Concepts of Programming Languages

# Syntax

Lecturer: Gabriele Keller
Tutor: Liam O'Connor
University of New South Wales
School of Computer Sciences & Engineering
Sydney, Australia

COMP 3161/9161 Week 3

UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

# Overview

- So far
  - judgements and inference rules
  - rule induction
  - grammars specified using inference rules
- This week
  - relations and inference rules
  - first-order abstract syntax
  - higher-order abstract syntax
  - substitution

# Judgements revisited

- A judgement states that a certain property holds for a specific object (which corresponds to a set membership)
- More generally, judgements express a relationship between a number of objects (*n*-ary relations)
- Examples:
  - *4 divides 16* (binary relationship)
  - `ail` *is a substring of* `mail` (binary)
  - *3 plus 5 equals 8* (tertiary)
- A *n*-ary relation implicitly defines sets of *n*-tuples
  - divides: {*(2, 0), (2,2), (2,4),... (3,0), (3,3), (3,6),...,(4,0),(4,4),(4,8),...*}
  - substring: {(“ ”,“`mail`”),(“`m`”,”`mail`”), (“`ma`”,”`mail`”), (“`ai`”,”`mail`”),...}
  - plus_equal: {*(0,0,0), (0,1,1),(0,2,2),.., (1,2,3), (2,2,4), (3,2,5),.....*}

# Relations

**Definition:** A binary relation $\boldsymbol{R}$ is

symmetric, iff for all $a, b$, $a\boldsymbol{R}b$ implies $b\boldsymbol{R}a$

reflexive, iff for all $a$, $a\boldsymbol{R}a$ holds

transitive, iff for all $a, b, c$, $a\boldsymbol{R}b$ and $b\boldsymbol{R}c$ implies $a\boldsymbol{R}c$

**Definition:**

A relation which is symmetric, reflexive, and transitive is called equivalence relation.

# Concrete Syntax

$$\frac{e_1\ \mathit{SExpr} \qquad e_2\ \mathit{PExpr}}{e_1 + e_2\ \ \mathit{SExpr}} \qquad\qquad \frac{e\ \mathit{PExpr}}{e\ \mathit{SExpr}}$$

$$\frac{e_1\ \mathit{PExpr} \qquad e_2\ \mathit{FExpr}}{e_1 * e_2\ \ \mathit{PExpr}} \qquad\qquad \frac{e\ \mathit{FExpr}}{e\ \mathit{PExpr}}$$

$$\frac{}{i\ \mathit{FExpr}}\ i \in \mathit{Int} \qquad\qquad \frac{e\ \mathit{SExpr}}{(e)\ \mathit{FExpr}}$$

- the inference rules for *SExpr* defined the concrete syntax of a simple language, including precedence and associativity
- the concrete syntax of a language is designed with the human user in mind
- not adequate for internal representation during compilation

# Concrete vs abstract syntax

- Example:
  - *1 + 2 * 3*
  - *1 + (2 * 3)*
  - *(1) + ((2) * (3))*
  - what is the problem?
- Concrete syntax contains too much information
  - these expressions all have different derivations, but semantically, they represent the same arithmetic expression
- After parsing, we're just interested in three cases: an expression is either
  - an addition
  - a multiplication or
  - a number

# Concrete vs abstract syntax

- we use terms of the form

  $(Operator\ arg_1\ arg_2\ ....)$

  to represent parsed programs unambiguously; e.g.,

```
Plus (Num 1) (Times (Num 2) (Num 3))
```

- we define the abstract syntax of arithmetic expressions as follows:

$$\frac{t_1\ expr \qquad t_2\ expr}{(\mathtt{Times}\ t_1\ t_2)\ expr} \qquad\qquad \frac{t_1\ expr \qquad t_2\ expr}{(\mathtt{Plus}\ t_1\ t_2)\ expr}$$

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

# Concrete vs abstract syntax

- Parsers
  - check if the program (sequence of tokens) is derivable from the rules of the *concrete syntax*
  - turn the derivation into an *abstract syntax tree*
- Transformation rules
  - we formalise this with inference rules as a binary relation $\leftrightarrow$:

We write

$$e \; SExpr \leftrightarrow t \; expr$$

iff the (concrete syntax) expression *e* corresponds to the (abstract syntax) expression *t*.

