

Syntax

Thomas Sewell
UNSW
Term 3 2024

Concrete Syntax

Arithmetic Expressions

$$\begin{array}{c}
 \frac{i \in \mathbb{Z}}{i \text{ Atom}} \quad \frac{a \text{ SExp}}{(a) \text{ Atom}} \quad \frac{e \text{ Atom}}{e \text{ PExp}} \quad \frac{e \text{ PExp}}{e \text{ SExp}} \\
 \frac{a \text{ Atom} \quad b \text{ PExp}}{a \times b \text{ PExp}} \quad \frac{a \text{ PExp} \quad b \text{ SExp}}{a + b \text{ SExp}}
 \end{array}$$

All the syntax we have seen so far is *concrete syntax*. Concrete syntax is described by judgements on *strings*.

Abstract Syntax

Working with concrete syntax directly is *unsuitable* for both compiler implementation and proofs. Consider:

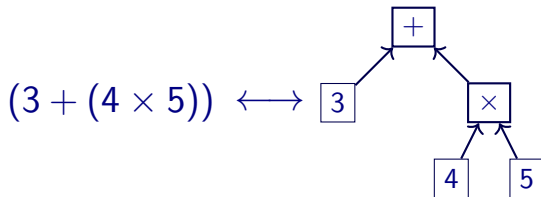
- $3 + (4 \times 5)$
- $3 + 4 \times 5$
- $(3 + (4 \times 5))$

TIMTOWTDI¹ makes life harder for us. Different derivations represent the same semantic program. We would like a representation of programs that is as simple as possible, removing any extraneous information. Such a representation is called *abstract syntax*.

¹“There is more than one way to do it”.

Abstract Syntax

Typically, the *abstract syntax* of a program is represented as a *tree* rather than as a string.



Writing trees in our inference rules would become unwieldy. We shall define a *term* language in which to express trees.

Terms

Definition

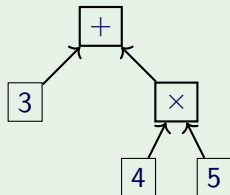
In this course, a *term* is a structure that can either be a *symbol*, like *Plus* or *Times* or *3*; or a *compound*, which consists of an *symbol* followed by one or more *argument subterms*, all in parentheses.

$$t ::= \text{Symbol} \mid (\text{Symbol } t_1 \ t_2 \ \dots)$$

These particular terms are also known as *s-expressions*. Terms can equivalently be thought of a subset of *Haskell* where the only kinds of expressions allowed are literals and data constructors.

Term Examples

Example



(Plus (Num 3) (Times (Num 4) (Num 5)))

Armed with an appropriate Haskell data declaration, this can be implemented straightforwardly:

```
data Exp = Plus Exp Exp
         | Times Exp Exp
         | Num Int
```

Concrete to Abstract

Concrete Syntax

$$\begin{array}{c}
 \frac{i \in \mathbb{Z}}{i \text{ Atom}} \quad \frac{a \text{ SExp}}{(a) \text{ Atom}} \quad \frac{e \text{ Atom}}{e \text{ PExp}} \quad \frac{e \text{ PExp}}{e \text{ SExp}} \\
 \frac{a \text{ Atom} \quad b \text{ PExp}}{a \times b \text{ PExp}} \quad \frac{a \text{ PExp} \quad b \text{ SExp}}{a + b \text{ SExp}}
 \end{array}$$

Abstract Syntax

$$\frac{i \in \mathbb{Z}}{(\text{Num } i) \text{ AST}} \quad \frac{a \text{ AST} \quad b \text{ AST}}{(\text{Plus } a \ b) \text{ AST}} \quad \frac{a \text{ AST} \quad b \text{ AST}}{(\text{Times } a \ b) \text{ AST}}$$

Now we have to specify a *relation* to connect the two!

Relations

Up until now, most judgements we have used have been *unary* — corresponding to a set of satisfying objects.

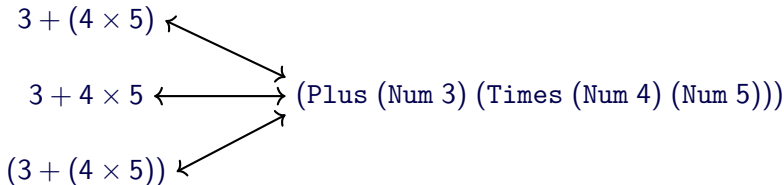
A judgement can also express a relationship between *two objects* (a *binary* judgement) or a *number of objects* (an *n-ary* judgement).

