Chapter 4
Interpretation

Any good software engineer will tell you that a compiler and an interpreter are interchangeable.

Tim Berners-Lee (1955–)

After lexing and parsing, we have the abstract syntax tree of a program as a data structure in memory. But a program needs to be executed, and we have not yet dealt with that issue.

The simplest way to execute a program is *interpretation*. Interpretation is done by a program called an *interpreter*, which takes the abstract syntax tree of a program and executes it by inspecting the syntax tree to see what needs to be done. This is similar to how a human evaluates a mathematical expression: We insert the values of variables in the expression and evaluate it bit by bit, starting with the innermost parentheses and moving out until we have the result of the expression. We can then repeat the process with other values for the variables.

There are some differences, however. Where a human being will copy the text of the formula with variables replaced by values, and then write a sequence of more and more reduced copies of a formula until it is reduced to a single value, an interpreter will keep the formula (or, rather, the abstract syntax tree of an expression) unchanged and use a symbol table to keep track of the values of variables. Instead of reducing a formula, the interpreter is a function that takes an abstract syntax tree and a symbol table as arguments and returns the value of the expression represented by the abstract syntax tree. The function can call itself recursively on parts of the abstract syntax tree to find the values of subexpressions, and when it evaluates a variable, it can look its value up in the symbol table.

This process can be extended to also handle statements and declarations, but the basic idea is the same: A function takes the abstract syntax tree of the program and, possibly, some extra information about the context (such as a symbol table or the input to the program) and returns the output of the program. Some input and output may be done as side effects by the interpreter.

T. Æ. Mogensen, *Introduction to Compiler Design*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-46460-7_4

We will in this chapter assume that the symbol tables are persistent, so no explicit action is required to restore the symbol table for the outer scope when exiting an inner scope. In the main text of the chapter, we don't need to preserve symbol tables for inner scopes once these are exited (so a stack-like behaviour is fine), but in one of the exercises we will need symbol tables to persist after their scope is exited.

4.1 The Structure of an Interpreter

An interpreter will typically consist of one function per syntactic category. Each function will take as arguments an abstract syntax tree from the syntactic category and, possibly, extra arguments such as symbol tables. Each function will return one or more results, which can be the value of an expression, an updated symbol table, or nothing at all (relying on side effects to, e.g., symbol tables).

These functions can be implemented in any programming language for which we already have an implementation. This implementation can also be an interpreter, or it can be a compiler that compiles to some other language. Eventually, we will need to either have an interpreter written in machine language or a compiler that compiles to machine language. For the moment, we just write interpretation functions in a notation reminiscent of a high-level programming language and assume an implementation of this exists. Additionally, we want to avoid being specific about how abstract syntax is represented, so we will use a notation that looks like concrete syntax to represent abstract syntax.

4.2 A Small Example Language

We will use a small (somewhat contrived) language to show the principles of interpretation. The language is a first-order functional language with recursive definitions. The syntax is given in Grammar 4.1. The shown grammar is clearly ambiguous, but we assume that any ambiguities have been resolved during parsing, so we have an unambiguous abstract syntax tree.

In the example language, a program is a list of function declarations. The functions are all mutually recursive, and no function may be declared more than once. Each function declares its result type and the types and names of its arguments. Types are `int` (integer) and `bool` (Boolean). There may not be repetitions in the list of parameters for a function. Functions and variables have separate name spaces. The body of a function is an expression, which may be an integer constant, a variable name, a sum-expression, a comparison, a conditional, a function call or an expression with a local declaration. Comparison is defined both on booleans (where ***false*** is considered smaller than ***true***) and integers, but addition is defined only on integers.

