

COMP3161/COMP9164 Supplementary Lecture Notes

Syntax

Gabriele Keller Johannes Åman Pohjola*

October 2, 2025

1 Concrete Syntax versus Abstract Syntax

The concrete syntax of a programming language is designed with the user/programmer in mind: it should be well structured and easy to read. The parser checks if a given program adheres to the concrete syntax and translates it into a suitable internal representation. The internal representation is usually quite different from the concrete syntax expression. To demonstrate why, let us go back to the arithmetic expressions example. Consider the following three expressions:

- $1 + 2 * 3$
- $(1) + (2) * (3)$
- $((1)) + (2 * 3)$

Syntactically, all three are different, but semantically, they denote exactly the same computation. Therefore they should ideally have the same internal representation. If we would have chosen a term representation of arithmetic expressions instead of an infix notation, we would not have had to worry about the ambiguity of the grammar nor about superfluous parentheses, and we could have defined the language with the following three simple rules:

$$\frac{i \in Int}{(\text{Num } i) \text{ } expr}$$

$$\frac{t_1 \text{ } expr \quad t_2 \text{ } expr}{(\text{Plus } t_1 \ t_2) \text{ } expr} \quad \frac{t_1 \text{ } expr \quad t_2 \text{ } expr}{(\text{Times } t_1 \ t_2) \text{ } expr}$$

and all three expressions above would correspond to the term `Plus (Num 1) (Times (Num 2) (Num 3))`. Such a term-based syntax is obviously not well suited as concrete syntax of a practical language — it would be a nightmare to write a program in this style. However, it is the appropriate format for the internal representation. A parser, therefore, has to translate expressions of the concrete syntax into terms of the abstract syntax.

2 First Order Abstract Syntax

How can we specify the translation of expressions of concrete syntax into abstract syntax terms? As we discussed previously, inference rules and judgements aren't just for properties of single objects, but also relationships between objects. Thus we use inference rules to define a relationship

$$e_1 \text{ } SExpr \longleftrightarrow e_2 \text{ } expr$$

*Just minor edits. Gabriele is the original, and main, author.

which holds if and only if the (concrete grammar) expression e_1 corresponds to (abstract grammar) expression e_2 .

We take the inference rules of $SExpr$ as basis, and add the translation for each case:

$$\frac{e_1 SExpr \longleftrightarrow e'_1 \text{ expr} \quad e_2 PExpr \longleftrightarrow e'_2 \text{ expr}}{e_1 + e_2 SExpr \longleftrightarrow (\text{Plus } e'_1 \ e'_2) \text{ expr}} \quad \frac{e PExpr \longleftrightarrow e' \text{ expr}}{e SExpr \longleftrightarrow e' \text{ expr}}$$

$$\frac{e_1 PExpr \longleftrightarrow e'_1 \text{ expr} \quad e_2 FExpr \longleftrightarrow e'_2 \text{ expr}}{e_1 * e_2 PExpr \longleftrightarrow (\text{Times } e'_1 \ e'_2) \text{ expr}} \quad \frac{e FExpr \longleftrightarrow e' \text{ expr}}{e PExpr \longleftrightarrow e' \text{ expr}}$$

$$\frac{e SExpr \longleftrightarrow e' \text{ expr}}{(e) FExpr \longleftrightarrow e' \text{ expr}} \quad \frac{i \in \text{Int}}{i FExpr \longleftrightarrow (\text{Num } i) \text{ expr}}$$

Previously, we used the inference rules to prove that an object had a certain property, e.g., that $1 + 3$ is indeed an $SExpr$. With relations, inference rules become more interesting. For example, we can view the rules as a description of how to construct a abstract syntax term for a given arithmetic expression, say $1 + 3 * 4$. In this case, we start off with the proof/derivation goal $1 + 2 * 3 SExpr \longleftrightarrow ???$. That is, the right hand side of the relation is not fixed yet. To apply the addition rule, however, it has to have the form $\text{Plus } ?? \ ??$. With every rule application, we know more about the exact form of this term, until we finally end up with $\text{Plus } (\text{Num } 1) (\text{Times } (\text{Num } 2) (\text{Num } (3)))$:

$$\frac{\frac{\frac{1 FExpr \longleftrightarrow (\text{Num } 1) \text{ expr}}{1 PExpr \longleftrightarrow (\text{Num } 1) \text{ expr}}{1 SExpr \longleftrightarrow (\text{Num } 1) \text{ expr}} \quad \frac{\frac{2 FExpr \longleftrightarrow (\text{Num } 2) \text{ expr}}{2 PExpr \longleftrightarrow (\text{Num } 2) \text{ expr}}{2 * 3 PExpr \longleftrightarrow (\text{Times } (\text{Num } 2) (\text{Num } 3)) \text{ expr}}}{1 + 2 * 3 SExpr \longleftrightarrow (\text{Plus } (\text{Num } 1) (\text{Times } (\text{Num } 2) (\text{Num } 3))) \text{ expr}}$$

The process of finding, for a given s such that $s SExpr$, a corresponding term t such that $t \text{ expr}$, with $s SExpr \longleftrightarrow t \text{ expr}$ is called *parsing*. A parser has to be complete, in the sense that for every $s SExpr$ it should find the corresponding abstract syntax term. Furthermore, it has to be unambiguous, and return for every $s SExpr$ a unique t . The inverse process is called *unparsing*. Since each of the parsing rules given above directly corresponds to a production rule of $SExpr$, it is trivial to show the completeness using rule induction.

