A Simple Langauge of Arithmetic Expressions

Syntax

and

"big-step" operational semantics

Syntax of ARITH

- Concrete syntax: the rules by which programs can be expressed as strings of characters
  - relevant to many of our desiderata 期望之物
  - (readability, familiarity, speed of compilation, ...)
- The same concepts can be embodied in many concrete, syntaxes, not all of them equally good.
  - Should semicolon separate or terminate statements?
  - How should comments be indicated?
- There are solid principles for concrete syntax.
  - Finite automata and context-free grammars.

有限自动机 和 上下文无关的语法

Abstract Syntax

- We ignore parsing issues and study programs given as abstract syntax trees.

AST

AST 是 parse tree 解析树

- An abstract syntax tree is a parse tree.
  - More convenient for formal and algorithmic manipulation.
  - Fairly independent of the concrete syntax.

AST 是 抽象语法， 独立于 具体语法

- An abstract syntax for ARITH

抽象语法

e ::= n	integer literals
| $e_1 + e_2$	sum
| $e_1 * e_2$	product

Thursday, January 10, 13

Analysis of ARITH

- Questions to answer:
  - What is the "meaning" of a given ARITH expression?
  - How would we go about evaluating an ARITH expression?
  - How are the evaluator and the meaning related?

An Operational Semantics

- Specifies how expressions should be evaluated.
- Defined by cases on the form of expressions:
  - n evaluates to n
    - n is a normal form, no need to evaluate further
  - $e_1 + e_2$ evaluates to n if
    - e_1 evaluates to n_1
    - e_2 evaluates to n_2
    - and n is the sum of n_1 and n_2
  - $e_1 * e_2$ evaluates to n if
    - e_1 evaluates to n_1
    - e_2 evaluates to n_2
    - and n is the product of n_1 and n_2

Alternative Formulation

- Notation: $e \Downarrow n$ means that "e evaluates to n"
  - This is a judgment
    a statement about a relation between e and n)
- Allows us to write evaluation rules more concisely
  - $n \Downarrow n$
  - $(e1 + e2) \Downarrow n$ if
    - $e1 \Downarrow n1$
    - $e2 \Downarrow n2$
    - and n is the sum of n_1 and n_2
  - $(e1 * e2) \Downarrow n$ if
    - $e1 \Downarrow n1$
    - $e2 \Downarrow n2$
    - and n is the product of n_1 and n_2

推理

Operational Semantics as Inference Rules

$$\frac{}{n \Downarrow n}$$

$$\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2 \qquad n \text{ is the sum of } n_1 \text{ and } n_2}{e_1 + e_2 \Downarrow n}$$

$$\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2 \qquad n \text{ is the product of } n_1 \text{ and } n_2}{e_1 * e_2 \Downarrow n}$$

- Meaning: "above the line" implies "below the line"
- These rules are
  - evaluation rules for the big-step operational semantics
  - derivation rules for the judgement $e \Downarrow n$

How to Read the Rules?

- Forward, as inference rules:
  - If we know that the hypothesis judgments hold then we can infer that the conclusion judgment also holds.
  - E.g., if we know that $e_1 \Downarrow 5$ and $e_2 \Downarrow 7$, then we can infer that $e_1 + e_2 \Downarrow 12$.

How to Read the Rules?

- Backward, as evaluation rules:
  - Suppose we want to evaluate $e_1 * e_2$
  - i.e., find n s.t. $e_1 * e_2 \Downarrow n$ is derivable using the rules.
  - By inspection of the rules we notice that the last step in the derivation of $e_1 * e_2 \Downarrow n$ **must be** the addition rule:
    - The conclusions of other rules would not match $e_1 * e_2 \Downarrow n$.
    - (This is called reasoning by <u>inversion</u> on the derivation rules.)
  - Thus we must find n_1 and n_2 such that $e_1 \Downarrow n_1$ and $e_2 \Downarrow n_2$ are derivable. And this is done recursively.
- Since there is exactly one rule for each kind of expression we say that the rules are <u>syntax-directed.</u>
  - At each step at most one rule applies.

Thursday, January 10, 13

Challenge!

- (3 + 5) * (4 + 2) evaluates to what?

- $(3 + 5) * (4 + 2) \;\Downarrow\; ?$

- I start, you continue ...

Haskell code for ARITH Operational Semantics

```
data Exp = IntExp Int
         | SumExp Exp Exp
         | MulExp Exp Exp

eval :: Exp -> Int
eval (IntExp n)      = n
eval (SumExp e1 e2) = (eval e1) + (eval e2)
eval (MulExp e1 e2) = (eval e1) * (eval e2)
```

Haskell code for ARITH Operational Semantics

```
test_eval :: Bool
test_eval = let e = MulExp (SumExp (IntExp 3) (IntExp 5))
                           (IntExp 2) in
            (eval e) == 16 -- True
```