Grammar 4.1 Example language for interpretation

$$
\begin{array}{lll}
\mathit{Program} & \rightarrow & \mathit{Funs} \\
\\
\mathit{Funs} & \rightarrow & \mathit{Fun} \\
\mathit{Funs} & \rightarrow & \mathit{Fun}\ \mathit{Funs} \\
\\
\mathit{Fun} & \rightarrow & \mathit{TypeId}\ (\ \mathit{TypeIds}\ ) = \mathit{Exp} \\
\\
\mathit{TypeId} & \rightarrow & \texttt{int}\ \textbf{id} \\
\mathit{TypeId} & \rightarrow & \texttt{bool}\ \textbf{id} \\
\\
\mathit{TypeIds} & \rightarrow & \mathit{TypeId} \\
\mathit{TypeIds} & \rightarrow & \mathit{TypeId}\ \texttt{,}\ \mathit{TypeIds} \\
\\
\mathit{Exp} & \rightarrow & \textbf{num} \\
\mathit{Exp} & \rightarrow & \textbf{id} \\
\mathit{Exp} & \rightarrow & \mathit{Exp} + \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \mathit{Exp} < \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \texttt{if}\ \mathit{Exp}\ \texttt{then}\ \mathit{Exp}\ \texttt{else}\ \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \textbf{id}\ (\ \mathit{Exps}\ ) \\
\mathit{Exp} & \rightarrow & \texttt{let}\ \textbf{id} = \mathit{Exp}\ \texttt{in}\ \mathit{Exp} \\
\\
\mathit{Exps} & \rightarrow & \mathit{Exp} \\
\mathit{Exps} & \rightarrow & \mathit{Exp}\texttt{,}\ \mathit{Exps}
\end{array}
$$

A program must contain a function called `main`, which has one integer argument and which returns an integer. Execution of the program is by calling this function with the input (which must be an integer). The result of this function call (also an integer) is the output of the program.

4.3 An Interpreter for the Example Language

An interpreter for this language must take the abstract syntax tree of the program and an integer (the input to the program) and return another integer (the output of the program). Since values can be both integers or booleans, the interpreter uses a value type that contains both integers and booleans (and enough information to tell them apart). We will not go into detail about how such a type can be defined but simply assume that there are operations for testing if a value is a boolean or an integer and operating on values known to be integers or booleans. If we during interpretation find that we are about to, say, add a boolean to an integer, we stop the interpretation with an error message. We do this by letting the interpreter call a function called **error**().

We will start by showing how we can evaluate (interpret) expressions, and then extend this to handle the whole program.

4.3.1 Evaluating Expressions

When we evaluate expressions, we need, in addition to the abstract syntax tree of the expression, also a symbol table *vtable* that binds variables to their values. Additionally, we need to be able to handle function calls, so we also need a symbol table *ftable* that binds function names to the abstract syntax trees of their declarations. The result of evaluating an expression is the value of the expression.

For terminals (variable names and numeric constants) with attributes, we assume that there are predefined functions for extracting these attributes. Hence, **id** has an associated function *getname*, that extracts the name of the identifier. Similarly, **num** has a function *getvalue*, that returns the value of the number.

Figure 4.2 shows a function $Eval_{Exp}$, that takes an expression *Exp* and symbol tables *vtable* and *ftable*, and returns a value, which may be either an integer or a boolean. Also shown is a function $Eval_{Exps}$, that evaluates a list of expressions to a list of values. We also use a function $Call_{Fun}$ that handles function calls. We will define this later.

The main part of $Eval_{Exp}$ is a case-expression (in some languages called `switch` or `match`) that identifies which kind of expression the top node of the abstract syntax tree represents. The patterns are shown as concrete syntax, but you should think of it as pattern matching on the abstract syntax. The column to the right of the patterns shows the actions needed to evaluate the expressions. These actions can refer to parts of the pattern on the left. An action is a sequence of definitions of local variables followed by an expression (in the interpreter) that evaluates to the result of the expression that was given (in abstract syntax) as argument to $Eval_{Exp}$.

We will briefly explain each of the cases handled by $Eval_{Exp}$.

