Syntax

Liam O'Connor

CSE, UNSW (and data61)

Semester 2 2018

Concrete Syntax

Arithmetic Expressions

$$\frac{i \in \mathbb{Z}}{i\ \textbf{Atom}} \qquad \frac{a\ \textbf{SExp}}{(a)\ \textbf{Atom}} \qquad \frac{e\ \textbf{Atom}}{e\ \textbf{PExp}} \qquad \frac{e\ \textbf{PExp}}{e\ \textbf{SExp}}$$

$$\frac{a\ \textbf{Atom} \qquad b\ \textbf{PExp}}{a \times b\ \textbf{PExp}} \qquad \frac{a\ \textbf{PExp} \qquad b\ \textbf{SExp}}{a + b\ \textbf{SExp}}$$

All the syntax we have seen so far is *concrete syntax*. Concrete syntax is described by judgements on strings, which describe the actual text input by the programmer.

Abstract Syntax

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

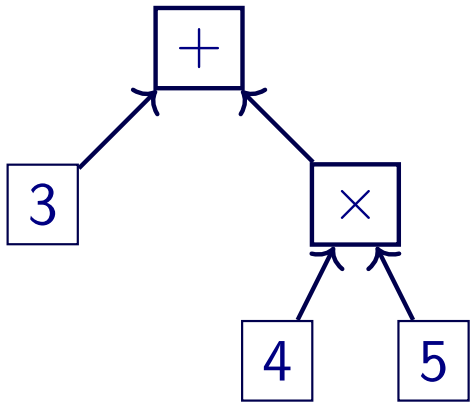
- $3 + (4 \times 5)$
- $3 + 4 \times 5$
- $(3 + (4 \times 5))$

TIMTOWTDI[1] 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.

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

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

Terms

Definition

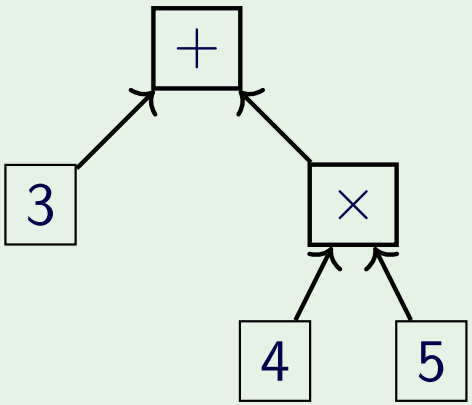
In this course, a *term* is a structure that can either be a symbol, like `Plus` or `Times` or `3`; or a compound, which consists of an symbol followed by one or more argument subterms, all in parentheses.

$$t ::= \texttt{Symbol} \mid \texttt{(Symbol } t_1 \ t_2 \ \ldots \texttt{)}$$

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:

$$\begin{array}{lcl} \textbf{data}\ \mathit{Exp} & = & \texttt{Plus}\ \mathit{Exp}\ \mathit{Exp} \\ & \mid & \texttt{Times}\ \mathit{Exp}\ \mathit{Exp} \\ & \mid & \texttt{Num}\ \mathit{Int} \end{array}$$

Concrete to Abstract

Concrete Syntax

$$\frac{i \in \mathbb{Z}}{i\ \textbf{Atom}} \qquad \frac{a\ \textbf{SExp}}{(a)\ \textbf{Atom}} \qquad \frac{e\ \textbf{Atom}}{e\ \textbf{PExp}} \qquad \frac{e\ \textbf{PExp}}{e\ \textbf{SExp}}$$

$$\frac{a\ \textbf{Atom} \qquad b\ \textbf{PExp}}{a \times b\ \textbf{PExp}} \qquad \frac{a\ \textbf{PExp} \qquad b\ \textbf{SExp}}{a + b\ \textbf{SExp}}$$

Abstract Syntax

$$\frac{i \in \mathbb{Z}}{(\texttt{Num}\ i)\ \textbf{AST}} \qquad \frac{a\ \textbf{AST} \qquad b\ \textbf{AST}}{(\texttt{Plus}\ a\ b)\ \textbf{AST}} \qquad \frac{a\ \textbf{AST} \qquad b\ \textbf{AST}}{(\texttt{Times}\ a\ b)\ \textbf{AST}}$$

