Compilers and interpreters

Raffaello Giulietti

DTI SUPSI

www.dti.supsi.ch

I assume you have enough computer science maturity so I can appeal to your intuition when presenting new concepts.

I will thus introduce ideas quite informally and then refine them to add almost all the rigor and the precision required for an implementation.

The exercises are dense and essential. Everybody should try them to get a firm understanding before proceeding further.

I'll use Java as the *host* language for the implementation of the *guest* language Funny.

Funny is a closure-based, dynamically, strongly typed, eager language.

Introductory exercise

What does this mean? Do some quick research, just to get an idea.

Getting the feel

What do you think this snippet will print?

```
s = {(x) -> x * x};
println(s(1 + 1.5));
```

Yes, it prints 6.25, as does the next one:

```
println({(x) -> x * x}(1 + 1.5));
```

And this one?

```
s = {(x) -> x * x};
println(s(s(5 + 2 * 3)));
```

Right, it prints 14641, as does this one:

```
println({(x) -> x * x}({(x) -> x * x}(5 + 2 * 3)));
```

The *expression* {(x) -> x * x} stands for the *function* that takes a *parameter* x, has no *locals* (local variables) and has *code* x * x, another expression.

Each time execution reaches an expression, the interpreter *evaluates* it, *reducing* it to a *value*. For a function, this means building a *closure*, a combination of the *current environment* with the function itself, e.g., the expression `{(x) -> x * x}`.

An environment describes the *bindings* from *ids* (identifiers) to their currently assigned values. Over time, assignments replace the current value in a binding with a new value.

A closure can later be *invoked*, e.g., as in `s(1 + 1.5)`. The expression before the parentheses is evaluated. In this case, this means simply *getting* the value of `s` in the current environment, obtaining the closure assigned to `s`. Further, the expressions in parentheses are evaluated, delivering the *arguments* of the invocation, `2.5` in this case. These are *applied* to the closure.

Application of the arguments proceeds by pushing the current environment on the *execution stack* for later retrieval. Then the *closure's environment* (that is, the one that was current at closure's build-time) is designated as the *enclosing environment*. It is paired with a *new frame*, establishing a new extended environment which is set as current.

The new frame contains a *fresh* mutable binding for each parameter and each local of the closure's function. Parameters are initialized with the arguments and locals are initialized with nil.

When an environment is searched for an id, the frame is searched first. On failure, search recursively proceeds in the enclosing environment.

Hence, an environment is a pair consisting of a frame and the enclosing environment. The execution of a program starts with an initial empty environment.

Once this preparatory phase completes, execution of the invocation proceeds by retrieving the closure's function and evaluating the function's code, an expression. This, in turn, can entail other invocations, as known from virtually every programming language.

When evaluation of the function's code terminates, the resulting value is returned as the result of the original invocation. Before continuing at the invocation site, however, the former current environment pushed on the execution stack at invocation is popped from there and set as current.

Exercise 1

Consider

```
s = {(x) -> x + y};
y = 1.2;
println(s(0.1)); // this prints 1.3
y = 2.3;
println(s(0.1)); // but this prints 2.4
```

Although the invocations of `s` are the same, the returned values are different. Indeed, the function refers to `y`, an id external to it.

Give a detailed explanation of what happens here. Assume that the entire snippet executes in an environment that contains a binding for `y`. (In Funny, `println` is a *reserved word*, not an id.)

Exercise 2

A closure is a value. Like any other value, it can be returned as the result of an invocation.

Study this:

```
binaryAdder = {(x) -> {(y) -> x + y}};
println(binaryAdder(123)(234)); // prints 357

fiveAdder = binaryAdder(5);
sixAdder = binaryAdder(6);
println(fiveAdder(8) * sixAdder(3)); // prints 117
```

Here, fiveAdder refers to a closure whose environment maintains a binding for x *even after the invocation* binaryAdder(5) *completes*. How does this happen in detail? Explain.

Similarly for sixAdder, but with a *different* binding for x.