- The value of a number is found as the *value* attribute to the node in the abstract syntax tree.
- The value of a variable is found by looking its name up in the symbol table for variables. If the variable is not found in the symbol table, the lookup-function returns the special value *unbound*. When this happens, an error is reported and the interpretation stops. Otherwise, it returns the value returned by *lookup*.
- At a plus-expression, both arguments are evaluated, then it is checked that they are both integers. If they are, we return the sum of the two values. Otherwise, we report an error (and stop).
- Comparison requires that the arguments have the same type. If that is the case, we compare the values, otherwise we report an error.
- In a conditional expression, the condition must be a boolean. If it is, we check if it is **true**. If so, we evaluate the then-branch, otherwise, we evaluate the else-branch. If the condition is not a boolean, we report an error.
- At a function call, the function name is looked up in the function environment to find its definition. If the function is not found in the environment, we report an error. Otherwise, we evaluate the arguments by calling $Eval_{Exps}$ and then call $Call_{Fun}$ to find the result of the call.

Fig. 4.2 Evaluating expressions

```
$Eval_{Exp}(Exp, vtable, ftable)$ = case $Exp$ of
----------------------------------------------------------------------
$\textbf{num}$              | $getvalue(\textbf{num})$
----------------------------------------------------------------------
$\textbf{id}$               | $v = lookup(vtable, getname(\textbf{id}))$
                            | if $v = unbound$
                            | then $\textbf{error}()$
                            | else $v$
----------------------------------------------------------------------
$Exp_1 + Exp_2$             | $v_1 = Eval_{Exp}(Exp_1, vtable, ftable)$
                            | $v_2 = Eval_{Exp}(Exp_2, vtable, ftable)$
                            | if $v_1$ and $v_2$ both are integers
                            | then $v_1 + v_2$
                            | else $\textbf{error}()$
----------------------------------------------------------------------
$Exp_1 < Exp_2$             | $v_1 = Eval_{Exp}(Exp_1, vtable, ftable)$
                            | $v_2 = Eval_{Exp}(Exp_2, vtable, ftable)$
                            | if $v_1$ and $v_2$ both are integers
                            | then if $v_1 < v_2$ then $\textbf{true}$ else $\textbf{false}$
                            | else if $v_1$ and $v_2$ both are booleans
                            | then if $v_1 = \textbf{false}$ then $v_2$ else $\textbf{false}$
                            | else $\textbf{error}()$
----------------------------------------------------------------------
if $Exp_1$                  | $v_1 = Eval_{Exp}(Exp_1, vtable, ftable)$
then $Exp_2$                | if $v_1$ is a boolean
else $Exp_3$                | then if $v_1 = \textbf{true}$
                            |   then $Eval_{Exp}(Exp_2, vtable, ftable)$
                            |   else $Eval_{Exp}(Exp_3, vtable, ftable)$
                            | else $\textbf{error}()$
----------------------------------------------------------------------
$\textbf{id}$ ( $Exps$ )    | $def = lookup(ftable, getname(\textbf{id}))$
                            | if def $= unbound$
                            | then $\textbf{error}()$
                            | else
                            |   $args = Eval_{Exps}(Exps, vtable, ftable)$
                            |   $Call_{Fun}(def, args, ftable)$
----------------------------------------------------------------------
let $\textbf{id}$ = $Exp_1$ | $v_1 = Eval_{Exp}(Exp_1, vtable, ftable)$
in $Exp_2$                  | $vtable' = bind(vtable, getname(\textbf{id}), v_1)$
                            | $Eval_{Exp}(Exp_2, vtable', ftable)$
```

```
$Eval_{Exps}(Exps, vtable, ftable)$ = case $Exps$ of
----------------------------------------------------------------------
$Exp$            | $[Eval_{Exp}(Exp, vtable, ftable)]$
----------------------------------------------------------------------
$Exp$ , $Exps$   | $Eval_{Exp}(Exp, vtable, ftable)$
                 |   $:: Eval_{Exps}(Exps, vtable, ftable)$
```

- A `let`-expression declares a new variable with an initial value defined by an expression. The expression is evaluated and the symbol table for variables is extended using the function *bind* to bind the variable to the value. The extended table is used when evaluating the body-expression, which defines the value of the whole expression. Note that we do not explicitly restore the symbol table after exiting the scope of the `let`-expression. The old symbol table is implicitly preserved.

