Lecture 3: Interpreters; An interpreter for SAE

CS4400 Programming Languages

Abstract Syntax Trees; Interpreters; Compilers; Meaning

Readings:

Overview

In this lecture, we will look at compilation vs interpretation, and we will implement our first simple interpreter. In our Haskell corner, we will talk more about types, in particular polymorphic types and overloading. We will also talk about program layout and local binding constructs.

Recap

In the previous lecture, we talked about different aspect of programming language syntax and representation. Concrete syntax is concerned with the text that represents a program to us, humans (via a text-editor, on Github, etc.). It takes care of all the details, such as how are expressions parenthesized, what is the separator between statements, how are code blocks delimited, etc. Abstract syntax hides those details and usually only captures the abstract structure of programming phrases. E.g., we will know that we are adding a number to a variable, but we don't care whether it was parenthesized in the original program text. We don't even care what the symbol for addition is.

To get a better handle on abstract syntax, its relation to a textual representation, and to prepare for the next step, let's do some examples.

Recall SAE and its representation in Haskell. Mentally parse the following expressions and write them down as a Haskell datatype.

```
(1 + 2) - 3
44 / ((2 * 5) + 1)
((1 - 42) + 8) / (11 * (3 + 21))
```

Aside: Abstract Syntax Trees

You might have realized that any (non-cyclic) structured datatype value forms a tree. Actually, the abstract syntax representation of term/phrases/expressions is called an *abstract syntax tree* (AST). These can be drawn as follows:

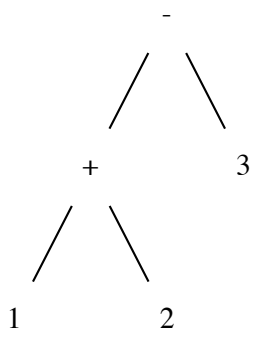

Figure 1: AST for (1 + 2) - 3

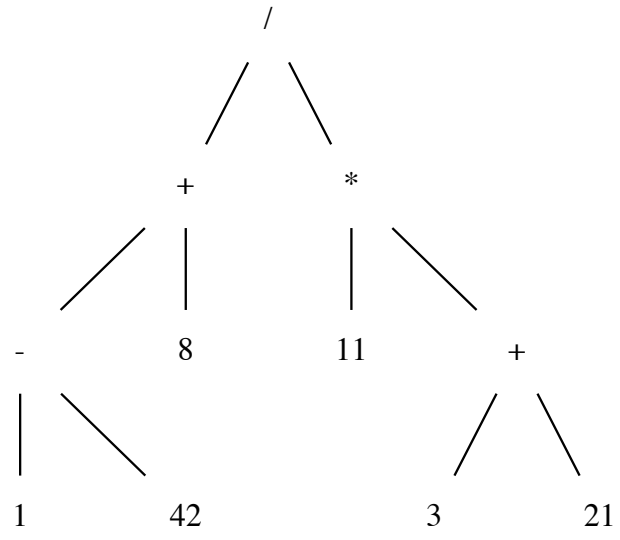

Figure 2: AST for ((1 - 42) + 8) / (11 * (3 + 21))

Interpreters and Compilers

Now that we (at least theoretically) know that there's a path from a program text to a much more useful representation as an abstract syntax tree.

Basically, programming languages are usually either interpreted or compiled. Some implementations use a mix of both

In a compiled situation, the program gets translated from the high-level programming language, represented as text, into a low-level *semantically equivalent* program, typically in machine code for a specific architecture.

There are multiple steps involved. Typically: - front-end - lexing + parsing - semantic analysis (type checking) - intermediate code generation - back-end - optimizations - target code generation - target code optimizations - linking

There can be variations on this but this is roughly a classic compiler. After the program is compiled it is typically used standalone.

With an interpreted language, the program, as well as the program's input, form the input of an interpreter. The interpreter is around during the whole execution of the program and forms an interface between the program and the rest of the system.

Some languages use a mix of both. A compiler (often just-in-time) will compile into some intermediate, lower-level representation, that gets interpreted upon execution. In this situation, the interpreter is usually called a *virtual machine*.

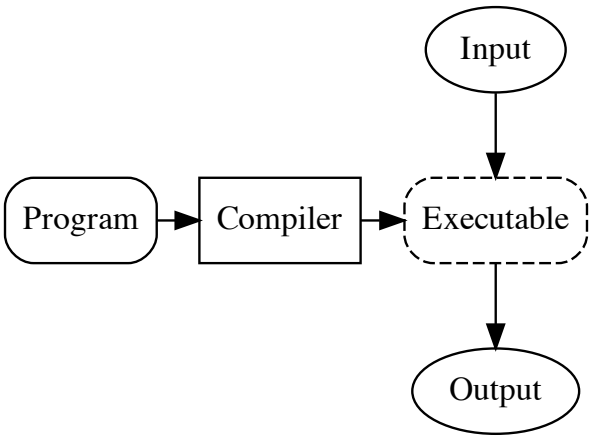

Figure 3: Compilation and execution, simplified view

Note on terminology: interpreter vs evaluator

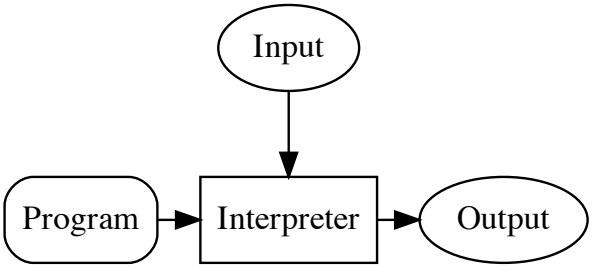

Figure 4: Typical interpreter flow

An interpreter for SAE

We have our data type for SAE. We'll build a recursive evaluator over this datatype.