Exercise 3

Exercise 2 shows how a binding can survive the invocation that created it. Since a binding is mutable, it can maintain private state. This example shows how closures can act as objects:

```
accountMaker = {(balance) -> {(amount) -> balance += amount}};
poorAccount = accountMaker(1000);
richAccount = accountMaker(1000000);

poorAccount(50);
richAccount(100000);
poorAccount(-70);
richAccount(130000);

println(poorAccount(0)); // this prints 980
println(richAccount(0)); // this prints 1230000
```

Give the details of how this works.

Compiling functions and invocations

To parse functions the compiler maintains a *current scope*.

Whenever a function is compiled, the ids for the parameters and the locals are collected in two lists. The compiler checks that all these ids appear only once. The current scope becomes the *enclosing scope*. It is paired with the ids and the pair becomes the new current scope. The ids can *hide* equally named ids in the enclosing scope.

For each occurrence of an id in the function's code, the compiler further checks whether the id is *in scope*, i.e., whether it appears, directly or indirectly, in the current scope. If not, it signals an error.

Upon termination of the parsing of the function, the current scope is restored to the enclosing scope. This imposes the stack discipline for scopes known from many programming languages.

Look at this example:

```
{x, y -> x = {(x) z -> z = x + y}; z = 0}
```

The compiler starts with some current scope *s0*. It parses the outer function, extending *s0* to the new current scope *s1* = ⟨{x, y}, *s0*⟩, a pair with the new ids in the list and *s0* as the *enclosing scope*.

While still parsing the outer function, the compiler begins to parse the inner function. It extends the current scope *s1* to *s2* = ⟨{x, z}, *s1*⟩, which becomes current. The occurrence of the id y in the inner function is found indirectly in the enclosing scope *s1*.

Once the parsing of the inner function completes, the current scope is reset to *s1*.

The compiler, however, cannot find the last occurrence of z in the current scope *s1*. Hence, it signals an error.

Now look at the first line in the example on page 10:

```
accountMaker = {(balance) -> {(amount) -> balance += amount}};
```

The AST (abstract syntax tree) generated by the compiler for the function is

```
: FunExpr
├─ balance                          parameters
├─                                  no locals
└─ : FunExpr                       code (expression)
   ├─ amount                           parameters
   ├─                                  no locals
   └─ : SetVarExpr                     code (expression)
      ├─ balance
      └─ balance + amount                  transformed compound assignment
```

A function (**FunExpr**) has three parts: a list of ids for the parameters, a list of ids for the locals and an expression representing its code. A function, however, *does not have a name*.

While generating the AST, the compiler first extends the current scope *s0* to *s1* = ⟨{balance}, *s0*⟩ and then to *s2* = ⟨{amount}, *s1*⟩. The inner assignment is parsed. The id amount is found directly in scope *s2* but balance is found indirectly in the enclosing scope *s1*.

The compiler continues by restoring the current scope to *s1* and finally to *s0*, without encountering scoping errors.

The second line on page 10

```
poorAccount = accountMaker(1000);
```

produces the AST

```
: SetVarExpr
├  poorAccount
└  : InvokeExpr
   ├  accountMaker        expression for the closure
   └  1000                list of expression for arguments
```

An invocation has two parts: an expression before the parentheses, which shall evaluate to a closure, and a list of expressions which are evaluated from left to right to obtain the arguments before they are applied to the closure as part of the invocation.

When expressions are evaluated upon *setting* ids, the evaluation is *eager*. When evaluation only happens when *getting* an id, it is *lazy*. Funny, as most other languages, is eager. Haskell is lazy.

Evaluating closures and interpreting applications

During execution, the interpreter maintains a *current environment* and an *execution stack* of environments.

Analogously to a scope, an environment consists of a *frame* and an *enclosing environment*. A frame, however, in not simply a list of ids but is a set of *mutable bindings* from ids to values.

A frame is written like {x ↦ 2, z ↦ "abc"}. An environment is denoted as a pair consisting of a frame and its enclosing environment, as in

$$e = \langle \{x \mapsto 2, z \mapsto \text{"abc"}\}, f \rangle$$

where the enclosing environment *f* might be something like

$$f = \langle \{x \mapsto 3, y \mapsto \text{true}\}, \text{null} \rangle$$

In environment *e*, getting the value of x returns 2, not 3. Getting y indirectly returns true from the enclosing environment *f*.