$Eval_{Exps}$ builds a list of the values of the expressions in the expression list. The notation is taken from SML and F#: A list is written in square brackets, and the infix operator :: adds an element to the front of a list. This operator can also be used in patterns to match non-empty lists and bind variables to the head and tail of this list.

Suggested exercises: 4.1.

4.3.2 Interpreting Function Calls

A function declaration explicitly declares the types of the arguments. When a function is called, we must check that the number of arguments is the same as the declared number, and that the values of the arguments match the declared types.

If this is the case, we build a symbol table that binds the parameter variables to the values of the arguments and use this in evaluating the body of the function. The value of the body must match the declared result type of the function.

$Call_{Fun}$ is also given a symbol table for functions, which is passed to the $Eval_{Exp}$ when evaluating the body.

$Call_{Fun}$ is shown in Fig. 4.3, along with the functions for *TypeId* and *TypeIds*, which it uses. The function Get_{TypeId} just returns a pair of the declared name and type, and $Bind_{TypeIds}$ checks the declared type against a value and, if these match, builds a symbol table that binds the name to the value (and reports an error if they do not match). $Binds_{TypeIds}$ also checks if all parameters have different names by seeing if the current name is already bound. *emptytable* is an empty symbol table. Looking any name up in the empty symbol table returns *unbound*. The underscore used in the last rule for $Bind_{TypeIds}$ is a wildcard pattern that matches anything, so this rule is used when the number of arguments do not match the number of declared parameters.

4.3.3 Interpreting a Program

Running a program is done by calling the `main` function with a single argument that is the input to the program. So we build the symbol table for functions, look up `main` in this and call $Call_{Fun}$ with the resulting definition and an argument list containing just the input.

Hence, $Run_{Program}$, which runs the whole program, calls a function $Build_{ftable}$ that builds the symbol table for functions. This, in turn, uses a function Get_{fname} that finds the name of a function. All these functions are shown in Fig. 4.4.

This completes the interpreter for our small example language.

$Call_{Fun}(Fun, args, ftable) =$ `case` Fun `of`	
$TypeId$ ($TypeIds$) $=$ Exp	$(f, t_0) = Get_{TypeId}(TypeId)$ $vtable = Bind_{TypeIds}(TypeIds, args)$ $v_1 = Eval_{Exp}(Exp, vtable, ftable)$ *if* v_1 is of type t_0 *then* v_1

$Get_{TypeId}(TypeId) =$ `case` $TypeId$ `of`	
`int` **id**	($getname$(**id**), `int`)
`bool` **id**	($getname$(**id**), `bool`)

$Bind_{TypeIds}(TypeIds, args) =$ `case` $(TypeIds, args)$ `of`	

($TypeId$,
$[v]$)
	$(x, t) = Get_{TypeId}(TypeId)$
	if v is of type t
	then $bind(emptytable, x, v)$
	else **error**()

($TypeId$ `,` $TypeIds$, $(v :: vs)$)	$(x, t) = Get_{TypeId}(TypeId)$ $vtable = Bind_{TypeIds}(TypeIds, vs)$ *if* $lookup(vtable, x) = unbound$ *and* v is of type t *then* $bind(vtable, x, v)$ *else* **error**()
_	**error**()

Fig. 4.3 Evaluating a function call

While we have illustrated interpretation mainly by a single example, the methods carry over to other languages: We build one or more function for each syntactic category. These may, in addition to the abstract syntax tree, also take other parameters such as symbol tables, and they return values that are used in other parts of the interpreter (or represent part of the output).