Example (Relations)

- 4 *divides* 16 (binary)
- mail *is an anagram of* liam (binary)
- 3 *plus* 5 *equals* 8 (ternary)

n-ary judgements where $n \geq 2$ are sometimes called *relations*, and correspond to an *n*-tuple of satisfying objects.

Parsing Relation



$$i \in \mathbb{Z}$$

$$\frac{i \in \mathbb{Z}}{i \text{ Atom} \longleftrightarrow (\text{Num } i) \text{ AST}}$$

$$\frac{a \text{ Atom} \longleftrightarrow a' \text{ AST} \quad b \text{ PExp} \longleftrightarrow b' \text{ AST}}{a \times b \text{ PExp} \longleftrightarrow (\text{Times } a' b') \text{ AST}}$$

$$\frac{a \text{ PExp} \longleftrightarrow a' \text{ AST} \quad b \text{ SExp} \longleftrightarrow b' \text{ AST}}{a + b \text{ SExp} \longleftrightarrow (\text{Plus } a' b') \text{ AST}}$$

$$\frac{e \text{ SExp} \longleftrightarrow a' \text{ AST} \quad e \text{ Atom} \longleftrightarrow a \text{ AST} \quad e \text{ PExp} \longleftrightarrow a \text{ AST}}{(e) \text{ Atom} \longleftrightarrow a' \text{ AST} \quad e \text{ PExp} \longleftrightarrow a \text{ AST} \quad e \text{ SExp} \longleftrightarrow a \text{ AST}}$$

$$\frac{e \text{ SExp} \longleftrightarrow a' \text{ AST}}{(e) \text{ Atom} \longleftrightarrow a' \text{ AST}} \quad \frac{e \text{ Atom} \longleftrightarrow a \text{ AST}}{e \text{ PExp} \longleftrightarrow a \text{ AST}} \quad \frac{e \text{ PExp} \longleftrightarrow a \text{ AST}}{e \text{ SExp} \longleftrightarrow a \text{ AST}}$$

Relations as Algorithms

The *parsing relation* \longleftrightarrow is an extension of our existing concrete syntax rules. Therefore it is **unambiguous**, just as those rules are. Furthermore, the abstract syntax can be **unambiguously** determined **solely** by looking at the left hand side of \longleftrightarrow .

An Algorithm

To determine the term corresponding to a particular string:

- 1 Derive the left hand side of the \longleftrightarrow (the concrete syntax) **bottom-up** until reaching axioms.
- 2 Fill in the right hand side of the \longleftrightarrow (the abstract syntax) **top-down**, starting at the axioms.

This process is called *parsing*.

Example

Rules

$$\begin{array}{c}
 \frac{i \in \mathbb{Z}}{i \mathbf{A} \longleftrightarrow (\text{Num } i)} \quad \frac{a \mathbf{S} \longleftrightarrow a'}{(a) \mathbf{A} \longleftrightarrow a'} \quad \frac{e \mathbf{A} \longleftrightarrow a}{e \mathbf{P} \longleftrightarrow a} \quad \frac{e \mathbf{P} \longleftrightarrow a}{e \mathbf{S} \longleftrightarrow a} \\
 \\
 \frac{a \mathbf{A} \longleftrightarrow a' \quad b \mathbf{P} \longleftrightarrow b'}{a \times b \mathbf{P} \longleftrightarrow (\text{Times } a' b')} \quad \frac{a \mathbf{P} \longleftrightarrow a' \quad b \mathbf{S} \longleftrightarrow b'}{a + b \mathbf{S} \longleftrightarrow (\text{Plus } a' b')}
 \end{array}$$

$$\begin{array}{c}
 \frac{1 \mathbf{A} \longleftrightarrow (\text{Num } 1) \mathbf{AST}}{1 \mathbf{P} \longleftrightarrow (\text{Num } 1) \mathbf{AST}} \quad \frac{\frac{2 \mathbf{A} \longleftrightarrow (\text{Num } 2) \mathbf{AST}}{2 \times 3 \mathbf{P} \longleftrightarrow (\text{Times } (\text{Num } 2) (\text{Num } 3)) \mathbf{AST}} \quad \frac{3 \mathbf{A} \longleftrightarrow (\text{Num } 3) \mathbf{AST}}{3 \mathbf{P} \longleftrightarrow (\text{Num } 3) \mathbf{AST}}}{2 \times 3 \mathbf{S} \longleftrightarrow (\text{Times } (\text{Num } 2) (\text{Num } 3)) \mathbf{AST}} \\
 \hline
 1 + 2 \times 3 \mathbf{S} \longleftrightarrow (\text{Plus } (\text{Num } 1) (\text{Times } (\text{Num } 2) (\text{Num } 3))) \mathbf{AST}
 \end{array}$$

The Inverse

What about the inverse operation to **parsing**?