Now assume that the current environment *e1* looks like

$$e1 = \langle \{\text{accountMaker} \mapsto \text{nil}, \text{poorAccount} \mapsto \text{nil}, \text{richAccount} \mapsto \text{nil}\}, e0 \rangle$$

where *e0* is the enclosing environment.

When execution reaches the code on page 10, the interpreter first builds a closure *am* by combining the current environment *e1* with the AST of the function (see page 14), forming

$$am = [e1, \{(\text{balance}) \text{ -> } \{(\text{amount}) \text{ -> balance += amount}\}\}].$$

It then assigns *am* to accountMaker, updating the binding in the current environment:

$$e1 = \langle \{\text{accountMaker} \mapsto am, \text{poorAccount} \mapsto \text{nil}, \text{richAccount} \mapsto \text{nil}\}, e0 \rangle$$

Environments are *extended* when a closure is *applied* to arguments during an invocation. The closure's environment (no, *not* the current environment!) is extended with a new frame.

The new frame has a new binding for each of the parameters and the locals of the closure's function. The parameters are initialized with the values of the arguments of the invocation, whereas the locals are initialized with the default value nil.

The current environment is pushed on the execution stack and the extended environment becomes current. The code of the closure's function is then evaluated in turn.

When execution of this code completes, the last computed value is returned as the result of the invocation. The former current environment is restored by popping it from the execution stack and making it current again.

Take the invocation in

```
poorAccount = accountMaker(1000);
```

The interpreter pushes the current environment *e1* on the execution stack, extends the environment of *am* (incidentally, the same as the current one) with the frame {balance ↦ 1000} initialized with the argument, and sets it as current:

$$e2 = \langle \{ \text{balance} \mapsto 1000 \}, e1 \rangle.$$

Once again, *it is fundamental to extend the closure's environment*[1]*, not the current one*. However, the twos often coincide, as in this case.

---

[1] Remember, the closure's environment is the one *that was current when the closure was built*.

Then the function's code, {(amount) -> balance += amount}, a FunExpr in this case, is evaluated, here merely building another closure. The new closure is built around the current environment *e2*, resulting in

$$pa = [e2, \{(\text{amount}) \text{ -> } \text{balance} \mathrel{+}= \text{amount}\}]$$

The invocation terminates, returning *pa* and restoring the current environment to *e1*, which was pushed on the execution stack before.

The current environment is then updated to reflect the assignment to poorAccount. Thus

$$e1 = \langle \{\text{accountMaker} \mapsto am, \text{poorAccount} \mapsto pa, \text{richAccount} \mapsto \text{nil}\}, e0 \rangle$$

Similarly, the assignment

```
richAccount = accountMaker(1000000);
```

first evaluates the invocation, creating the environment

$$e3 = \langle \{\text{balance} \mapsto 1000000\}, e1 \rangle$$

then proceeds as above, returning the closure

$$ra = [e3, \{(\text{amount}) \text{ -> balance += amount}\}]$$

Note that the fresh frame {balance ↦ 1000000} in *e3* has a fresh binding for balance, totally independent from that of the similar frame in *e2*. (The AST of the function, though, is shared between *pa* and *ra*: it is built only once by the compiler.)

After the assignment to richAccount, the current environment is

$$e1 = \langle \{\text{accountMaker} \mapsto am, \text{poorAccount} \mapsto pa, \text{richAccount} \mapsto ra\}, e0 \rangle$$

Now consider the invocation

```
poorAccount(50);
```

The interpreter saves the current environment *e1* onto the stack, extends the closure's environment *e2* (this time it differs from the current one) with a new frame and sets it as current:

$$e4 = \langle \{\text{amount} \mapsto 50\}, e2 \rangle$$

The interpreter then continues by executing the function's code `balance += amount` in the current environment *e4*, thus indirectly updating *e2*, the enclosing environment:

$$e2 = \langle \{\text{balance} \mapsto 1050\}, e1 \rangle$$

The invocation returns by resetting the current environment to *e1*. This has no binding for `balance`, but *pa*'s environment still holds it.