Suggested exercises: 4.5.

4.4 Advantages and Disadvantages of Interpretation

Once you have a abstract syntax tree, interpretation is often the simplest way of executing a program. However, it is also a relatively slow way to do so. When we perform an operation in the interpreted program, the interpreter must first inspect the abstract syntax tree to see what operation it needs to perform, then it must check that the types of the arguments to the operation match the requirements of the operation, and only then can it perform the operation. Additionally, each value must include

Fig. 4.4 Interpreting a program

$Run_{Program}(Program, input) =$ `case` $Program$ `of`	
$Funs$	$ftable = Build_{ftable}(Funs)$ $def = lookup(ftable,$ `main`$)$ if def $= unbound$ $then$ ***error***$()$ $else$ $Call_{Fun}(def, [input], ftable)$

$Build_{ftable}(Funs) =$ `case` $Funs$ `of`	
Fun	$f = Get_{fname}(Fun)$ $bind(emptytable, f, Fun)$
$Fun\ Funs$	$f = Get_{fname}(Fun)$ $ftable = Build_{ftable}(Funs)$ $if\ lookup(ftable, f) = unbound$ $then\ bind(ftable, f, Fun)$ $else$ **error**$()$

$Get_{fname}(Fun) =$ `case` Fun `of`	
$TypeId$ ($TypeIds$) $=$ Exp	$(f, t_0) = Get_{TypeId}(TypeId)$ f

sufficient information to identify its type, so after doing, say, an addition, we must add type information to the resulting number.

It should be clear from this that we spend much more time on figuring out what to do, and whether doing it is O.K., than on actually doing it.

To get faster execution, we can use the observation that a program that executes each part of the program only once will finish quite quickly. In other words, any time-consuming program will contain parts that are executed many times. The idea is that we can do the inspection of the abstract syntax tree and the type checking only once for each program part, and only repeat the actual operations that are performed in this part. Since performing the operations is a small fraction of the total time spent in an interpreter, we can get a substantial speedup by doing this. This is the basic idea behind *static type checking* and *compilation*.

Static type checking checks the program *before* it is executed for *potential* mismatches between the types of values and the requirements of operations. It does this check for the whole program regardless of whether all parts will actually be executed, so it may report errors even if an interpretation of the program would finish without errors. So static type checking puts extra limitations on programs, but reduces the time needed at runtime to check for errors and, as a bonus, reports potential problems before a program is executed, which can help when debugging a program. We look at static type checking in Chap. 5. Static type checking does, however, need some time to do the checking before we can start executing the program, so the time from doing an edit in a program to executing it will increase slightly.

Compilation gets rid of the abstract syntax tree of the source program by translating it into a target program (in a language of which we already have an implementation) that only performs the operations in the source program, having done (most of) the checks during compilation. Usually, the target language is a low-level language such as machine code, but it can also be another high-level language. Like static checking, compilation must (at least conceptually) complete before execution can begin, so it adds delay between editing a program and running it.

Usually, static checking and compilation go hand in hand, but you can find compilers for languages with dynamic (run-time) type checking as well as interpreters for statically typed languages.

Some implementations combine interpretation and compilation: The first few times a function is called, it is interpreted, but if it is called sufficiently often, it is compiled and all subsequent calls to the function will execute the compiled code. This is often called *just-in-time compilation*, though this term was originally used for just postponing compilation of a function until just before the first time it is called, hence reducing the delay from editing a program to its execution, but at the cost of adding small delays for compilation during execution. We will in this book only look at "pure" interpretation and compilation, though.

4.5 Further Reading

A step-by-step construction of an interpreter for a LISP-like language is shown in [2]. A survey of programming language constructs (also for LISP-like languages) and their interpretation is shown in [1].

