Interpretation and Compilation of Programming Languages Part 2 - Syntax and Semantics

João Costa Seco

March 12, 2014

In this lecture we discuss two fundamental aspects of the programming language design: syntax (concrete and abstract) and semantics. We discuss how the syntax of a programming language can be represented in an inductive data type, hence defining programs as structured values. We proceed to define the semantics of a small programming language by means of an interpreter function, implemented by an inductive algorithm. We introduce the first programming language, comprising basic values, that we will use as running example in all entire course lectures.

1 Syntax and Semantics

Syntax and semantics are two important aspects of the design process of a programming language. A programming language is defined by combining a set of syntactic constructs, which is assigned a semantic meaning. The most important issue in programming language research is, no doubt, at the abstract level, and consists in devising constructs (abstract syntax) whose semantics are better suited to solve the problem in hands. But, when developing a programming language, targeted at effectively building systems, one has to carefully consider both the abstract and concrete syntaxes. Issues like keyword choice, operator priority, indentation sensitivity , etc., must be considered.

The syntax of a language can be studied from two complementary perspectives. The **concrete** syntax of a programming language refers to the actual (formal and precise) representation of programs, as text files, box diagrams, flow charts, or some UI builder screens. The **abstract** syntax of a programming language defines the essence and structure of each language

```
S ::= ... | if E then S else S | if E then S | ...
```

Figure 1: Ambiguous grammar for conditional statements

construct. This can immediately be mapped into data types that can be manipulated by programs (interpreters and compilers).

The syntax of a language is the first mechanism of establishing the expressiveness of a language, by allowing or limiting the use of a certain number of language constructs and their combinations. In this course we will primarily focus on text-based languages, at the concrete syntax level, but note that all concepts, techniques, and tools can be applied to other kind of languages.

2 Concrete Syntax

The concrete syntax of a programming language defines the set of artefacts accepted as syntactically correct programs of the language. The notion of a "legal" program of a given programming language is incrementally refined by other layers. Instances of such refinements can be well-typed programs, programs that satisfy a formal specification, programs that succeed on unit or integration tests, etc..

We now focus on having a precise definition of the concrete syntax for a programming language, which can be used to produce the first layer of a compiler or interpreter *front-end*. Issues like syntactic ambiguity are relevant at this stage. For instance, the code fragment in Figure 2 needs to be explicitly disambiguated. A grammar fragment for conditional statements, like the one in Figure 1, does not clearly identify which conditional statement contains the *else* branch in line 5. In languages like C or Java, the rule is to associate the `else` branch to the nearest `if` statement. If indentation is considered, in languages like Python or Haskell, the resulting derivation tree would be different.

Other examples are illustrated in the Ruby language code fragment Figure 3, where `name` can be used in distinct ways. An identifier with the form `@name` is syntactically different from an identifier with the form `name`. The former denotes an object instance variable, and the latter denotes a local variable, finally, the occurrence of `name` in a string literal, using a `#{...}` section, is in fact an occurrence of the object instance variable.

Both situations regard syntactic issues of a particular language, which we

```
1    if ( x > 0 )
2      if ( x < 10 )
3        return x;
4    else return 10;
```

Figure 2: Syntactic ambiguity in a C-like language.

```
class Hello
  def init(name)
    @name = name
  end

  def hello
    puts "Hello #{@name}!"
  end
end
```

Figure 3: Syntactic detail of use of identifiers in Ruby.

will now start to deal with.

Consider our first programming language (*VAL*), that allows basic operations on integer and boolean values. The concrete syntax of language *VAL* includes the terminal tokens that represent integer literals (*num*), and boolean literals (*true*). Recall the BNF specification language, and inspect the grammar given in Figure 4 for language *VAL*.

Our first concern is to define the language concrete syntax and corresponding parser. We first need to define all the terminals (operators and keywords), and its non-terminals, which define all the allowed structures (expressions or programs). We also need to eliminate, by design, all the ambiguities of the language. In the language above, it is necessary to carefully define the priorities and associativity of all the operators and language constructs. Consider for instance, an expression like `1 == 2 + 3`, written without parenthesis. According to the grammar in Figure 4, it has two possible derivation trees, that corresponds to either `(1 == 2) + 3`, or `1 == (2 + 3)`. The former is not the intuitive structure for the expression, the latter is clearly the intended expression. The same can be said about an expression like `1−2+3`, where the associativity of the operators is important. In this case, we consider that the correct derivation tree is given by `(1−2)+3`. We can express that kind of priority and associativity properties by redesigning the grammar as depicted in Figure 5.

```
E ::= num | E '+' E | E '-' E | E '*' E | E '/' E
    | E '<=' E | E '<' E | E '>' E | E '>=' E | E '==' E
    | 'true' | 'false' | E 'and' E | E 'or' E | 'not' E
    | 'if' E 'then' E 'else' E
    | '(' E ')'
```

Figure 4: *VAL* concrete syntax.

```
E ::= D | 'if' E 'then' E 'else' E
D ::= A | D 'or' A
A ::= C | A 'and' C
C ::= P | C '<=' P | C '<' P | C '>' P | C '>=' P | C '==' P
P ::= T | P '+' T | P '-' T
T ::= N | T '*' F | T '/' F
N ::= F | 'not' F
F ::= num | 'true' | 'false' | '(' E ')'
```

Figure 5: Rewritten grammar for *VAL* language

Notice that keywords and operators are used inside '...' , *num* denotes all integer literals, and non-terminals are given by capital letters. Terminals are accepted, and translated into a data type (token), by the lexical analyser (lexer). The language sentences, or programs, are accepted and translated into a data type by the syntactical analyser (parser). Lexing and parsing techniques are out of the scope of this course, and further reading are advised using for instance [AP02]. To bootstrap further work on lexing and parsing, refer to lab assignment 1.