Unparsing

Unparsing, also called *pretty-printing*, is the process of starting with the **term** on the right hand side of \longleftrightarrow and attempting to synthesise a string on the left.

Problem

There are **many** concrete strings for a given abstract syntax term. The algorithm is *non-deterministic*.

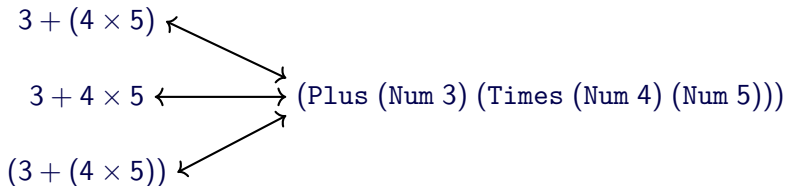
While it is desirable to have:

$$parse \circ unparsed = id$$

It is not usually true that:

$$unparse \circ parse = id$$

Example



Going from **right to left** requires some formatting guesswork to produce readable code.

Algorithms to do this can get quite involved!

Let's implement a parser for arithmetic. to coding

Adding Let

Let us extend our arithmetic expression language with **variables**, including a `let` construct to give them values.

Concrete Syntax

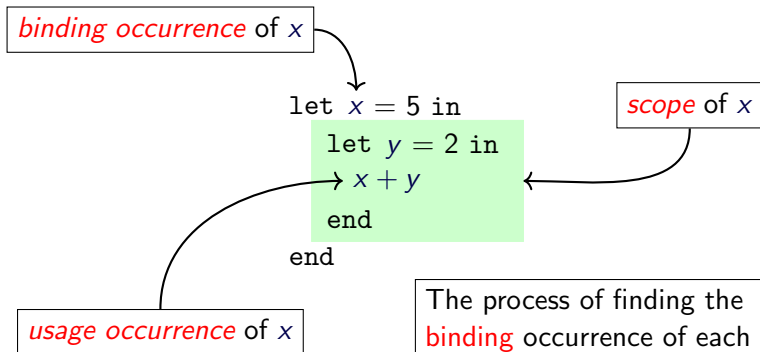
$$\frac{x \text{ Ident}}{x \text{ Atom}} \quad \frac{x \text{ Ident} \quad e_1 \text{ SExp} \quad e_2 \text{ SExp}}{\text{let } x = e_1 \text{ in } e_2 \text{ end Atom}}$$

Example

```
let x = 3 in
  x + 4
end
```

```
let x = 3 in
  let y = 4 in x + y end
end
```

Scope



The process of finding the *binding* occurrence of each used variable is called *scope resolution*. Usually this is done *statically*. If no binding can be found, an *out of scope* error is raised.

Shadowing

What does this program evaluate to?

```
let x = 5 in
  let x = 2 in
    x + x
  end
+ x end
```

x is *shadowed* here

This program results in 9.

α -equivalence

What is the difference between these two programs?

let $x = 5$ in	let $a = 5$ in
let $x = 2$ in	let $y = 2$ in
$x + x$	$y + y$
end	end
end	end

They are **semantically** identical, but differ in the choice of **bound variable names**. Such expressions are called **α -equivalent**.

We write $e_1 \equiv_{\alpha} e_2$ if e_1 is α -equivalent to e_2 . The relation \equiv_{α} is an **equivalence relation**. That is, it is **reflexive**, **transitive** and **symmetric**.

The process of consistently renaming variables that preserves α -equivalence is called **α -renaming**.

Substitution

A variable x is *free* in an expression e if x occurs in e but is not bound in e .

Example (Free Variables)

The variable x is free in $x + 1$, but not in `let $x = 3$ in $x + 1$ end.`

A *substitution*, written $e[x := t]$ (or $e[t/x]$ in some other courses), is the replacement of all *free* occurrences of x in e with the term t .

Example (Simple Substitution)

$(5 \times x + 7)[x := y \times 4]$ is the same as $(5 \times (y \times 4) + 7)$.