Usually, many different concrete expressions correspond to a single abstract expression

# Concrete vs abstract syntax

- Example:
  - ★ *1 + 2 * 3*  *SExpr* ↔ `Plus (Num 1) (Times (Num 2) (Num 3)))` *expr*
  - ★ *1 + (2 * 3)*  *SExpr* ↔ `Plus (Num 1) (Times (Num 2) (Num 3)))` *expr*
  - ★ *(1) + ((2)*(3))* *SExpr* ↔ `Plus (Num 1) (Times (Num 2) (Num 3)))` *expr*

# Concrete vs abstract syntax

- Formal definition: we define a parsing relation $\leftrightarrow$ formally as an extension of the structural rules of the concrete syntax.

$$\frac{e_1\ \mathit{SExpr} \leftrightarrow e_1'\ \mathit{expr} \qquad e_2\ \mathit{PExpr} \leftrightarrow e_2'\ \mathit{expr}}{e_1 \mathbin{\mathbf{+}} e_2\ \ \mathit{SExpr} \leftrightarrow (\texttt{Plus}\ e_1'\ e_2')\ \mathit{expr}} \qquad \frac{e\ \mathit{PExpr} \leftrightarrow e'\ \mathit{expr}}{e\ \mathit{SExpr} \leftrightarrow e'\ \mathit{expr}}$$

$$\frac{e_1\ \mathit{PExpr} \leftrightarrow e_1'\ \mathit{expr} \qquad e_2\ \mathit{FExpr} \leftrightarrow e_2'\ \mathit{expr}}{e_1 * e_2\ \ \mathit{PExpr} \leftrightarrow (\texttt{Times}\ e_1'\ e_2')\ \mathit{expr}} \qquad \frac{e\ \mathit{FExpr}}{e\ \mathit{PExpr}}$$

$$\frac{}{i\ \mathit{FExpr} \leftrightarrow (\texttt{Num}\ i)\ \ \mathit{expr}}\ i \in \mathit{Int} \qquad \frac{e\ \mathit{SExpr} \leftrightarrow e'\ \mathit{expr}}{(e)\ \mathit{FExpr} \leftrightarrow e'\ \mathit{expr}}$$

# Concrete vs abstract syntax

- Formal definition: we define a parsing relation $\leftrightarrow$ formally as an extension of the structural rules of the concrete syntax.

$$\frac{e_1\ \mathit{SExpr} \leftrightarrow e_1'\ \mathit{expr} \quad e_2\ \mathit{PExpr} \leftrightarrow e_2'\ \mathit{expr}}{e_1 + e_2\ \ \mathit{SExpr} \leftrightarrow (\texttt{Plus}\ e_1'\ e_2')\ \mathit{expr}} \qquad \frac{e\ \mathit{PExpr} \leftrightarrow e'\ \mathit{expr}}{e\ \mathit{SExpr} \leftrightarrow e'\ \mathit{expr}}$$

$$\frac{e_1\ \mathit{PExpr} \leftrightarrow e_1'\ \mathit{expr} \quad e_2\ \mathit{FExpr} \leftrightarrow e_2'\ \mathit{expr}}{e_1 * e_2\ \ \mathit{PExpr} \leftrightarrow (\texttt{Times}\ e_1'\ e_2')\ \mathit{expr}} \qquad \frac{e\ \mathit{FExpr} \leftrightarrow e'\ \mathit{expr}}{e\ \mathit{PExpr} \leftrightarrow e'\ \mathit{expr}}$$

$$\frac{}{i\ \mathit{FExpr} \leftrightarrow (\texttt{Num}\ \ i)\ \ \mathit{expr}}\ i \in \mathit{Int} \qquad \frac{e\ \mathit{SExpr} \leftrightarrow e'\ \mathit{expr}}{\textbf{(}e\textbf{)}\ \mathit{FExpr} \leftrightarrow e'\ \mathit{expr}}$$

# The translation relation ↔

- The binary syntax translation relation
    - $e \leftrightarrow e'$

  can be viewed as translation function
    - input is $e$
    - output is $e'$
    - derivations are unambiguously determined by $e$
        - *since the grammar of the concrete syntax was unambiguous*
    - $e'$ is unambiguously determined by the derivation
        - *for each concrete syntax term, there is only one rule we can apply at each step*

# The translation relation ↔

- Derive the abstract syntax as follows:

  (1) bottom up, decompose the concrete expression e according to the left hand side of ↔

  (2) top down, synthesise the abstract expression e’ according to the right hand side of each ↔ from the rules used in the derivation.

