**Aexp, A little language for arithmetic expressions**

**Syntax**

**Concrete syntax**

≈ phonemes, characters, words, tokens — the raw stuff of language

*Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, …*

**Grammar**

≈ collection of formation rules to organize parts into a whole. E.g.,
- words into noun phrases, verb phrases, …, sentences
- key words, tokens, … into expressions, statements, …, programs

- Abstract syntax
  ≈ a structure (e.g., labeled tree or data structure) showing how a “phrase” breaks down into pieces according to a specific rule.

---

**References**

- Andrew Pitts’ Lecture Notes on Semantics of Programming Languages
  We’ll be following the Pitts’ notes for a while and mostly using his notation.

- Matthew Hennessy’s Semantics of programming languages:
  [https://www.scss.tcd.ie/Matthew.Hennessy/splexternal2015/LectureNotes/Notes14%20copy.pdf](https://www.scss.tcd.ie/Matthew.Hennessy/splexternal2015/LectureNotes/Notes14%20copy.pdf)
  is very readable and very good.

- There are many of other good references in Hennessy’s reading list:
**Aexp’s abstract syntax**

Grammar:

\[ a ::= n \mid (a_1 + a_2) \mid (a_1 - a_2) \mid (a_1 \ast a_2) \]

\[ n ::= \ldots \]

In Haskell:

```haskell
data AExp = Num Integer
           | Plus AExp AExp
           | Minus AExp AExp
           | Times AExp AExp
```

((2 + 5) \ast 13) - 9

Minus (Times (Plus (Num 2) (Num 5))) (Num 13)

(Num 9)

As a Parse Tree

![Parse Tree Diagram](image)

**What do Aexp expression mean?**

Big-step rules

\[ a ::= n \mid (a_1 + a_2) \mid (a_1 - a_2) \mid (a_1 \ast a_2) \]

**PLUS:**

\[ \frac{a_1 \downarrow v_1 \quad a_2 \downarrow v_2}{(a_1 + a_2) \downarrow v} (v = v_1 + v_2) \]

**MINUS:**

\[ \frac{a_1 \downarrow v_1 \quad a_2 \downarrow v_2}{(a_1 - a_2) \downarrow v} (v = v_1 - v_2) \]

**MULT:**

\[ \frac{a_1 \downarrow v_1 \quad a_2 \downarrow v_2}{(a_1 \ast a_2) \downarrow v} (v = v_1 \ast v_2) \]

**NUM:**

\[ \frac{n \downarrow v}{(N[\langle n \rangle] = v)} \]

**Notes**

- \( a \downarrow v \equiv \) expression \( a \) evaluates to value \( v \).
- \( \downarrow \) is an evaluation relation.
- Upstairs assertions are called premises.
- Downstairs assertions are called conclusions.
- Parenthetical equations on the side are called side conditions.
- \( N : \text{numerals} \rightarrow \mathbb{Z} \). I.e., \( N[-43] = -43 \).
- The NUM\_BS rule is an example of an axiom.

**Digression: Rules, 1**

**General Format for Rules**

\[ \text{Name:} \quad \frac{\text{premise}_1 \quad \cdots \quad \text{premise}_k}{\text{conclusion}} \quad (\text{side condition}) \]

**Example**

- **Modus Ponens:**
  \[ p \implies q \quad p \quad \frac{q}{\downarrow} \]

- **Transitivity:**
  \[ x \equiv y \quad y \equiv z \quad \frac{x \equiv z}{\downarrow} \]

- **PLUS:**
  \[ \frac{a_1 \downarrow v_1 \quad a_2 \downarrow v_2}{(a_1 + a_2) \downarrow v} (v = v_1 + v_2) \]

**Digression: Rules, 2**

**Definition**

A rule with no premises is an axiom.

**Definition**

A rule is sound if and only if the conclusion is true whenever the premises (and side-condition—if any) are true.

**Question**

So an axiom is sound when . . . ?
Digression: Rules, 3

General Format for Rules

Name: \( \text{premise}_1, \ldots, \text{premise}_k \) (side condition)

conclusion

Proofs from gluing together rule applications

<table>
<thead>
<tr>
<th>Num: ( \frac{2 \downarrow 2}{2} )</th>
<th>Num: ( \frac{5 \downarrow 5}{5} )</th>
<th>Num: ( \frac{13 \downarrow 13}{13} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plus: ( \frac{(2 + 5) \downarrow 7}{(2 + 5 = 7)} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Times: ( \frac{(2 + 5) \downarrow 91}{(2 + 5 \times 13) = 91} )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The big-step semantics in Haskell

A Haskell version of the abstract syntax

```
data Aexp = Num Integer
  | Add Aexp Aexp
  | Sub Aexp Aexp
  | Mult Aexp Aexp
```

The big-step semantics as an evaluator function

\[
aBig \ (\text{Add } a1 \ a2) = (aBig \ a1) + (aBig \ a2)
\]
\[
aBig \ (\text{Sub } a1 \ a2) = (aBig \ a1) - (aBig \ a2)
\]
\[
aBig \ (\text{Mult } a1 \ a2) = (aBig \ a1) \times (aBig \ a2)
\]
\[
aBig \ (\text{Num } n) = n
\]

Rules can also be the basis of a computation

<table>
<thead>
<tr>
<th>( \frac{(2 + 5) \times 13 \downarrow ??}{??} )</th>
<th>( \downarrow \downarrow )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{2 \downarrow 2 \ 5 \downarrow 5}{(2 + 5) \downarrow 7 \ 13 \downarrow 13} )</td>
<td></td>
</tr>
</tbody>
</table>

Do these rules make sense?

