Building an Interpreter

To follow along, check out the programming assignments project, and look into the folder called interpreter.

Building an interpreter for a programming language is one of the best ways to understand the language's precise semantics. An interpreter consists of various components. First of all, a "program" in the language comes in essentially 4 forms:

- The string of actual characters in the program file.
- The "tokens" that these characters form, like the words in a "normal" language.
- The "surface language" constructs that are essentially sentences formed out of those tokens.
- The "core language" constructs that are obtained from the surface language ones by means of a "desugaring" process. For instance the "and" of two expressions might turn into an "if-then-else" and so on.

Building an interpreter therefore consists of various components:

- A **lexer**, that takes the initial string of characters and produces the tokens. This is typically based on regular expression rules that identify the different tokens.
- A **parser**, that takes the list of tokens and forms surface language constructs out of them. Both of these steps are based on the syntax rules for the language.
- A **desugarer**, that takes the surface language construct and converts it into an equivalent core language construct.
- An **evaluator**, that takes a core language construct and evaluates it. It needs to also manage the dynamic environment and possibly other relevant information.
- If the language has static type-checking, then the **type-checker** needs to run before the evaluator in order to determine if the program properly type-checks, before evaluating it.

From our point of view the interesting part is the evaluator, and perhaps the desugarer. But we will look at all pieces. If you load the latest version of the assignments project you will find an "interpreter" folder. We will now discuss the various parts in that folder. You will get to work with the files in that folder both for your next assignment and for your final project.

Types

The “types” files are the heart of the implementation. There are two files, one named types.mli and one named types.ml.

The “mli” file is an interface file, basically implementing what we called *module signatures*. It contains definitions for the methods and types that other files need to access. More precisely, it contains the following:

- The exceptions that you may need to raise, that suggest something went wrong in the process (say you are using an identifier that doesn’t have a value yet).
- A variant type exprS for the different “surface language” expressions, the kinds of constructs that the language’s user would enter as instructions.
- A variant type exprC for the different “core language” expressions, the fundamental building blocks of the language.
- A variant type value for the kinds of values that the evaluation of an expression in the language may produce.
- An “environment” type env to use for symbol lookup, along with methods for storing something in an environment or looking something up in it.
- A desugar function that turns a surface language expression to a core language expression. You will need to implement a new branch of this function after each construct you add to the language.
- Two functions, interp and evaluate, that turn a core language expression into a value. One of the two functions expects an environment, and we will discuss the importance of this later. The most important part of the implementation of the interpreter is the code you will be adding to these expressions.
- A function valToString to be used for printing the resulting value back at the user.

The types.ml file contains implementations on all the functions described above, along with other helper functions. The majority of the implementation of a new programming language lies in the code in this file. This file essentially contains the *semantics* of your programming language. The other files are more concerned with syntax.

Parsing

Parsing turns the program from a string into an internal “surface language” representation. It is separated in two steps, the “lexer” and the “parser”. The two relevant files are lexer.mll and parser.mly. These are files with a special syntax. We will add to them as we expand the language.

You can read about these two files and their syntax at this document[1].

[1] http://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html

parser.mly is the first file to look at. It defines the series of "tokens", along with their precedence rules. It also contains a series of instructions on what to do when a specific sequence of tokens appears in a file; it is converted into a corresponding surface language construct. The file has the following parts:

- An initial section where the Types are loaded from the types.mli file.
- A section where tokens are listed. This contains for instance the following:
  - %token <float> FLOAT says that there is a token called FLOAT that contains in it a value of type float. Not all tokens need to contain information.
  - %token DBLSEMI says that there is a token called DBLSEMI that contains no extra information.
  - %nonassoc FLOAT is an instruction that suggests that the FLOAT tokens behave in a non-associative way (so we can't have say 3 of them in a row without specifying parentheses; not that it makes sense to have even two floats next to each other anyway).
- A small section where the result types are stated:
  - %start main says that the main parsing unit, which will be listed further down the file, will be the one called main.
  - %type <Types.resultS> main says that the return type of a unit called main is Types.resultS. In other words the results of our parsers are surface language expressions.
- The main part follows, where a series of "production rules" are listed, the first one being: main: | headEx DBLSEMI { $1 } ; which says that a "full expression" to be processed would consist of some series of tokens that match a headEx, specified later in the file, followed by the DBLSEMI token. Further, the $1 indicates that the value to be returned is the one that comes from the first thing in the list, namely headEx.

  After that comes: headEx: | expr { $1 } ; that specifies that a headEx constists of an expr. Finally: expr: | FLOAT { NumS $1 } ; says that one of the possibilities for an expression is for a FLOAT, and in that case we use the number stored in there, which was of type float, and put it together with the NumS constructor to form the resulting value. A lot of what we will have to do is add more cases below this one.

lexer.mll is the other file. It will define specific expressions to capture the specified tokens. It contains the following:

- An initial section that loads the Parser in order to have access to the tokens defined there, as well as define some exceptions.
- The next that follows is a series of let statements that define various regular expressions. After that the rule section specifies the order in which these regular expressions should be matched and what they should be converted to.

Driver

Finally, there is a driver.ml file that essentially puts all the other files together. You will likely not need to mess much with this file, but you can do so if you want to change the interpreter's external behavior.

Compiling

The README file contains instructions on how to "compile" all the above files into one outcome. The steps are as follows:

- ocamlyacc parser.mly processes the parser file, and produces parser.ml and parser.mli.
- ocamllex lexer.mll processes the lexer.mll file, and produces lexer.ml.
- ocamlc −c types.mli parser.mli lexer.ml parser.ml types.ml driver.ml compiles the different OCAML files, and produces output files with extensions .cmi or .cmo.
- ocamlc −o lang lexer.cmo parser.cmo types.cmo driver.cmo links the different object files together into one executable called lang.
- From this point on you can run the executable using ./lang. This starts an interactive interpreter section that will look a bit like utops, minus the fancy features.

Depending on which file you change, you may need to repeat one or more of the above steps.