- Example: derivation for *1 + 2 * 3* (we abbreviate *SExpr, PExpr, FExpr* with *S, P, F* respectively, and *expr* with *e*

$$\dfrac{\dfrac{\dfrac{\dfrac{1\ \mathit{Int}}{1\ F \leftrightarrow \texttt{(Num 1)}\ e}}{1\ P \leftrightarrow \texttt{(Num 1)}\ e}}{1\ S \leftrightarrow \texttt{(Num 1)}\ e} \quad \dfrac{\dfrac{\dfrac{2\ \mathit{Int}}{2\ F \leftrightarrow \texttt{(Num 2)}\ e}}{2\ P \leftrightarrow \texttt{(Num 2)}\ e} \quad \dfrac{3\ \mathit{Int}}{3\ F \leftrightarrow \texttt{(Num 3)}\ e}}{2 * 3\ P \leftrightarrow \texttt{(Times (Num 2) (Num 3))}\ e}}{1 + 2 * 3\ S \leftrightarrow \texttt{Plus (Num 1)(Times (Num 2)(Num 3))}\ e}$$

# Parsing and inference rules

- The parsing problem

  Given a sequence of tokens $s\ SExpr$ , find $t$ such that

  $$s\ SExpr \leftrightarrow t\ expr$$

- Requirements

  A parser should be

  - **total** for all expressions that are correct according to the concrete syntax, that is
    - there must be a $t\ expr$ for every $s\ SExpr$
  - **unambiguous**, that is for every $t_1$ and $t_2$ with
    - $s\ SExpr \leftrightarrow t_1$ expr and $s\ SExpr \leftrightarrow t_2$ expr

      we have $t_1 = t_2$

# Parsing and pretty printing

- The parsing problem

  Given a sequence of tokens *s SExpr* , find *t* such that

  $$s\ SExpr \leftrightarrow t\ expr$$

- What about the inverse?
  - given *t expr*, find *s SExpr*
- The inverse of parsing is unparsing
  - unparsing is often ambiguous
  - unparsing is often partial (not total)
- Pretty printing
  - unparsing together with appropriate formatting us called pretty printing
  - due to the ambiguity of unparsing, this will usually not reproduce the original program (but a semantically equivalent one)

# Parsing and pretty printing

## Example

Given the abstract syntax term

```
Times (Num 3) (Times (Num 4) (Num 5)))
```

pretty printing may produce the string

*“3 * 4 * 5” or “(3 * 4) * 5”*

- it’s best to chose the most simple, readable representation
- but usually, this requires extra effort

# Bindings

- Local variable bindings (let)

  Let's extend our simple expression language with one feature

  - variables and variable bindings
  - `let v = e`$_1$ `in e`$_2$

- Example:

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

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

- Concrete syntax (adding two new rules):

$$\frac{id\ Ident}{id\ FExpr} \qquad \frac{e_1\ SExpr \quad e_2\ SExpr}{\texttt{let}\ id = e_1\ \texttt{in}\ e_2 \quad FExpr}$$

# Bindings

- First order abstract syntax:

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

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

$$\frac{id\ \ Ident}{(\texttt{Var}\ \ id)\ \ expr} \qquad \frac{(\texttt{Var}\ \ \texttt{id})\ \ expr \quad t_1\ expr \quad t_2\ expr}{(\texttt{Let}\ id\ \ t_1\ \ t_2)\ \ expr}$$

# Bindings

- Scope
  - `let` $x = e_1$ `in` $e_2$ introduces -or binds- the variable $x$ for use within its ***scope*** $e_2$
  - we call the occurrence of $x$ in the left-hand side of the binding its binding ***occurrence*** (or defining occurrence)
  - occurrences of $x$ in $e_2$ are ***usage occurrences***
  - finding the binding occurrence of a variable is called ***scope resolution***
- Two types of scope resolution
  - ***static scoping***: scoping resolution happens at compile time
  - ***dynamic scoping***: resolution happens at run time (discussed later in the course

# Bindings

Example:

```
let
  x = y
in let y = 2
   in x
```

scope of y

scope of x

***Out of scope variable***: the first occurrence of y is ***out of scope***

# Bindings

Example:

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

Shadowing: the inner binding of `x` is shadowing the outer binding

- *static scoping*

```
const int b = 5;
int foo()
{
   int a = b + 5;
   return a;
}

int bar()
{
   int b = 2;
   return foo();
}

int main()
{
   foo(); // returns 10
   bar(); // returns 10
   return 0;
}
```

- *dynamic scoping*:

```
const int b = 5;
int foo()
{
   int a = b + 5;
   return a;
}

int bar()
{
   int b = 2;
   return foo();
}

int main()
{
   foo(); // returns 10
   bar(); // returns 7
   return 0;
}
```

example from: http://msujaws.wordpress.com

# Bindings

Example:

what is the difference between these two expressions?

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

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

α-equivalence:

- they only differ in the choice of the bound variable names
- we call them α-equivalent
- we call the process of consistently changing variable names α-renaming
- the terminology is due to a conversion rule of the λ-calculus
- we write $e_1 \equiv_\alpha e_2$ if two expressions are α-equivalent
- the relation $\equiv_\alpha$ is a equivalence relation

# Substitution

- Free variables
  - a free variable is one without a binding occurrence
    - `let x = 1 in x + y` y is free in this expression
- Substitution: replacing all occurrences of a free variable *x* in an expression *e* by another expression *e'* is called substitution
- Example: substituting `x` with `2 * y` in
    - `5 * x + 7` yields
    - `5 * (2 * y) + 7`

# Substitution

- We have to be careful when applying substitution:
  - `let y = 5 in y * x + 7`
  - `let z = 5 in z * x + 7`

α-equivalent

  - substitute `x` by `2 * y` in both
    - `let y = 5 in y * (2 * y) + 7`
    - `let z = 5 in z * (2 * y) + 7`

not α-equivalent anymore!

  - the free variable `y` of `2 * y` is captured in the first expression

# Substitution

- Capture-free substitution: to substitute `e'` for `x` in `e` we require the free variables in `e'` to be different from the variables in `e`
- We a can always arrange for a substitution to be capture free
  - use α-renaming of `e'` (the expression replacing the variable)
  - change all variable names that occur in `e` and `e'`
  - or use fresh variable names

# Higher-order abstract syntax

- A problem with (first-order) abstract syntax

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

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

$$\frac{id\ \ Ident}{(\texttt{Var}\ \ id)\ \ expr} \qquad \frac{(\texttt{Var}\ \ id)\ \ expr \quad t_1\ expr \quad t_2\ expr}{(\texttt{Let}\ id\ \ t_1\ \ t_2)\ \ expr}$$

- Defining and usage occurrence of variables are treated the same
  - abstract syntax doesn't differentiate between binding and using occurrence of a variable
  - it's difficult to identify α-equivalent expressions
  - variables are just terms, like numbers

# Higher-order abstract syntax

- Higher-order abstract syntax has variables and abstraction as special constructs
- A term of the form *x.t* is called an abstraction
- Structure of a higher-order term: a higher-order term can have one of four forms:

  (1) a constant (e.g., int, string)

  (2) a variable *x*

  (3) (Operator $t_1$ ... $t_n$)
    - `Num 4`
    - `Plus x (Num 4)`

  (4) *x.t* (i.e., the variable *x* is bound in term *t*)
    - `x. Plus x (Num 1)`
    - `x.y. Plus x y`

# Higher-order abstract syntax

- Higher-order abstract syntax for let-expressions

first-order

$$\frac{id\ \ Ident}{(\texttt{Var}\ \ id)\ \ expr} \qquad \frac{\texttt{var(id)}\ \ expr \quad t_1\ expr \quad t_2\ expr}{(\texttt{Let}\ id\ t_1\ t_2)\ \ expr}$$

higher-order

$$\frac{id\ \ Ident}{id\ \ expr} \qquad \frac{t_1\ expr \qquad t_2\ expr}{(\texttt{Let}\ t_1\ \ id.t_2)\ \ expr}$$

- Mapping of concrete to higher-order syntax

$$\frac{e_1\ SExpr \leftrightarrow t_1\ expr \qquad e_2\ SExpr \leftrightarrow t_2\ expr}{\texttt{let}\ id = e_1\ \texttt{in}\ e_2\ \texttt{end}\ SExpr \ \leftrightarrow\ (\texttt{Let}\ \ t_1\ \ id.t_2)\ expr}$$

$$\frac{id\ Ident}{id\ FExpr \leftrightarrow id\ expr}$$

- Example:

`let x = 5 in x+y` *SExpr* ↔ `(Let (Num 5) (x.Plus x y))` *expr*

# Substitution

**Definition:** *A notation for substitution*

We write

*t[x:=t']*

to denote a term *t* where all the occurrences of *x* have been replaced by the term *t'*.