Problems with substitution

Consider these two α -equivalent expressions.

`let $y = 5$ in $y \times x + 7$ end`

and

`let $z = 5$ in $z \times x + 7$ end`

What happens if you apply the substitution $[x := y \times 3]$ to both expressions? You get two **non- α -equivalent** expressions!

`let $y = 5$ in $y \times (y \times 3) + 7$ end`

and

`let $z = 5$ in $z \times (y \times 3) + 7$ end`

This problem is called **capture**.

Variable Capture

Capture can occur for a substitution $e[x := t]$ when a bound variable in e clashes with a free variable occurring in t .

Fortunately

It is **always possible** to avoid capture.

- **α -rename** the offending bound variable to an unused name, or
- If you have access to the free variable's definition, renaming the free variable, or
- Use a **different abstract syntax representation** that makes capture impossible (More on this later).

Abstract Syntax for Variables

We shall extend our AST and parsing relation to include a definition for `let` and variables.

Let Syntax

$$\begin{array}{c}
 \frac{x \text{ Ident}}{x \text{ Atom} \longleftrightarrow (\text{Var } x) \text{ AST}} \\
 \frac{x \text{ Ident} \quad e_1 \text{ SExp} \longleftrightarrow a_1 \text{ AST} \quad e_2 \text{ SExp} \longleftrightarrow a_2 \text{ AST}}{\text{let } x = e_1 \text{ in } e_2 \text{ end Atom} \longleftrightarrow (\text{Let } x \ a_1 \ a_2) \text{ AST}}
 \end{array}$$

First Order Abstract Syntax

Consider the following two pieces of abstract syntax:

```
(Let "x" (Num 5) (Plus (Num 4) (Var "x")))
```

```
(Let "y" (Num 5) (Plus (Num 4) (Var "y")))
```

This demonstrates some problems with our abstract syntax approach.

- 1 Substitution capture is a problem.
- 2 α -equivalent expressions are **not equal**. Determining if an expression is α -equivalent requires us to search for a consistent α -renaming of variables.
- 3 Avoiding capture or scope errors requires computing free-variable sets and this has to be implemented for every part of our language.

de Bruijn Indices


One popular approach is *de Bruijn indices*.

Key Idea

- 1 Remove all identifiers from binding expressions like Let.
- 2 Replace the identifier in a Var with a number indicating how many binders we must skip in order to find the binder for that variable.

```
(Let "a" (Num 5)
 (Let "y" (Num 2)
  (Plus (Var "a") (Var "y")))))
```

```
(Let (Num 5)
 (Let (Num 2)
  (Plus (Var 1) (Var 0)))))
```



Debruijnification

Algorithm

Given a piece of *first order abstract syntax* with explicit variable names, we can convert to de Bruijn indices by keeping a *stack* of variable names, pushing onto the stack at each Let and popping after the variable goes out of scope. When a usage occurrence is encountered, replace the variable name with its *first position* in the stack (starting at the top of the stack).

This approach naturally eliminates *shadowing*. It is always possible to name every variable that is in scope.

It's also possible, but harder, to have de Bruijn indices going in the other direction (from the bottom of the stack, upwards).

de Bruijn Substitution

Substitution can be made **capture avoiding** without considering free-variable sets.

$$\begin{aligned}
 (\text{Num } i)[n := t] &= (\text{Num } i) \\
 (\text{Plus } a \ b)[n := t] &= (\text{Plus } a[n := t] \ b[n := t]) \\
 (\text{Times } a \ b)[n := t] &= (\text{Times } a[n := t] \ b[n := t]) \\
 (\text{Var } m)[n := t] &= \begin{cases} t & \text{if } n = m \\ (\text{Var } (m - 1)) & \text{if } m > n \\ (\text{Var } m) & \text{otherwise} \end{cases} \\
 (\text{Let } e_1 \ e_2)[n := t] &= (\text{Let } e_1[n := t] \ e_2[n + 1 := t_{\uparrow 0}])
 \end{aligned}$$

Where $e_{\uparrow n}$ is an **up-shifting** operation defined as follows:

$$\begin{aligned}
 (\text{Num } i)_{\uparrow n} &= (\text{Num } i) \\
 (\text{Plus } a \ b)_{\uparrow n} &= (\text{Plus } a_{\uparrow n} \ b_{\uparrow n}) \\
 (\text{Times } a \ b)_{\uparrow n} &= (\text{Times } a_{\uparrow n} \ b_{\uparrow n}) \\
 (\text{Var } m)_{\uparrow n} &= \begin{cases} (\text{Var } (m + 1)) & \text{if } m \geq n \\ (\text{Var } m) & \text{otherwise} \end{cases} \\
 (\text{Let } e_1 \ e_2)_{\uparrow n} &= (\text{Let } e_{1\uparrow n} \ e_{2\uparrow n+1})
 \end{aligned}$$

Examining de Bruijn indices

How do de Bruijn indices stack up against explicit names?