4.6 Exercises

Exercise 4.1 We extend the language from Sect. 4.2 with Boolean operators. We add the following productions to Grammar 4.1:

$$\begin{aligned} &Exp \rightarrow \texttt{not}\ Exp \\ &Exp \rightarrow Exp\ \texttt{and}\ Exp \end{aligned}$$

When evaluating `not` e, we first evaluate e to a value v that is checked to be a boolean. If it is, we return $(\neg\, v)$, where $\neg$ is logical negation.

When evaluating e_1 `and` e_2, we first evaluate e_1 and e_2 to values v_1 and v_2 that are both checked to be booleans. If they are, we return $(v_1 \wedge v_2)$, where $\wedge$ is logical conjunction.

Extend the interpretation function in Fig. 4.2 to handle these new constructions as described above.

Exercise 4.2 Add the productions

$$\begin{aligned} &\textit{Exp} &&\rightarrow \textbf{floatconst} \\ &\textit{TypeId} &&\rightarrow \texttt{float}\ \textbf{id} \end{aligned}$$

to Grammar 4.1. This introduces floating-point numbers to the language. The operator $+$ is overloaded so it can do integer addition or floating-point addition, and $<$ is extended so it can also compare pairs of floating point numbers.

(a) Extend the interpretation functions in Figs. 4.2–4.4 to handle these extensions.
(b) We now add implicit conversion of integers to floats to the language, using the rules: Whenever an operator has one integer argument and one floating-point argument, the integer is converted to a float. Extend the interpretation functions from question a) above to handle this.

Exercise 4.3 The language defined in Sect. 4.2 declares types of parameters and results of functions. The interpreter in Sect. 4.3 adds explicit type information to every value, and checks this before doing any operations on values. So, we could omit type declarations and rely solely on the type information in values.

Replace in Grammar 4.1 *TypeId* by **id** and rewrite the interpretation functions in Fig. 4.3 so they omit checking types of parameters and results, but still check that the number of arguments match the declaration and that no parameter name is repeated.

Exercise 4.4 In the language defined in Sect. 4.2, variables bound in `let`-expressions have no declared type, so it is possible to write a program where a `let`-bound variable in a let-expression sometimes is bound to an integer and at other times to a Boolean value.

Write an example of such a program.

Exercise 4.5 We extend the language from Sect. 4.2 with functional values. These require *lexical closures*, which is a record/struct/tuple that contains an argument name, an expression, and an environment. We assume symbol tables are fully persistent, so environments in closures persist. We add the following productions to Grammar 4.1:

$$\begin{aligned} &\textit{TypeId} &&\rightarrow \texttt{fun}\ \textbf{id} \\ &\textit{Exp} &&\rightarrow \textit{Exp}\ \ \textit{Exp} \\ &\textit{Exp} &&\rightarrow \texttt{fn}\ \textbf{id}\ \ \texttt{=>}\ \textit{Exp} \end{aligned}$$

The notation for anonymous functions is taken from Standard ML. Evaluating $\texttt{fn}\ x\texttt{=>}e$ in an environment *vtable* produces a functional value f, which is a lexical closure consisting of the name x, the expression e, and the environment *vtable*. When f is applied to an argument v, it is checked that v is an integer. If this is the case, e is evaluated in *vtable* extended with a binding that binds x to v. We then check if the result w of this evaluation is an integer, and if so use it as the result of the function application.

When evaluating e_1 e_2, we evaluate e_1 to a functional value f and e_2 to an integer v, and then apply f to v as described above.

Extend the interpreter from Fig. 4.3 to handle these new constructions as described above.

References

1. Abelson, H., Sussman, G.J., Sussman, J.: Structure and Interpretation of Computer Programs. MIT Press (1996). Also downloadable from https://mitpress.mit.edu/sicp/full-text/sicp/book/
2. Steele, G.L., Sussman, G.J.: The Art of the Interpreter or, The Modularity Complex. Tech. Rep. AIM-453, Massachusetts Institute of Technology, Cambridge, MA, USA (1978)