Similarly, the invocation

```
richAccount(100000);
```

updates the binding of balance in *ra*'s environment *e3*. It does so by first extending *e3* to

$$e5 = \langle \{\text{amount} \mapsto 100000\}, e3 \rangle$$

and then executing the function's code balance += amount. As above, this indirectly updates the binding of balance in *e3* to 1100000. Finally, the invocation returns, re-establishing *e1* as the current environment.

The grand result is that the two counters are updated independently, since the incrementing assignment operates in different environments, updating different bindings.

Exercise 4

Funny does not offer structured data types. However, here is how to implement an immutable pair.

Below, pair refers to a closure that accepts two parameters and returns something which can be passed to first and second to get the two original parameters back. This is *behaviorally* a pair.

Study this example carefully. It might be hard to understand it at first sight. Suddenly, however, it becomes very clear.

```
pair = {(f, s) -> {(p) -> p(f, s)}};
first = {(p) -> p({(f, s) -> f})};
second = {(p) -> p({(f, s) -> s})};

tree = pair(1, pair(pair("two", 42), pair(true, nil)));
println(second(first(second(tree)))); // prints 42
```

Other basics

Above, for the sake of discussion, we assumed that some current scope and some current environment were already in place.

In reality, it all begins with a Funny *program*. This is a parameterless function, possibly with locals, like in

```
{x, y -> …}
```

The Funny runtime first compiles the function and then asks the interpreter to execute the invocation

```
{x, y -> …}()
```

We learned before what this entails and won't repeat it here.

The point is that compilation starts with the *empty* scope and interpretation starts with the *empty* execution stack and the *empty* environment as the current one.

Here are some notes about Funny:

- There's no *global scope*. Every id must be declared inside some function before it is used.
- There are only two ways to declare an id: as a parameter or as a local. The parameters and locals of a function must all be different from each other.
- As usual, a declaration can *hide* an outer declaration for the same id.
- Functions are always anonymous.
- For simplicity, the compiler just throws an exception and terminates when it encounters an error. Of course, an industrial-grade compiler should reasonably keep going.

More on Funny

Funny syntax is *expression-oriented*. Functions, conditionals, loops, assignments, etc., are all expressions. At runtime, the interpreter *evaluates* them, producing a *value*. *Literals* are expressions that are already in value form: thus, when the interpreter evaluates the expression 2.5, it simply returns it as it is, without further ado.

Funny has booleans, closures, numbers, strings and nil as values. It does *not* offer more complex values like arrays, tuples, structures or objects.

Except for closures, every other value also has a literal form.

A conditional returns the value of the branch that happens to be evaluated at runtime.

A loop is used mainly for its *side effects* on environments, not for its intrinsic value. Indeed, it always evaluates to nil.

The value of an assignment is the new value of its left-side id. Assignments are used to update the bindings in environments.

The value of a sequence of expressions is the value of its last expression. The subexpressions of a sequence, except the last, are there *only* for their side effects on the current environment.

Here's an example:

```
x += sign = if x >= 0 then 0 else -1 fi;
whilenot n <= 0 do n -= 1 od;
```

When x is negative, sign is set to -1 and x is decremented. Otherwise, sign is set to 0 and x is still updated, although with a new value which happens to be numerically equal to its current value. The whole snippet has value nil, the value of the last expression, the loop. Whereas this value by itself is uninteresting, evaluating the loop can change the environment many times.

To express recursion, nothing new is required:

```
{isOdd, isEven ->
    isOdd = {(n) -> if n == 0 then false else isEven(n - 1) fi};
    isEven = {(n) -> if n == 0 then true else isOdd(n - 1) fi};

    println(isEven(1000));
}
```

This complete Funny program prints true. (Sure, the program has some nasty problems: can you spot them?)

Ids in Funny are as in Java, e.g., identifier and 识别码.