Theorem

Suppose \( e \in \text{Aexp} \).
Then there is a unique integer \( v \) such that \( e \downarrow v \).

Proof (by rule induction).

Case: NUM. This is immediate.

Case: PLUS.
By IH, there are unique \( v_1 \) and \( v_2 \) such that \( a_1 \downarrow v_1 \) and \( a_2 \downarrow v_2 \).
By arithmetic, there is a unique \( v \) such that \( v = v_1 + v_2 \).
Hence, there is a unique \( v \) such that \( a_1 + a_2 \downarrow v \).

Cases: MINUS and MULT. These follow mutatis mutandis.
What do \texttt{Aexp} expression mean?

**Small-step rules**

\[ a ::= n \mid (a_1 + a_2) \mid (a_1 - a_2) \mid (a_1 \times a_2) \mid v \]

\[
\begin{align*}
PLUS-1_{\text{SSS}}: & \quad a_1 \rightarrow a'_1 \quad (a_1 + a_2) \rightarrow (a'_1 + a_2) \\
PLUS-2_{\text{SSS}}: & \quad a_2 \rightarrow a'_2 \quad (a_1 + a_2) \rightarrow (a_1 + a'_2) \\
PLUS-3_{\text{SSS}}: & \quad (v_1 + v_2) \rightarrow v \quad (v = v_1 + v_2) \\
NUM_{\text{SSS}}: & \quad n \rightarrow v \quad (N[[n]] = v)
\end{align*}
\]

**Notes**

- These are rewrite rules.
- We now allow values in expressions.
- \( a \rightarrow a' \) is a transition.
- \( a \rightarrow a' \) \( \equiv \) expression \( a \) evaluates (or rewrites) to \( a' \) in one-step.
- \( v \) is a terminal expression.
- The rules for \(-\) and \(\times\) follow the same pattern as the rules for \(+\).

**Class exercise**

Show:

\[
((3 \times 2) + (8 - 3)) \times (5 - 2)
\]

\[
\begin{cases}
(6 + (8 - 3)) \times (5 - 2) \\
((3 \times 2) + 5) \times (5 - 2) \\
((3 \times 2) + (8 - 3)) \times 3
\end{cases}
\]

**Some full small-step derivations of transitions**

The derivations show that the steps in the transition sequence below are legal (i.e., follow from the rules).

\[
\begin{align*}
& \quad \text{MINUS}_3 \quad (8 - 3) \rightarrow 5 \\
& \quad \text{PLUS}_2 \quad (6 + (8 - 3)) \rightarrow (6 + 5) \\
& \quad \text{MULT}_1 \quad ((6 + 5) \times (5 - 2)) \rightarrow ((6 + 5) \times (5 - 2)) \\
& \quad \text{PLUS}_3 \quad (6 + 5) \rightarrow 11 \\
& \quad \text{MULT}_2 \quad (11 \times (5 - 2)) \rightarrow 11 \times (5 - 2) \\
& \quad \text{MINUS}_3 \quad (5 - 2) \rightarrow 3 \\
& \quad \text{MULT}_2 \quad (11 \times (5 - 2)) \rightarrow 11 \times 3 \\
& \quad \text{MULT}_3 \quad (11 \times 3) \rightarrow 33
\end{align*}
\]

There is a lattice of transitions.
Properties of operational semantics

**Definition**
A transition system \((\Gamma, \sim, T)\) is deterministic when for all \(a, a_1,\) and \(a_2:\)
If \(a \sim a_1\) and \(a \sim a_2,\) then \(a_1 = a_2.\)

**Theorem**
The big-step semantics for \(Aexp\) is deterministic.

*The proof is an easy rule induction.*

**Theorem**
The given small-step semantics \((Aexp \cup Z, \Rightarrow, Z)\) fails to be deterministic, but for all \(a \in Aexp\) and \(v_1, v_2 \in Z,\) if \(a \Rightarrow^* v_1\) and \(a \Rightarrow^* v_2,\) then \(v_1 = v_2.\)

*This proof is tricky because of the nondeterminism.*

A deterministic small-step semantics for \(Aexp\)

\[ a ::= n \mid (a_1 + a_2) \mid (a_1 - a_2) \mid (a_1 \times a_2) \mid v \]

**PLUS-1'**\[\begin{align*}
a_1 & \rightarrow a'_1 \\
 a_1 + a_2 & \rightarrow a'_1 + a_2
\end{align*}\]

**PLUS-2'**\[\begin{align*}
a_2 & \rightarrow a'_2 \\
v_1 + a_2 & \rightarrow v_1 + a'_2
\end{align*}\]

**PLUS-3'**\[\begin{align*}
v_1 + v_2 & \rightarrow v \\
 (v = v_1 + v_2)
\end{align*}\]

**NUM**\[\begin{align*}
n & \rightarrow v \\
(N \llbracket n \rrbracket = v)
\end{align*}\]

The leftmost path through the lattice of transitions

Why multiple flavors of semantics?

They provide different views of computations.
- Big-step is good for reasoning about how the (big) pieces of things fit together.
- Small step is good at reasoning about the (small) steps of a computation fit together.
- Small step semantics is much better at modeling inherent nondeterminism (e.g., in concurrent programs).