Overview

So far:
- Judgements and inference rules
- Rule Induction
- Grammars specified using inference rules

Week 3:
- Judgments and relations
- First order abstract syntax
- Higher order abstract syntax
- Transition systems

Judgements and Relations

- A judgment states that a certain property holds for a specific object
- More generally, judgements express a relationship between a number of objects

Examples:
- 4 divides 16
- ail is a substring of sail
- 3 plus 5 equals 8

Infix notation to denote binary relations:
- 4 div 16
- ail substr sail

Relations

A binary relation \( R \) is
- symmetric, iff \( a R b \) implies \( b R a \)
- reflexive, iff for all \( a \), \( a R a \) holds
- transitive, iff for all \( a_1, a_2, a_3 \) if \( a_1 R a_2 \) and \( a_2 R a_3 \) then \( a_1 R a_3 \)

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

Concrete Syntax

- The inference rules for Expr/SExpr defined the concrete syntax of simple language
- Concrete syntax of a language is designed with the human user in mind
- Usually not adequate for internal representation during compilation
Grammar for arithmetic expressions:

\[
\begin{align*}
SExpr & \rightarrow e \\
SExpr + SExpr & \rightarrow e \\
SExpr \times SExpr & \rightarrow e \\
FExpr & \rightarrow e \\
FExpr & \rightarrow SExpr \\
int FExpr & \rightarrow e \\
\end{align*}
\]

The expressions
1. \(1 + 2 \times 3\)
2. \((1) + (2) \times (3)\)
3. \(((1)) + (2 \times 3)\)
all have different derivations, but semantically, represent the same arithmetic expression.

**First Order Abstract Syntax**

For an internal representation, only three cases are relevant:
- an addition,
- a multiplication, or
- a number

Terms of the form \(o(a_1, a_2, \ldots, a_n)\) represent this information unambiguously:

\[
\text{plus (num (1), times (num (2), num (3)))}
\]

**Abstract Grammar**

The definition of the abstract grammar of arithmetic expression therefore consists of only three rules:

\[
\begin{align*}
\text{num (int) expr} & \rightarrow e \\
\text{plus (t_1 expr, t_2 expr)} & \rightarrow e \\
\text{times (t_1 expr, t_2 expr)} & \rightarrow e
\end{align*}
\]

A parser
- checks if the program (sequence of tokens) is correct with respect to concrete syntax
- represents it as abstract syntax tree

How can the concrete syntax be translated into abstract syntax?
- We use inference rules to define a relation \(\rightarrow\)
  \[
  e_1 SExpr \rightarrow e_2 expr
  \]
  if and only if the (concrete grammar) expression \(e_1\) corresponds to (abstract grammar) expression \(e_2\)

Examples:
1. \(1 + 2 \times 3 SExpr \rightarrow \text{plus (num (1), times (num (2), num (3))) expr}\)
2. \(1 + (2 \times 3) SExpr \rightarrow \text{plus (num (1), times (num (2), num (3))) expr}\)
We define → over the structural rules for \( SExpr \), \( PExpr \) and \( FExpr \):

\[
\begin{align*}
e_1 SExpr & \rightarrow e_1' \ expr \\
e_2 PExpr & \rightarrow e_2' \ expr \\
e_3 FExpr & \rightarrow e_3' \ expr \\
e_1 SExpr + e_2 SExpr & \rightarrow plus(e_1', e_2') \ expr \\
e_1 SExpr \times e_2 SExpr & \rightarrow times(e_1', e_2') \ expr \\
\end{align*}
\]

Derivation of the abstract syntax for the expression \( 1 + 2 \times 3 \):

We abbreviate \( SExpr \), \( PExpr \), and \( FExpr \) with \( S \), \( P \), \( F \), respectively, and \( expr \) with \( e \).

To derive the abstract syntax expression:

- move from bottom up, decomposing the expression into the concrete syntax components, then
- starting from the leaves of the proof tree, determine the right hand side of each →, the abstract syntax expression.

Parsing: given a sequence of tokens \( s \ SExpr \), find \( t \ expr \) such that \( s \ SExpr \rightarrow t \ expr \)

A parser should be

- total on all expressions which are correct according to the concrete syntax:
- there is a \( t \ expr \) for every \( s \ SExpr \?
- unambiguous:
- there is no \( s \ SExpr \) for which there are two distinct terms \( t_1 \) and \( t_2 \), such that \( s \ SExpr \rightarrow t_1 \ expr \) and
  \( s \ SExpr \rightarrow t_2 \ expr \)

Unparsing and Pretty Printing

Unparsing: given a term \( t \ expr \), find \( e \ SExpr \) for which \( t \ expr \rightarrow e \ SExpr \)

Pretty printing: convert abstract syntax expression back to a string — will in almost all cases, not reproduce the original program, but a semantically equivalent.

Example: Given the term

\( \text{times (num (3)), times (num (4), num (5))} \)

unparsing and pretty printing may produce the string

- "3 * 4 * 5"
- "3 * (4 * 5)"

usually most simple, readable representation is chosen.
**Higher Order Abstract Syntax**

We extend our language of arithmetic expression by variables and variable bindings:

**Examples:**

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

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

**Concrete Syntax:**

- `e SExpr` (e) `FEExpr`
- `int FEExpr`
- `id FEExpr`
- `e1 SExpr e2 SExpr`
- `let id = e1 in e2 end FExpr`

---

**Higher Order Abstract Syntax**

**Scope Resolution:**
- A binding `let x = e1 in e2` introduces (or binds) the variable `x` for use within its scope `e2`.
- Finding the binding occurrence for a variable is called **scope resolution**.
- **Static scoping:**
  - Resolution happens at compile time.
- **Dynamic scoping:**
  - Resolution happens at run time (discussed later in the course).

---

**Variable out of scope:**

```plaintext
let x = y
in let y = 2
in x
end
end
```

the first occurrence of `y` is out of scope.

**Shadowing:**

```plaintext
let x = 5
in let x = 3
in x+y
end
end
```

the inner binding of `x` is shadowing the outer binding.
The following two expressions represent exactly the same computation:

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

They only differ in the choice of the bound variable names: this is called $\alpha$-equivalence.

We write $e_1 \equiv e_2$ if two expressions are $\alpha$-equivalent. The relation $\equiv$ is an equivalence relation.