The literals are rather standards:

- numbers: 123, 12.3, .3, 12.3e-45, 1234.5678;
- strings: "Hello, world!\n", "你好";
- booleans: false, true;
- nil: nil.

Rather than presenting a concrete lexical syntax, however, a tokenizer is provided for your convenience. The tokenizer also supports /*arbitrarily /* nested comments /*like these*/, something that /*cannot*/ be formalized */ with regular expressions */. Why are regular expressions insufficient for this?

The grammar

Funny's grammar is presented next. You might not be accustomed to this way of writing a grammar. I'm sure you will understand it nevertheless. As usual

- * means 0 or more repetitions;
- + means 1 or more repetitions;
- ? means an option, (0 or 1 repetitions);
- | means an alternative.

In what follows, Num, True, False, String, Nil and Id stand for the tokens as returned by the tokenizer.

```
program ::= function Eos .
function ::= "{" optParams optLocals optSequence "}" .

optParams ::= ( "(" optIds ")" )? .
optLocals ::= optIds.
optSequence ::= ( "->" sequence )? .

optIds::= ( id ( "," id )* )? .
id ::= Id .
```

This shouldn't come to a surprise. Here Eos is the token for the end-of-stream: a program is a function *followed by nothing else*.

Sequences of characters enclosed in quotation marks are tokens.

An empty optSequence is translated to nil.

```
sequence ::= optAssignment ( ";" optAssignment )* .
optAssignment := assignment? .
assignment ::= Id ( "=" | "+=" | "-=" | "*=" | "/=" | "%=" ) assignment
    | logicalOr .
```

An empty optAssignment is translated to nothing at all. This implies that, although the semicolon ";" is a *separator* rather than a *terminator*, the syntax allows liberal use of it in a sequence, even in a terminal position. However, a totally empty sequence is still translated to nil: every expression must have a value!

The grammar also reflects the fact that assignments *associate to the right*: x = y = 5 indeed means x = (y = 5). This makes sense, since assignments are expressions, thus have values at runtime. Association to the left wouldn't make any sense.

```
logicalOr ::= logicalAnd ( "||" logicalOr )? .
logicalAnd ::= equality ( "&&" logicalAnd )? .
equality ::= comparison ( ( "==" | "!=" ) comparison )? .
comparison ::= add ( ( "<" | "<=" | ">" | ">=" ) add )? .
add ::= mult ( ( "+" | "-" ) mult )* .
mult ::= unary ( ( "*" | "/" | "%" ) unary )* .
unary ::= ( "+" | "-" | "!" ) unary
    | postfix .
```

All this should be rather familiar. The grammar reflects the operators' priorities, from lower to higher.

The use of *repetitions* avoids the left recursion problem that makes a LL(*) parsing impossible. However, it requires stating explicitly that same level operators *associate to the left*: 2 / 3 * 4 means (2 / 3) * 4. With a left-recursive grammar, this would be implicit.

Another way is to make use of right-recursive productions (see exercise 6 on page 41.

```
postfix ::= primary args* .
args ::= "(" ( sequence ( "," sequence )* )? ")" .
primary ::= num | bool | nil | string
    | getId
    | function
    | subsequence
    | cond
    | loop
    | print .
num ::= Num .
bool ::= True | False .
nil ::= Nil .
string ::= String .
getId ::= Id .
subsequence ::= "(" sequence ")" .
cond ::= ( "if" | "ifnot" ) sequence "then" sequence ( "else" sequence )? "fi" .
loop ::= ( "while" | "whilenot" ) sequence ( "do" sequence )? "od" .
print ::= ( "print" | "println" ) args .
```

That's it!

Exercise 5

The grammar is *almost* LL(1). There is a place, however, where it is LL(2). This requires the tokenizer to be able to *step back* by one token. Take a look at its implementation.

Can you spot where the grammar is, indeed, LL(2)?