We can use the inference rules given above in another direction, and instead of deriving a term $t \text{ expr}$ for a given s , we can start with an abstract syntax term and derive a matching arithmetic expression. This process is called *unparsing*. Unparsing is, in our example and in general, ambiguous. It is also not necessarily complete, although it is for the arithmetic expression we defined. Going one step further and converting the concrete syntax token sequence back into a string is called *pretty printing*. Pretty printing is useful, for example, to view intermediate code in a compiler, or in tools which re-format a program. Pretty printing is even more ambiguous than unparsing, since we are free to add spaces and new lines in many places. A pretty printer aims at choosing the most readable representation, which is of course a very subjective measure.

3 Higher Order Abstract Syntax

Let us extend our simple language of arithmetic expressions by introducing variables and a let-construct to bind variables to values:

```
let x = 3
in x+1

let x = 3
in let y = x+1
in x+y
```

We need to extend the set of rules which define the concrete and abstract syntax accordingly.

Concrete Syntax: A let-expression acts like a parenthesised expression, and therefore behaves like an *FExpr* for priority purposes. As with numbers, where we use *int* to denote an integer without further specifying it, we use *id* to represent an identifier:

$$\frac{e_1 \text{ SExpr} \quad e_2 \text{ SExpr}}{\text{let } id = e_1 \text{ in } e_2 \text{ FExpr}}$$

Abstract Syntax: Variables are represented by terms, similar to numbers, with the exception that the argument of the term is a string *id* which contains the name of the variable. A let-expression is represented by a term which requires three arguments: an identifier (bound variable), a term (the term the variable is bound to), and the body of the let-expression, i.e., the term in which the variable is defined. Since the first argument has to be an identifier, we can drop the *var* operator in the argument position:

$$\frac{}{(\text{Var } id) \text{ expr}} \quad \frac{(\text{Var } id) \text{ expr} \quad t_1 \text{ expr} \quad t_2 \text{ expr}}{(\text{Let } id \ t_1 \ t_2) \text{ expr}}$$

3.1 Scope of a Variable

Given an expression `let x = t1 in t2`, the variable *x* is bound in *t₂*, but not in *t₁*.

The term *t₂* is called the *scope* of *x*, and *x* is bound to the value of *t₁* everywhere in *t₂*. In particular, in the following situation where the same variable is bound twice, the outer binding is *shadowed* by the inner binding, and the value of the expression is 10.

```
let x = 3
in let x = 5
   in x + x
```

Note that the scope of a variable in a let-binding is defined differently in Haskell: *x* is bound in *t₁* and *t₂*. As a consequence, the compiler accepts an expression

```
let
  x = x+1
in x
```

even though the expression cannot be evaluated and leads to a run-time exception.

3.2 Representation of Variables

In the first order abstract syntax definition of arithmetic expressions, variables are treated just like numbers and represented by terms, although they play a special role. Higher order abstract syntax addresses this shortcoming, and provides variables and variable bindings as part of the meta-language: first order terms can either

1. be a constant (e.g, ints, strings), or
2. have the form (*Op t₁ ... t_n*), where *t₁* to *t_n* are terms,
 - (Num 4)
 - (Plus (Num 2) (Num 1))

In addition, a higher order term can

3. be a variable
4. have the form *x.t'*, meaning the variable *x* is bound in term *t'*
 - x. (Plus x (Num 1))

- $x.y$. (Plus $x y$)

A term of the form $x.t$ is called an *abstraction*. It is a term whose value depends¹ on the value of the variable x . In this respect, it is similar to a function body.