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

Relations

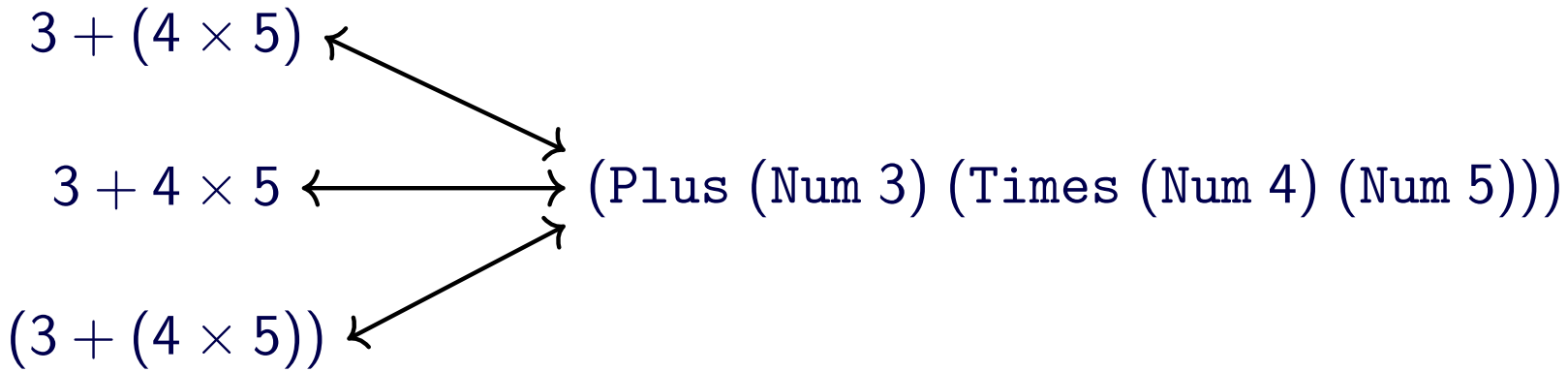
Up until now, most judgements we have used have been *unary* — corresponding to a set of satisfying objects.
It's also possible for a judgement to express a relationship between two objects (a *binary* judgement) or a number of objects (an *n-ary* judgement).

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



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

$$\frac{a\ \textbf{Atom} \longleftrightarrow a'\ \textbf{AST} \qquad b\ \textbf{PExp} \longleftrightarrow b'\ \textbf{AST}}{a \times b\ \textbf{PExp} \longleftrightarrow (\texttt{Times}\ a'\ b')\ \textbf{AST}}$$

$$\frac{a\ \textbf{PExp} \longleftrightarrow a'\ \textbf{AST} \qquad b\ \textbf{SExp} \longleftrightarrow b'\ \textbf{AST}}{a + b\ \textbf{SExp} \longleftrightarrow (\texttt{Plus}\ a'\ b')\ \textbf{AST}}$$

$$\frac{a\ \textbf{SExp} \longleftrightarrow a\ \textbf{AST}}{(a)\ \textbf{Atom} \longleftrightarrow a\ \textbf{AST}} \qquad \frac{e\ \textbf{Atom} \longleftrightarrow a\ \textbf{AST}}{e\ \textbf{PExp} \longleftrightarrow a\ \textbf{AST}} \qquad \frac{e\ \textbf{PExp} \longleftrightarrow a\ \textbf{AST}}{e\ \textbf{SExp} \longleftrightarrow a\ \textbf{AST}}$$

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 for a particular concrete syntax can be unambiguously determined solely by looking at the left hand side of $\longleftrightarrow$.

An Algorithm

To determine the abstract syntax corresponding to a particular concrete syntax:

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 of converting concrete to abstract syntax is called *parsing*.

Example

Rules

$$\frac{i \in \mathbb{Z}}{i\ \mathbf{A} \longleftrightarrow (\texttt{Num}\ i)} \qquad \frac{a\ \mathbf{S} \longleftrightarrow a'}{(a)\ \mathbf{A} \longleftrightarrow a'} \qquad \frac{e\ \mathbf{A} \longleftrightarrow a}{e\ \mathbf{P} \longleftrightarrow a} \qquad \frac{e\ \mathbf{P} \longleftrightarrow a}{e\ \mathbf{S} \longleftrightarrow a}$$

$$\frac{a\ \mathbf{A} \longleftrightarrow a' \qquad b\ \mathbf{P} \longleftrightarrow b'}{a \times b\ \mathbf{P} \longleftrightarrow (\texttt{Times}\ a'\ b')} \qquad \frac{a\ \mathbf{P} \longleftrightarrow a' \qquad b\ \mathbf{S} \longleftrightarrow b'}{a + b\ \mathbf{S} \longleftrightarrow (\texttt{Plus}\ a'\ b')}$$

