Lecture 2: Abstract Syntax, BNF, Algebraic Datatypes

CS4400 Programming Languages

Language; Syntax; Concrete syntax; Abstract syntax; Inductive definitions; Grammars; BNF; Haskell types; Algebraic datatypes; Representing abstract syntax in Haskell

Readings: **EPL**, Ch. 1; **TAPL**, 3.1-3.2

Overview

In this lecture we talk about concrete vs abstract syntax and BNF, a widely used notation for specifying syntax. On the Haskell side of things, we'll look at types, type signatures, algebraic datatypes and, in particular, their use for representing abstract syntax trees.

Warning: Contains some math.

Syntax

If you took a course on formal languages, you might recall that a language is a set words (sometimes more appropriately: *expressions*, *terms*, *phrases*) over an alphabet (which is a set of symbols). Every programming language is a formal language (but not every formal language is a programming language).

However, symbols can be arbitrarily combined in many ways, but we often recognize only some combinations as part of a language.

How can we tell what belongs to a language and what doesn't?

Rules $\rightarrow$ *Syntax*

Syntax is a collection of rules that determine which strings are valid, i.e., which strings belong to a given language. In turn, syntax defines the set of valid strings, thus, defines the language.

Concrete vs Abstract Syntax

When dealing with programming language, we often talk about the concrete syntax and abstract syntax of a language.

Concrete syntax

- Simply put: what we seen when we look at a program
- All the details, such as, semicolons, parentheses, braces, brackets, precedence of operators, etc.
- E.g., the same integer can be written down in multiple ways
- Gives us the necessary information to *validate* and *parse* strings of characters as programs / phrases of a programming language

What does it consist of? How is it specified?

1. *Lexical* details

- What are the keywords
- What are the operators, separators, grouping characters
- Valid variable names
- Literals:
    - numbers (integers, reals, ...) – think of examples of number representations
    - strings
    - etc.
- Defines words of the language, also called *tokens*
- *Regular languages* are usually sufficient, most often specified using *regular expressions*
    - Examples:

    ```
    DIGIT := [0123456789]
    INTEGER := DIGIT+
    REAL := INTEGER '.' INTEGER
    ```

2. Structure

- How are tokens combined to form phrases of a language
- E.g., how are expressions built up from numbers, operators, parentheses, variable names, ...
- Specified using *context-free grammars*, sometimes context-sensitive grammars
- We'll see an example later

Abstract syntax

- Abstracts from the concrete representation
- Hides "distractions", such as, grouping of expressions and statements – you usually won't see any parentheses in an abstract syntax spec
- Captures the (abstract) *structure* of a language phrase

When discussing semantics of programming languages, we want to start from abstract syntax and forget about all the details of what the string representation looks like.

Specifying Syntax

There are multiple ways of specifying abstract syntax. Remember, that we are basically defining a set of (abstract) phrases. We have different ways of defining syntactic sets.

One is enumeration: for a language with just a few simple phrases, we could just list them all. But for more interesting examples, we use *inductive definitions*.

As an example, we will use a language of simple arithmetic expressions with integers, addition, subtraction, multiplication, and division. Let's call it SAE. E.g., - `1` is in SAE - `2 + 3` $\in$ SAE - `4 * 5 + 31` $\in$ SAE

Here are a few equivalent ways of writing an inductive definition of a syntax:

Variant 1:

1. If n is an integer, then n is in SAE (If $n \in \mathbb{Z}$ then $n \in$ SAE}
2. If e_1 and e_2 are in SAE then so are:

   (a) e_1 `+` e_2
   (b) e_1 `*` e_2
   (c) e_1 `-` e_2
   (d) e_1 `/` e_2

If we want to be pedantic, we add a clause saying "nothing is in SAE, unless it follows from the above two rules".

If we really wanted to be pedantic, we would define what is an integer in SAE

Why is `1 + 3 / 4` a SAE expression?

Because `1`, `3` and `4` are integers and so are SAE expressions using 1. Since both `1` and `3` are SAE expressions, then so is `1 + 3` using 2. Since `1 + 3` is a SAE expression and `4` is also, then so is `1 + 3 / 4` using 2.

Variant 2 (inference rules):

1. $$\frac{n \in \mathbb{Z}}{n \in \text{SAE}}$$

2.
$$\frac{e_1 \in \text{SAE} \quad e_2 \in \text{SAE}}{e_1 \texttt{ + } e_2 \in \text{SAE}} \quad \frac{e_1 \in \text{SAE} \quad e_2 \in \text{SAE}}{e_1 \texttt{ * } e_2 \in \text{SAE}} \quad \cdots$$

These are *inference rules*. They can be read as "If (aka premise) holds, then so does (aka conclusion)". This style is used heavily in some styles of semantics. We will be mainly using inference rules when we talk about types. But they might come in handy at other times to quickly write some ideas down.