3 Abstract syntax

The result of the syntactical analysis is, not only the acceptance of a program, but an abstract representation that can be manipulated and transformed. This representation is fairly independent from the actual form of the language constructs. It retains the essential characteristics of the language, by representing its structure and subcomponents of each language construct.

We call it the abstract syntax of the language, a data structure that allows the definition of an algorithm over programs. We represent it as an inductive abstract data type, a recursive and compositional definition, that allows the definition of recursive and compositional algorithms over it. Interpreters, compilers, and program verification algorithms are good examples of

```
/* ListInt.h */

typedef struct ListInt ListInt;

ListInt* nil();
ListInt* cons(int elem, ListInt *tail);
```

Figure 6: Interface do módulo `ListInt` em C

inductive definitions.

3.1 Inductive data types

An inductive data type definition, as a list of elements of type T, can be defined by a set of construction rules, like the following:

Definition 3.1 (List)**.** *A list of elements of type T, is inductively defined by:*

1. *`nil` is a list of elements of type T (the empty list)*
2. *if x is a value of type T and L is a list of elements of type T, then `cons`(x, L) is a list of elements of type T.*
3. *there are no lists of elements of type T, except the ones defined by rules* 1 *and* 2*.*

This type definition has some components that deserve to be highlighted: a name for the type (*List*), and a set of value constructors (`nil` and `cons`). The values of a list of integers, *ListInt*, can be built using the following constructors:

1. `nil` : $() \rightarrow$ *ListInt*
2. `cons` : *Integer* $\times$ *ListInt* $\rightarrow$ *ListInt*

That corresponds, for instance, to a `C` module with the interface described in Figure 6, to an OCaml module as in Figure 7, or to a Java interfaces as the one in Figure 8.

```
module type List = Sig
  type intList
  val nil:intList
  val cons: int -> intList -> intList
end
```

Figure 7: Interface of module `ListInt` in OCaml.

```
interface ListInt {...};

interface ListFactory {
  static ListInt nil();
  static ListInt cons(int elem, ListInt tail);
};
```

Figure 8: Interface of module `ListInt` in Java.

4 Abstract Syntax Trees

An abstract syntax tree is a value of a inductive data type, that represents programs in a given programming language. The derivation tree of a grammar defining a language, corresponds roughly to the structure of an abstract syntax tree.

It is the role of the syntactical analyser (parser) to transform a sequence of tokens into a value of the data type that represents the programming language. A fragment of language *VAL* with arithmetic expressions on integer numbers can be represented by the abstract data type $CALC$, and the following type constructors:

1. `num` : $Integer \rightarrow CALC$
2. `add` : $CALC \times CALC \rightarrow CALC$
3. `sub` : $CALC \times CALC \rightarrow CALC$
4. `mul` : $CALC \times CALC \rightarrow CALC$
5. `div` : $CALC \times CALC \rightarrow CALC$

This abstract data type definition corresponds to the `OCaml` sum type defined in Figure 9, and to the set of `Java` interface and classes of 10.

The abstract representation allows the direct definition of inductive algorithms, to express meanings (interpreters), properties (type systems), or code translations (compilers).

```
type calc =
    Num of int
  | Add of calc * calc
  | Sub of calc * calc
  | Mul of calc * calc
  | Div of calc * calc
```

Figure 9: OCaml data-type, of programs of language *CALC*

Exercise 1. Consider the concrete syntax of the language constructions presented next. Identify and present the necessary abstract syntax constructors and corresponding abstract syntax tree.

1. `x`
2. `0::[1;2;3]`
3. `f(0)`
4. `x = x + 1;`
5. `for(var i = 0; i < N; i++) { a += a * i; }`
6. `for(Integer i : sizes) { s += s + i; }`
7. `function (x) { return x*3; }`
8. ```
   class A {
     int a;

     A() { a = 0 }
   }
   ```

5 Structural Operational Semantics

A structural operational semantics is a possible way of defining the precise meaning of programs of a programming language. It consists in a function, defined by case analysis, that describes how the denotation of a program can be obtained from the denotations of its components. One can define several different semantic functions for a given programming language, hence obtaining different kinds of denotations for programs. Interpreters, compilers, and type systems are instances of semantic functions, of different natures, related to a programming language.
   ```

```
interface ASTNode {...}

class ASTAdd {
  ASTNode left, right;
  ...
}

class ASTSub {
  ASTNode left, right;
  ...
}

class ASTMul {
  ASTNode left, right;
  ...
}

class ASTDiv {
  ASTNode left, right;
  ...
}
```

Figure 10: Java data-type `ASTNode`, of programs of language $CALC$

Since we know how to represent a program as a value, using a data type to define the set of all possible programs, we may build and manipulate values of such types. We will use functions on programs to compute their meaning (denotation). The semantics of a language can be characterised by a function whose domain is the set of programs (e.g. $PROG$), and target set is the set of possible denotations (e.g. the set of all integer values). The function assigns a (single) denotation to each (syntactically correct) program, or program fragment.