$$\dfrac{\dfrac{\dfrac{}{1\ \mathbf{A} \longleftrightarrow (\texttt{Num}\ 1)\ \mathbf{AST}}}{1\ \mathbf{P} \longleftrightarrow (\texttt{Num}\ 1)\ \mathbf{AST}} \qquad \dfrac{\dfrac{}{2\ \mathbf{A} \longleftrightarrow (\texttt{Num}\ 2)\ \mathbf{AST}} \qquad \dfrac{\dfrac{}{3\ \mathbf{A} \longleftrightarrow (\texttt{Num}\ 3)\ \mathbf{AST}}}{3\ \mathbf{P} \longleftrightarrow (\texttt{Num}\ 3)\ \mathbf{AST}}}{2 \times 3\ \mathbf{P} \longleftrightarrow (\texttt{Times}\ (\texttt{Num}\ 2)\ (\texttt{Num}\ 3))\ \mathbf{AST}}}{1 + 2 \times 3\ \mathbf{S} \longleftrightarrow (\texttt{Plus}\ (\texttt{Num}\ 1)\ (\texttt{Times}\ (\texttt{Num}\ 2)\ (\texttt{Num}\ 3)))\ \mathbf{AST}}$$

The Inverse

What about the inverse operation to parsing?

Unparsing

Unparsing, also called *pretty-printing*, is the process of starting with the abstract syntax on the right hand side of the parsing relation $\longleftrightarrow$ and attempting to synthesise a concrete syntax on the left.

Problem

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

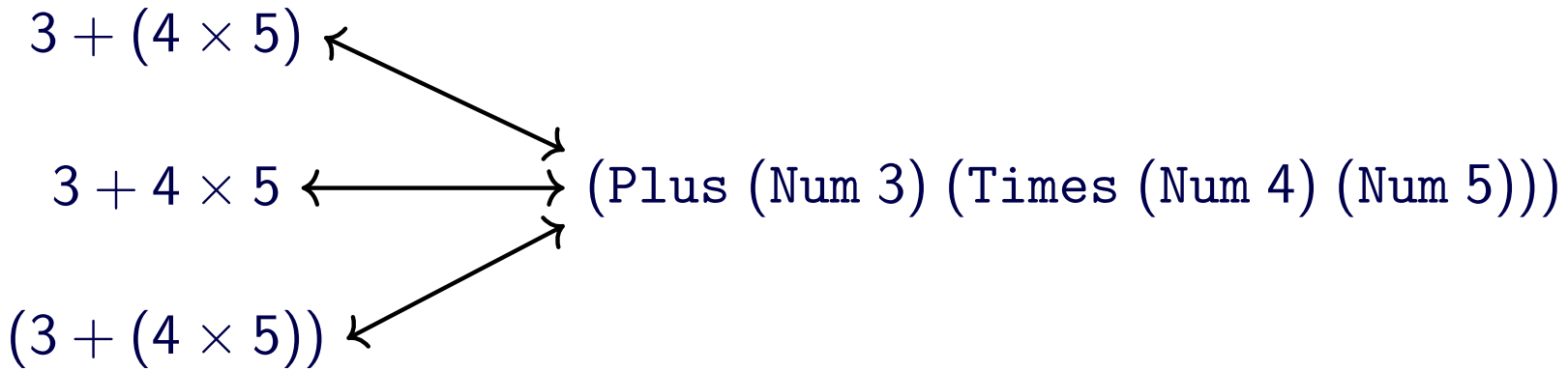
While it is desirable to have:

$$parse \circ unparse = id$$

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

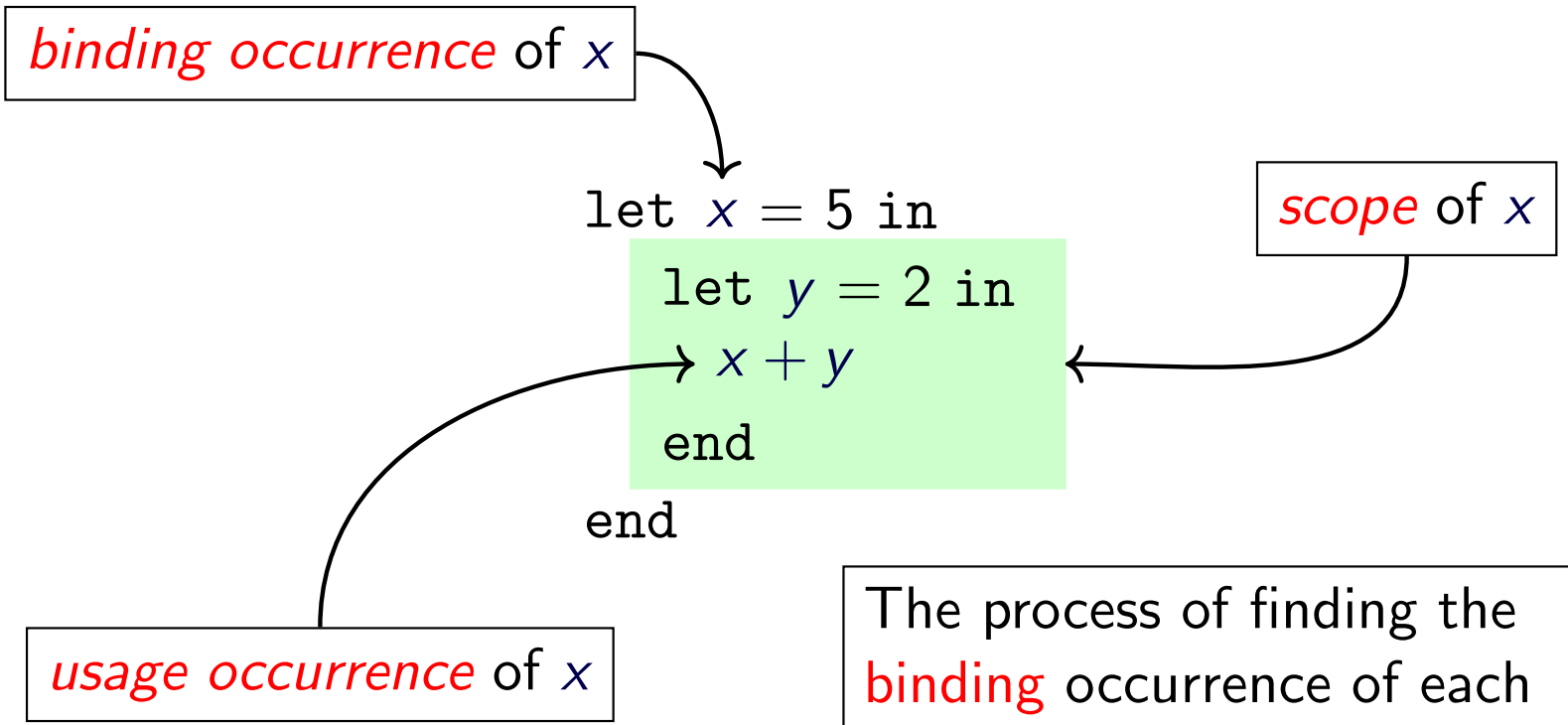
$$\frac{x\ \textbf{Ident}}{x\ \textbf{Atom}} \qquad \frac{x\ \textbf{Ident} \qquad e_1\ \textbf{SExp} \qquad e_2\ \textbf{SExp}}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end}\ \textbf{Atom}}$$

Example

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

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

Scope

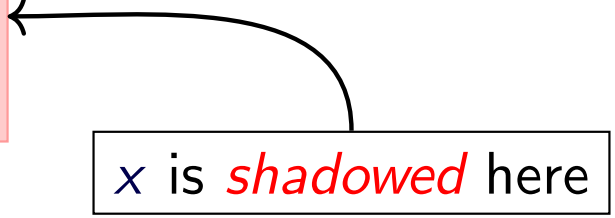


The process of finding the binding occurrence of each used variable is called *scope resolution*. Usually this is done statically. If no binding can be found, an *out of scope* error is raised.

Shadowing

What does this program evaluate to?

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



This program results in 4.

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 $\equiv_\alpha$ 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 $\texttt{let } x = 3 \texttt{ in } x + 1 \texttt{ end}$.

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.

$$\texttt{let } y = 5 \texttt{ in } y \times x + 7 \texttt{ end}$$

and

$$\texttt{let } z = 5 \texttt{ in } z \times x + 7 \texttt{ end}$$

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

$$\texttt{let } y = 5 \texttt{ in } y \times (y \times 3) + 7 \texttt{ end}$$

and

$$\texttt{let } z = 5 \texttt{ in } z \times (y \times 3) + 7 \texttt{ end}$$

This problem is called *capture*.

Variable Capture

Capture can occur for a substitution $e[x := t]$ whenever there is a bound variable in the expression e with the same name as a free variable occuring 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 next lecture).

To whiteboard Let's define *capture-avoiding substitution* for our **AST** abstract syntax.