Metavariables Note, that n, e_1 and e_2 are what we call *meta-variables*. They usually stand for a (restricted or arbitrary) piece of syntax. For example e_1 + e_2 matches any

Why is `1 + 3 / 4` a SAE expression?

$$\frac{\dfrac{\dfrac{1 \in \mathbb{Z}}{1 \in \text{SAE}} \quad \dfrac{3 \in \mathbb{Z}}{3 \in \text{SAE}}}{\texttt{1 + 3} \in \text{SAE}} \quad \dfrac{4 \in \mathbb{Z}}{4 \in \text{SAE}}}{\texttt{1 + 3 / 4} \in \text{SAE}}$$

This is a *derivation* (tree). We obtain it by taking a rule, substituting its meta-variables so that the conclusion contains what we are trying to show (in this case `1 + 3 / 4` $\in$ SAE with e_1 replaced by `1 + 3` and e_2 by 4), then applying further rules to the first rule's premises, keeping the substitutions consistent. We repeat this, until we reach rules that do not have premises to which rules need to be applied.

Variant 3 (Grammar in Backus-Naur Form aka BNF):

```
<SAE> ::= <INTEGER>
        | <SAE> + <SAE>
        | <SAE> * <SAE>
        | <SAE> - <SAE>
        | <SAE> / <SAE>
```

When talking about syntax we will be mostly using grammars in BNF notation like this. Here, meta-variables are things enclosed in angle-brackets: `<SAE>` and `<INTEGER>`. The other symbols (`+`, `-`, `*`, `/`, the integers) are what we call *terminals* or *terminal symbols*. They are called that, because they cannot be replaced by any other symbol: meta-variables can be replaced by an appropriate combination of meta-variables and terminals (following the grammar), but terminals are final.

Each line of the grammar is called a *production*. The `<SAE>` before `::=` is called the *head* of the production(s).

Now, why is `1 + 3 / 4` a SAE expression?

```
  <SAE> => <SAE> / <SAE>
        => <SAE> + <SAE> / <SAE>
        => <INTEGER> + <SAE> / <SAE>
        => 1 + <SAE> / <SAE>
        => 1 + <INTEGER> / <SAE>
        => 1 + 3 / <SAE>
        => 1 + 3 / <INTEGER>
        => 1 + 3 / 4
```

This is also called a derivation.

BNF nicely corresponds to algebraic data types in Haskell and we will visit this at the end of the lecture.

For comparison, here is what a concrete syntax BNF for SAE might look like in its full glory:

```
<Expression> ::= <Expression> + <Term>
               | <Expression> - <Term>
               | <Term>

<Term> ::= <Term> * <Factor>
         | <Term> / <Factor>
         | <Factor>

<Factor> ::= ( <Expression> )
           | <Integer>

<Integer> ::= <Integer> <Digit>
            | <Digit>

<Digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Haskell Corner

We have seen some basic examples of Haskell: how to write expressions, how to define basic function using basic pattern matching, quards and conditional expressions.

Today we will talk about basic types, type signatures and defining our own types. We will also look more at pattern matching and working with lists.

Types

I mentioned some basic types in Haskell: `Integer`, `Bool`. Another basic type is `Char`. There is also a second integer type called `Int`. The difference is that, whereas `Integer` is a type of integers of arbitrary length (try it out!), `Int` is the type of bounded machine integers. Unfortunately, we cannot just pick one and stick with it, since both are being used by Haskell's libraries. We might have to convert one to the other from time to time. Also, in Haskell, strings (`String`)are not a basic (primitive) type.

We talked about lists. A list type is written `[a]`, where `a` is the name of the element type. For example, `[Integer]` is the type of lists of integers. By extension, `[[Bool]]` is the type of lists of lists of booleans.

Type names (actually type *constructors*) always start with an upper-case letter.

Type Signatures

Just like you were used to in Fundies 1, we will be writing down type signatures in Haskell, along with purpose statements, *for every top level definition*. Unlike in Fundies 1, and similarly to Fundies 2, Haskell actually checks our types. In reality, we wouldn't really need to write down most signatures, as GHC is smart enough to *infer* the types for us. However, the types that we get this way are often too general, which can lead to errors and error messages that are difficult to process. Moreover, type signature serve as a documentation for our code that is moreover checked by the compiler. Bottom-line: always write type signatures.

Type signatures are similar to the ones you were writing in Fundies 1, but Haskell uses `::` instead of `:` and we use `->` to also separate arguments, not just the input and output of a function. More on why we use these arrows later. Here is a simple example:

```haskell
name :: String
name = "Ferd"

x :: Integer
x = 4

first5Primes :: [Integer] -- list of integers
first5Primes = [2, 3, 5, 7, 11]
```

Here are function examples. First in BSL:

```
;; area-of-square : Number -> Number
;; Compute the area of a square.
(define (area-of-square a) (* a a))

;; make-list : Number Number -> ListOf Number
;; Generate a list with the given element repeated n-times.
(define (make-list n x)
  (if (> n 0) (cons x (make-list (- n 1) x)) null))
```

and then the equivalent in Haskell:

```haskell
-- |Compute the area of a square.
areaOfSquare :: Integer -> Integer
areaOfSquare a = a * a

-- |Generate a list with the given element repeated n-times.
makeList :: Integer -> Integer -> [Integer]
makeList n x =
  if n > 0 then x : makeList (n - 1) x
           else []
```

Defining Types

The easiest way we can introduce a new type, is to introduce a type synonym. Although this is cheating – we are not really introducing a new type, just giving another type a new name. Type synonuyms are introduced using the **type** keyword.

```haskell
type MyNumber = Integer

type ListOfStrings = [String]
```

In fact, `String` is just a synonym – for a list of characters:

```haskell
type String = [Char]
```

Defining Algebraic Datatypes

Algebraic Datatypes (ADTs), or inductive or recursive datatypes are introduced using the **data** keyword. Cases are serparated using `|`. These can be simple enumerations:

```haskell
data Color = Red | Green | Blue

data Direction = Up
               | Down
               | Left
               | Right

data Bool = False | True
```

or they can be tagged values:

```haskell
data NumberOrName = Number Integer
                  | Name String
```

or they can be proper recursive / inductive datatype definitions:

```haskell
data MyIntegerList = Nil
                   | Cons Integer MyIntegerList
```

In these examples, the first word of each case (e.g., `Nil`, `Number`, `Red`, `Up`) is a *constructor*, implying that it *constructs* a value of the corresponding datatype. Just like type constructors, they always start with an upper-case letter. What comes after the constructors (in the `Number`, `Name` and `Cons` cases), are the types of their arguments. E.g., to construct a `Name` value of the `NumberOrName` type, I need to give it an argument of type `String`:

```haskell
Name "Zaphod Beeblebrox"
```

How does this relate to abstract syntax? Well, the above, actually, specifies the abstract syntax of integer lists:

```
  <MyIntegerList> ::= Nil
                    | Cons <Integer> <MyIntegerList>
```

See the similarity?

ADTs and Pattern Matching

Functions operating on ADTs are specified using pattern matching:

```haskell
myLength :: MyIntegerList -> Integer
myLength Nil = 0
myLength (Cons n l) = 1 + myLength l


-- we can also use the case construct:
colorToString :: Color -> String
colorToString color =
  case color of
      Red -> "red"
      Green -> "green"
      Blue -> "blue"
```

As you can see, (basic) patterns are built up from constructors and variables. A variable matches any value and the matched value is then bound to the variable inside the right-hand side of the defining equation, or on the right-hand side of the `->` arrow if we are matching using **`case`**.

If we are not interested in some value, we can use the "wildcard" pattern _, which matches anything and discards whatever it matched.

```haskell
isName :: NumberOrName -> Bool
isName (Name _) = True
isName _ = False
```

A Note on Lists

In fact, "native" Haskell lists are datatypes just like any other. They just happen to use special constructors for the nil (`[]`) and cons (infixed `:`) cases. Otherwise they behave just like other datatypes:

```haskell
length
```

Back to Abstract Syntax

Now let us translate SAE's BNF into a Haskell datatype. We cannot use the terminal symbols themselves, so we will use mnemonics: `Add`, `Sub`, `Mul`, and `Div`.

```haskell
data SAE = Number Integer  -- each case needs to have a constructor!!
         | Add SAE SAE
         | Mul SAE SAE
         | Sub SAE SAE
         | Div SAE SAE
```

For comparison, here is our original BNF definition:

```
<SAE> ::= <INTEGER>
        | <SAE> + <SAE>
        | <SAE> * <SAE>
        | <SAE> - <SAE>
        | <SAE> / <SAE>
```