- 1 Substitution capture **solved**.
- 2 α -equivalent expressions are now **equal**.
- 3 We still must define substitution machinery **by hand** for each type of expression.
- 4 It is still possible to make **malformed syntax** – indices that overflow the stack, for example.
- 5 Naming the correct variable changes from a **rare** issue to a **standard** one.

Two out of four isn't bad, but can we do better by changing the **term language**?

Higher Order Terms

We shall change our term language to include **built-in** notions of variables and binding.

t	$::=$	Symbol	(symbols)
		x	(variables)
		$t_1 t_2$	(application)
		$x. t$	(binding or abstraction)

As in Haskell, we shall say that application is **left-associative**, so

$$(\text{Plus } (\text{Num } 3) (\text{Num } 4)) = ((\text{Plus } (\text{Num } 3)) (\text{Num } 4))$$

Now the **binding** and **usage** occurrences of variables are distinguished from regular symbols in our term language. Let's see what this lets us do...

Representing Let

$$\frac{a_1 \text{ AST} \quad a_2 \text{ AST}}{(\text{Let } a_1 (x. a_2)) \text{ AST}}$$

We no longer need a rule for variables, because they're baked into the structure of terms.

How would we represent this AST in **Haskell**?

```
data AST = Num Int
         | Plus AST AST
         | Times AST AST
         | Let AST ???(AST → AST)
```

So `let x = 3 in x + 2 end` becomes, in Haskell:

```
(Let (Num 3) (λx → Plus x (Num 2)))
```

Substitution

We can now define substitution across **all** terms **in the meta-logic**:

$$\begin{aligned}\text{Symbol}[x := e] &= \text{Symbol} \\ y[x := e] &= \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases} \\ (t_1 \ t_2)[x := e] &= t_1[x := e] \ t_2[x := e] \\ (y. \ t)[x := e] &= \begin{cases} (y. \ t) & \text{if } x = y \\ (y. \ t[x := e]) & \text{if } y \notin \text{FV}(e) \\ \text{undefined} & \text{otherwise} \end{cases}\end{aligned}$$

Where $\text{FV}(\cdot)$ is the set of all **free variables** in a term:

$$\begin{aligned}\text{FV}(\text{Symbol}) &= \emptyset \\ \text{FV}(x) &= \{x\} \\ \text{FV}(t_1 \ t_2) &= \text{FV}(t_1) \cup \text{FV}(t_2) \\ \text{FV}(x. \ t) &= \text{FV}(t) \setminus \{x\}\end{aligned}$$

Cheating Outrageously

Substitution **capture** is still a problem in HOAS. But it is **not our problem**. Because substitution is defined in the **meta-language**, it's the job of the implementors of the meta-language (if any) to deal with capture issues.

- When **Haskell** is our meta-language, it's the job of the GHC developers.
- When we are doing proofs in our **meta-logic**, there is no implementation, so we can just say that we assume α -equivalent terms to be equal, and therefore assume that variables are always renamed to avoid capture.

So, we have solved the problem by making it someone else's problem. **Outrageous cheating!**

Evaluating All Approaches

	HOAS		FOAS	
	Proofs	Haskell	Strings	de Bruijn
Capture	Cheat	Cheat	Problem	Solved
α -equivalence	Cheat	Cheat	Problem	Solved
Generic subst.	Solved	Solved	Problem	Problem
Malformed syntax	Cheat	Cheat	Problem	Problem

- In **embedded languages** and in pen and paper **proofs**, HOAS is very common.
- In conventional language implementations and machine-checked formalisations, de Bruijn indices are more popular.
- In your assignments, strings will be used 😊

More Approaches

There are yet more approaches to naming in the research literature.

- Locally nameless: use de Bruijn indices for local variables, but a name-carrying representation for free variables.
- Nominative sets: build on permutations of name sets rather than updates of name sets, to get a theory with nicer properties.