- Example:

```
(Plus x y) [x :=(Num 1)]  = (Plus (Num 1) y)
```

**Definition:** *Renaming*

If we replace a variable in the binding and the body of an abstraction, it is called renaming, and the resulting term is α-equivalent to the original term:

$$x.t \equiv_\alpha y.t\ [x{:=}\ y]$$

*if y doesn't occur free in t (or $y \notin FV(t)$)*

# Substitution

- A inductive definition of *FV(t)*:

$FV(x) = \{x\}$

$FV(Op\ t1\ \ldots\ tn) = FV(t1) \cup \ldots. \cup FV(tn)$

$FV(x.t) = FV(t) \setminus \{x\}$

- Substituting one variable by another:

$x[x{:=}y] = y$

$z[x{:=}y] = z$, if $z \neq x$

$(Op\ t_1 \ldots t_n)\ [x{:=}y] = Op\ (t_1\ [x{:=}y]) \ldots (t_n\ [x{:=}y])$

$x.t\ [x{:=}y] = x.t$

$z.t\ [x{:=}y] = z.\ (t\ [x{:=}y])$, if $x \neq z$, $y \neq z$

$y.t\ [x{:=}y] =$ undefined, if $x \neq y$

# Substitution

- Substituting a variable by a term u:

$x\ [x{:=}u] = u$

$z\ [x{:=}u] = z$, if $z \neq x$

$(Op\ t_1 \ldots t_n)\ [x{:=}u] = Op\ (t_1\ [x{:=}u]) \ldots (t_n\ [x{:=}u])$

$x.t\ [x{:=}u] = x.t$

$z.t\ [x{:=}u] = z.\ (t\ [x{:=}u])$, if $x \neq z$, $z \notin FV(u)$

$y.t\ [x{:=}u] =$ undefined, if $y \in FV(u)$

# The untyped λ-Calculus

- Introduced in the 1936 by mathematician Alonzo Church
- Very simple, turing complete formalism for computations
- λ-terms:

$$\frac{}{\mathit{id}\ \lambda\text{-term}} \qquad \frac{t\ \lambda\text{-term}}{\lambda\, \mathit{id}.t\ \lambda\text{-term}}$$

$$\frac{t\ \lambda\text{-term} \quad s\ \lambda\text{-term}}{(t\ \ s)\ \lambda\text{-term}}$$

where $\mathit{id}$ is an identifier

# The untyped λ-Calculus

- The calculus has three rules:
  - α-conversion
    - if t $\equiv_\alpha$ s, then the two terms are equivalent in the calculus
  - β-reduction
    - $(\lambda\ x.t)s$ can be reduced to $t[x:=s]$
  - η-conversion
    - $\lambda\ x.\ (f\ x)$ is equivalent to $f$ if $x$ is not free in $f$

# The untyped λ-Calculus

- The simple untyped λ-calculus has no constants, conditional, built-in operations
- Natural numbers, arithmetic, logic operations can all be encoded in terms of the simple λ-calculus
  - `0 := λf.λx.x`
  - `1 := λf.λx.(f x)`
  - `2 := λf.λx.(f (f x))`
  - `3 := λf.λx.(f (f (f x)))`
  - successor function (+1): `λn.λf.λx.(f ((n f) x))`

# The untyped λ-Calculus

- Boolean values
  - `True:  λx.λy.x`
  - `False: λx.λy.y`
  - `If:    λc.λt.λe.c t e`

# Example: Boolean Expressions

- Consider a language of boolean expressions:

  $$bexpr ::= True \mid False \mid \neg\, bexpr \mid (bexpr) \mid$$
  $$bexpr \wedge bexpr \mid bexpr \vee bexpr$$

- Assume the usual precedence and associativity rules apply
- What would the rules for the concrete syntax of this language look like?
- What are the rules for abstract syntax of the language?
- What happens if we introduce variables?