of the environments. Each nested expression may have a different environment.

B: I tried, but I have trouble about the yellow one. The green one is created when we evaluate `(x => y => x + y)(2)` and it contains `x:2`. I'm not sure about the yellow one.



A: The yellow environment is actually the same as the red one. Remember that we evaluate `(x => y => x + y)(2)` as the operator part of `(x => y => x + y)(2)(3)`. When we evaluate the operator and the operand, we use `env` without extend it.

B: I see. This is the new picture.



A: Any way, only the green environment contains `x:2`. When we evaluate the outermost call `(x => y => x + y)(2)(3)`, we can't see `x:2` because the red environment is empty.

B: What can we do about this? Can we do a substitution and create `y => 2 + y`?

A: No. As I said, we don't do substitutions in this interpreter.

B: Then I have no idea.

A: Here is a way. When `interp` sees a function, it can bundle the function and the current environment together, forming a so-called *closure*.

B: What does a closure look like?

A: For this example, the closure will contain the function `y => x + y` and the green environment.

B: I see. The green environment contains `x:2`. But I'm still not sure what to do.

A: You need to define a new datatype `closure`. It has two members "fun" and "env", so the visitors will be called `closureFun` and `closureEnv`.

B: *(Write the definition of `closure` datatype and show it to the teacher.)*

A: After you have the closure datatype, the function branch will just be like this:

```
else if (isFunction(exp))
{
  return closure(exp, env);
}
```

B: That's simple.

A: There is one more change you need for the call branch. For the call, we no longer get a function when evaluating the operator. We get a closure instead.

B: What do we do with the closure?

A: Do you remember why we have the closure?

B: To access the information in the green environment.

A: We use this environment so that we can know the variables' values when the function was created.

B: For example, `x` as in `y => x + y`?

A: Yes. We call `x` a *free variable* because it is not a parameter of this function. Its value came from outside of the function.

B: I see, *free variable*.

A: So in the recursive call of `interp` on the function body, we don't use the `env` parameter. We use the environment stored in the closure.

B: You mean something like this?

```
else if (isCall(exp))
{
  var op = interp(callOp(exp), env);
  var arg = interp(callArg(exp), env);

  if (isClosure(op))
  {
    var f = closureFun(op);
    var newEnv = extEnv(funParam(f), arg, closureEnv(op));
    return interp(funBody(f), newEnv);
  }
  else
  {
    throw "Calling non-function";
```

```
    }
  }
```

A: Correct.

B: That feels strange. Won't I miss any useful information in the current `env`?

A: You won't miss anything because the function is not created here. It is created where we could see the green environment.

B: I see. So it makes sense to bundle the environment at the moment where the function is created.

A: Yes. This will solve all our problems.

B: Nice.

A: This is all I will teach you about interpreters. There are some extensions to the interpreter which I left as exercises. They will deepen your understanding of interpreters. If you have questions, just let me know. (Exercise Set 7)

B: Thank you!

(Exercise omitted for the sample.)