What is the type of our evaluator?

```
eval :: SAE -> Integer
```

> "And evaluation/interpretation of SAE as an integer."

We need structure: What is the template for the SAE type?

```
eval (Number n) = _
eval (Add e1 e2) = _
eval (Sub e1 e2) = _
eval (Mul e1 e2) = _
eval (Div e1 e2) = _
```

We recurse over the SAE datatype.

What is the base case

Final code

```
-- <SAE> ::= <Number>
--         | <SAE> + <SAE>
--         ...

-- Abstract syntax tree for SAE
data SAE = Number Integer
         | Add SAE SAE
         | Sub SAE SAE
```

```
          | Mul SAE SAE
          | Div SAE SAE

-- Examples
sae1, sae2 :: SAE
sae1 = Sub (Add (Number 1) (Number 2)) (Number 3)
sae2 = Div (Number 44) (Add (Mul (Number 2) (Number 5)) (Number 1))

-- Evaluates an SAE by computing the corresponding integer.
eval :: SAE -> Integer
eval (Number n) = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Sub e1 e2) = eval e1 - eval e2
eval (Mul e1 e2) = eval e1 * eval e2
eval (Div e1 e2) = eval e1 `div` eval e2

test1 = eval sae1 == 0
test2 = eval sae2 == 4
```

Haskell Corner

Last time, we talked about basic types, as well as defining our own datatype using **`data`**.

Side note: I completely neglected to mention the floating point types: single precision is `Float`, double precision is `Double`.

Polymorphic types in a nutshell

What is the type of the function `length`, which computes the length of a list:

```
-- Computes the length of the given list.
length :: ? -> Int
length [] = 0
length (_ : l) = 1 + length l
```

We know it's a list? Should it be an integer list? A string list? A char list?

In Haskell, saying "a list of any type", introduces a *polymorphic type*, that is a type that can take many shapes:

```
length :: [a] -> Int
```

Here, `a` is a *type variable*, meaning "any type". Question is it the same as `Any` in the student languages?

Consider:

```
-- Take every other element from the given list
everyOther :: ?? -> ??
everyOther (_ : x : l) = x : everyOther l
everyOther _ = []
```

What should we substitute for `??` ?

The function does not care, if the argument is a list of integers or a list of strings or a list of functions from integer to string. But we do know:

a) both the input and the output is a list
b) the function preserves the type of elements, and thus the input and output lists have the same type.

We express this by using the same type variable:

```
everyOther :: [a] -> [a]
```

When we apply `everyOther` to a list of, say, integers, the type variables get instantiated wiht `Integer`, which then gives us the output type, which will also be a list of integers.

Higher-order functions

Polymorphic

- map

  ```
  map' :: (a ->b) -> [a] -> [b]
  map' _ [] = []
  map' f (x : xs) = f x : map' f xs
  ```

More in next lecture.

Type-classes

Haskell allows overloading of names. That is, the same function name can be used for different types of arguments. This is similar to Java's overriding mechanism. Somewhat confusingly, the feature that makes it possible in Haskell, is called *type classes*, but a type class is really more like Java's *interface*. A type class specifies a few function that have to be implemented for specific types. Once there is an implementation (an *instance*), we say that the type is a member of the given typeclass. We will look at how to define type classes later. Here, we will just introduce some of useful type classes.

Show

Show is a type class for types that can be pretty-printed as (converted into) strings. Many Haskell datatypes are in this class. When you type an expression in GHCi, this is what it uses to print the value of the expression. Provides the function:

```
show :: a -> String
```

Eq

Class of types that can be compared for equality and inequality using:

```
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
```

Ord

Class of types with an ordering. Provides:

```
(<) :: a -> a -> Bool
(<=) :: a -> a -> Bool
(>) :: a -> a -> Bool
(>=) :: a -> a -> Bool
min :: a -> a -> a
max :: a -> a -> a
```

Num

Class of numberic types. Provided functions:

```
(+) :: a -> a -> a
(-) :: a -> a -> a
(*) :: a -> a -> a
negate :: a -> a
abs :: a -> a
signum :: a -> a
```

Integral

Includes integral types that are also in `Num`, but additionally also allow integer division and integer remainder:

```
div :: a -> a -> a
mod :: a -> a -> a
```

- Fractional Includes non-integral members of `Num` which also allow fractional division and reciprocation:

```
(/) :: a -> a -> a
recip :: a -> a
```

Read

Includes types whose values can be read from a string (i.e., converted from a string). Provides:

```
read :: String -> a
```

The deriving mechanism

Some classes are deemed so useful and so straightforward to define, that Haskell is happy to do such definitions for us, behind the scenes. This is what the "deriving mechanism" provides. To derive the default instances for some of the above classes for our datatypes, we can simply annotate our data type definition with the **`deriving`** keyword, followed by a list of type classes:

```haskell
data Shape = Circle Float
           | Rectangle Float Float
           deriving (Eq, Ord, Show, Read)
```

Using this definition we can check equality, order shapes, print them as strings and even read them from strings. Now comparing for equality is straightforward. But what does it mean for a shape to be greater than another shape? For the same shape, we get what we expect. However what should the result of the below comparison be? Try to enter it into GHCi and see what the result it. Can you guess what rules are at play with the deriving mechanism?

```haskell
Rectangle 0.5 1 < Circle 3.14e10
```