Different kinds of interpreting functions will necessarily have different target sets (denotation sets). For instance, a type system is an interpretation function that assigns a denotation of the set of possible value types to programs (expressions). A compiler is a function whose target set are programs in an intermediate/machine language. We call operational semantics to a, syntax-directed structural definition of the semantic function that assigns value denotations to programs. The basic principle in this kind of definition, is that the semantic denotation of a construct is based on the semantic denotation of its components.

```
let rec eval e =
  match e with
    Number n -> n
  | Add(l,r) -> (eval l)+(eval r)
  | Sub(l,r) -> (eval l)-(eval r)
  | Mul(l,r) -> (eval l)*(eval r)
  | Div(l,r) -> (eval l)/(eval r)
```

Figure 11: Evaluation function for *CALC* in OCaml

```
interface ASTNode {
  int eval();
}

class ASTAdd {
  ASTNode left, right;

  int eval() {
    return left.eval() + right.eval();
  };
}
```

Figure 12: Evaluation function for *CALC* in Java.

The structural operational semantics of programming language *CALC* is here defined by a semantic function, called *eval*, that assigns an expression its value. The domain of such function is the set of arithmetic *CALC* and its target set is the set of integer values:

$$eval : CALC \rightarrow Integer$$

The function is inductively defined in the cases of the data type *CALC*, as it is the case of the OCaml code in Figure 11. The type of function `eval` is `calc->int`. This definition is compositional, which means that we can extend the language with new language constructions, by only defining their semantics alone.

The same interpretation function can be defined, using an object-oriented style, by method `eval` of the classes implementing the ASTNode interface (Figure 10), as depicted in Figure 12. Check the starter code given to learn yet another strategy to implement the `eval` function using the Visitor pattern.

Notice that the function in Figure 11 yields a result for all syntactically correct expression, except in the case of a division by zero. This is an unexpected error in our implementation, i.e. the interpreter does not detect the error and crashes. Also, the target set of the eval function is directly mapped to the `int` data type.

We will have to extend our definition to consider language *VAL* (Figure 4) where the set of valid results is $\mathit{Integer} \cup \{\texttt{true}, \texttt{false}\}$. There is an infinite number of expressions for which the result is not defined.

Exercises

Exercise 2 ($\star$). Consider the language of regular expressions, whose concrete syntax contains the terminal symbols `*`, `+`, `?`, `|`, `(`, and `)`, and is defined by the following grammar:

$$
\begin{array}{rcl}
E & ::= & E\ \langle \texttt{*} \rangle \\
 & | & E\ \langle \texttt{+} \rangle \\
 & | & E\ \langle \texttt{?} \rangle \\
 & | & E\ \langle \texttt{|} \rangle\ E \\
 & | & E\ E \\
 & | & \langle \texttt{(} \rangle\ E\ \langle \texttt{)} \rangle \\
 & | & \langle \mathit{char} \rangle
\end{array}
$$

Represent the abstract syntax of the language above using:

1. an hierarchy of Java classes.
2. an inductive data type in Ocaml

Exercise 3 ($\star\star\star$). Implement an inductive algorithm to translate a regular expression into an automaton.

Note: For the purpose of this exercise, an automaton can be characterized by a Java object or an OCaml function that recursively analyses a string given as input.

Note: Consider that each operation produces an automaton with a pair of starting and final states.

Note: An automaton for each non-terminal is obtained by combining the automatons generated from the sub-expressions (λ transitions are crucial to connect (sub)automatons).

Exercise 4 ($\star$). Write 10 different regular expressions that test the constructions of the language.

References

[AP02] A.W. Appel and J. Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2002.

A Haskell code

```
module Calc where

{- Definition of the Abstract Syntax of the language CALC -}

data CALC =
    Num Int
  | Add CALC CALC
  | Sub CALC CALC
  | Mul CALC CALC
  | Div CALC CALC
  deriving Show

{- Examples -}

a = (Num 1)

b = (Add (Num 1) (Sub (Num 3) (Num 2)))

 ---------------------------------------------------------------------------
{- Semantics                                                               -}
 ---------------------------------------------------------------------------

eval :: CALC -> Int
eval (Num n) = n
eval (Add e e') = (eval e) + (eval e')
eval (Sub e e') = (eval e) - (eval e')
eval (Mul e e') = (eval e) * (eval e')
eval (Div e e') = div (eval e) (eval e')
-- division by zero causes an exception to occur
```