An abstraction $x.t$ is said to *bind* all occurrences of x in t . All variables of a term which are not bound at the position they occur are called *free variables* of that term. We denote the set of free variables of a term by $FV(t)$. It is inductively defined as follows:

$$\begin{aligned} FV(int) &= \{\} \\ FV(x) &= \{x\} \\ FV(o(t_1, \dots, t_n)) &= FV(t_1) \cup \dots \cup FV(t_n) \\ FV(x.t) &= FV(t) \setminus \{x\} \end{aligned}$$

For example, x is in $FV(\text{plus}(x, \text{let}(5, x.x)))$, which corresponds to the concrete syntax expression

$x + (\text{let } x = 5 \text{ in } x)$

since

$$\begin{aligned} &FV((\text{Plus } x (\text{Let } (\text{Num } 5) (x.x)))) \\ &= FV(x) \cup FV(\text{Let } (\text{Num } 5) (x.x)) \\ &= \{x\} \cup FV(5) \cup FV(x.x) \\ &= \{x\} \cup (FV(x) \setminus \{x\}) \\ &= \{x\} \cup (\{x\} \setminus \{x\}) \\ &= \{x\} \cup \{\} \end{aligned}$$

If we want to use higher order syntax, we have to change the rules for variables and let-bindings in the definition of the abstract syntax:

$$\frac{}{id \text{ expr}} \quad \frac{t_1 \text{ expr} \quad t_2 \text{ expr}}{(\text{Let } t_1 (id.t_2)) \text{ expr}}$$

and adapt the the translation accordingly:

$$\frac{}{id \text{ FExpr} \longleftrightarrow (id \text{ expr})} \quad \frac{e_1 \text{ SExpr} \longleftrightarrow t_1 \text{ expr} \quad e_2 \text{ SExpr} \longleftrightarrow t_2 \text{ expr}}{\text{let } id = e_1 \text{ in } e_2 \text{ FExpr} \longleftrightarrow (\text{Let } t_1 (id.t_2)) \text{ expr}}$$

Now, the operator **let** accepts only two arguments, one being the right hand side, the second the abstraction of the body of the body.

3.3 Substitution and α -equivalence

Consider, for example, the following two expressions:

let	let
x = 3	y = 3
in	in
x+1	y+1

They express exactly the same computation and only differ in the choice of the variable names. They are represented by the term **Let** (Num 3) (x. Plus x (Num 1)) and **Let** (Num 3) (y. Plus y (Num 1)) respectively. If two terms, as in the above example, can be made identical by renaming the variables, they are called α -equivalent, written \equiv_α . As the name suggests, \equiv_α is an equivalence relation. This means \equiv_α is

1. reflexive: for all terms t , $t \equiv_\alpha t$

¹More precisely, may depend, since it is possible that x does not actually occur in t , as in $x.1$

2. symmetric: for all terms t_1, t_2 , if $t_1 \equiv_\alpha t_2$ then $t_2 \equiv_\alpha t_1$
3. transitive: for all terms t_1, t_2 , and t_3 : if $t_1 \equiv_\alpha t_2$ and $t_2 \equiv_\alpha t_3$ then $t_1 \equiv_\alpha t_3$

If we want to determine the value of a let-expression, at some point, we have to replace the variable in the body with the value the variable is bound to. This process of replacing a variable with a value, or in general, an arbitrary term, is called *substitution*. We use the notation:

$$t[x := t']$$

to describe a term t where every free occurrence of x has been replaced by t' . We can rename the variables in a term now by replacing the variable at its binding occurrence, and substituting it wherever it occurs freely in the term:

$$x.t \equiv_\alpha y.t[x := y], \quad \text{if } y \notin FV(t)$$

We have to be careful about the choice of y , though. If we try to rename x to y in the term $x. x + y$ we do not want to end up with the term $y. y + y$, since the y in the original term is now captured, and the new term is not α -equivalent to the original term anymore. Therefore, we require that the new variable does not occur freely anywhere in the original term.

Let us now give the exact definition of substitution, first for variables:

$$\begin{aligned} x[x := y] &= y \\ z[x := y] &= z, \text{ if } x \neq z \\ (Op\ t_1 \dots t_n)[x := y] &= (Op\ t_1[x := y] \dots t_n[x := y]) \\ x.t[x := y] &= x.t \\ z.t[x := y] &= z.t[x := y] \text{ if } x \neq z, y \neq z, \\ y.t[x := y] &= \text{undefined if } x \neq y \end{aligned}$$

To avoid the problem of capturing, we require that the variable which is introduced does not occur anywhere in the binding position of a term. Similarly, if we substitute terms, we require that none of the free variables in the new term occurs at a binding position in the original term:

$$\begin{aligned} x[x := u] &= u \\ z[x := u] &= z, \text{ if } x \neq z \\ (Op\ t_1 \dots t_n)[x := u] &= (Op\ t_1[u := x] \dots t_n[u := x]) \\ x.t[x := u] &= x.t \\ z.t[x := u] &= z.t[u := x] \text{ if } x \neq z, z \notin FV(u), \\ y.t[x := u] &= \text{undefined if } y \in FV(u) \end{aligned}$$

In practice, it does not matter that substitution is only partially defined, because we can always rename the variable such that clashes do not occur. Many compilers will actually rename all the variables defined by the user and map them to distinct names to simplify further compilation steps. When convenient, we will silently assume that the programs we are dealing with have distinct names.