Exercise 6

Rewrite the grammar using only recursion, never repetitions. Get rid of left-recursive productions by applying the usual transformations. E.g., the left-recursive production

```
add ::= add ( "+" | "-" ) mult
    | mult .
```

is transformed into the right-recursive productions

```
add ::= mult restAdd .
restAdd ::= ( ( "+" | "-" ) mult restAdd )? .
```

Semantics

The semantics of Funny has been explained in some details with respect to closures and invocations. The rest is very straightforward. Here are some notes:

- The compiler transforms compound assignments, e.g., `x *= y`, into simple assignments, like `x = x * y`.
- Like in many other programming languages, `||` and `&&` are *short-circuiting*: the compiler transforms them into equivalent conditionals.
- Numerical computations are carried out on the underlying Java class `java.math.BigDecimal`.
- Addition on strings means concatenation. Adding a string and something else first converts the non-string to a string representation.

Almost LL(1) parsing

The grammar has been organized to facilitate the construction of a LL(1) parser (almost).

The compiler maintains references to the tokenizer and to the current scope.

As usual for top-down recursive-descent parsing:

- each production is implemented by a method with a similar name;
- repetitions are implemented with `while` loops;
- options are implement with `if` or `switch` conditionals;
- a non-terminal in a production's body means a call to the associated method;
- a terminal is compared to the current token, which is then skipped on a match, or an error is signaled otherwise.

Building the AST

The nodes in the AST are represented by types which are subtypes of Expr:

```
abstract class Expr
    abstract Val eval(Env env);
```

The method is the evaluator of the interpreter, the fundamental mechanism that reduces an expression to its value in the given environment. Subtypes include: BinaryExpr, FunExpr, GetVarExpr, IfExpr, InvokeExpr, PrintExpr, SeqExpr, SetVarExpr, UnaryExpr, WhileExpr and Val.

Type Val represents values, i.e., expressions which return themselves when evaluated:

```
abstract class Val extends Expr
    Val eval(Env env) { return this; }
```

Subtypes thereof are: BoolVal, ClosureVal, NilVal, NumVal and StringVal.

Let's study some examples on how to build the AST. The production unary is right-recursive on purpose, to reflect the right-associativity of unary operators. Thus, the AST for a row of unary operations shall be built from right to left, even on a left to right parsing! Here's a sketch of the code, where the methods now return instances of Expr, representing the corresponding abstract syntax (sub)tree:

```
private Expr unary() {
    if (!isUnaryOp())
        return postfix();
    Type oper = type();
    next();
    return new UnaryExpr(oper, unary());
}
```

The interesting part is the last line. The recursive invocation of the method in the argument position of the constructor for UnaryExpr perfectly matches the right to left buildup of the AST.

Left-associative operations, like addition, are better described by left-recursive productions. But this would make a LL(1) parsing impossible. The first way out to overcome some forms of left-recursion is to use repetitions, as in the production for add. This leads to code similar to the following for building the AST:

```
private Expr add() {
    Expr expr = mult();
    while (isAddOp()) {
        Type oper = type();
        next();
        expr = new BinaryExpr(oper, expr, mult());
    }
    return expr;
}
```

The structure is quite different from above. The loop *requires the updating* of a local variable, whereas the previous solution was rather *functional*, without real side-effects.

The other alternative is to *rewrite* left-recursive productions, as done in exercise 6 (p. 41). This leads to code like the following:

```
private Expr add() {
    return restAdd(mult());
}

private Expr restAdd(Expr expr) {
    if (!isAddOp())
        return expr;
    Type type = type();
    next();
    return restAdd(new BinaryExpr(type, expr, mult()));
}
```

Since no updates are required (only initializations), the code is quite functional. The second method is rather peculiar, as it really needs an additional parameter to pass information around. But it works and builds a left-leaning AST *despite a right-recursive production*!

These are all pragmatic examples of *syntax-directed translations*, where the structure of the parse tree *drives* the translation into some other form, the AST in this case. The Funny compiler, and those of many other languages, does not even need to build the parse tree explicitly. It only does so implicitly, during the check that the input text matches the grammar. While doing so, it builds the AST at the same time, almost as a by-product.

Syntax-directed translation is supported by a nice and rich theory, usually covered in some details in most textbooks on compilation.