2. A Toy Calculator

2.0 Abstract Syntax

Let us consider a toy language the *abstract syntax* of which consists of expressions in an inductively defined set Exp that is built up using the following three constructions:

1. (Base case: Integer numerals) We choose the constructor `numeral(N)` to be the PROLOG *coding* for the numeral N that is (in turn) a representation of the integer n. (Note: we shall use the terminology "numeral" for a syntactic representation of the mathematical notion of a *number*).

2. (Inductive case: plus) The sum of two expressions $E1, E2$; We choose the constructor `plus(E1,E2)` to be the PROLOG *coding* for the expression $E1 + E2$ (that is a representation of the mathematical expression $e_1 + e_2$).

3. (Inductive case: times) The product of two expressions $E1, E2$; We choose the constructor `times(E1,E2)` to be the PROLOG *coding* for the expression $E1 * E2$ (that is a representation of the mathematical expression $e_1 \times e_2$.)

2.1 Toy Denotational Semantics

We will first specify a *mathematical evaluator* as a function (i.e., a functional relation) between expressions in *abstract syntax* and their mathematical meaning (*semantics*) as integers. Let us follow a convention that the syntactic elements are written using upper-case, usually in green, but underlined in contexts where the colour cannot be shown properly. In the presentation below, we will follow the convention of having N represent the integer value n.

The functional specification of the evaluator, a prototypical "*definitional interpreter*", is given as a mathematical function $eval : Exp \rightarrow \mathbb{Z}$. This function is recursive, and defined following the inductive structure of (the abstract syntax of) expressions. In specifying the function, we follow the now-standard convention of delimiting the syntactic expression (which I would normally highlight in green) by enclosing it in special brackets $[\![\quad]\!]$. The operators $+$ and $\times$ in the right-hand side of the equations in the definition below are *integer addition* and *multiplication*.

$$eval[\![\ N \ ]\!] = n$$

$$eval[\![E_1 + E_2]\!] = eval[\![E_1]\!] + eval[\![E_2]\!]$$

$$eval[\![E_1 * E_2]\!] = eval[\![E_1]\!] \times eval[\![E_2]\!]$$

Programming Exercise (Solved below): Try to code the *eval* function in PROLOG as a predicate `eval(E,V)`.

> When coding this function in PROLOG, which can also be used as a "meta-language" for defining the syntax and semantics of our toy language, we will

exploit its computational features, in particular pattern-matching. Relational specifications are fairly easy to encode in PROLOG. Functional specifications are special cases, in which the function is expressed as a relation. Coding the definitional evaluator in PROLOG involves writing out the function *eval* in a relational form — where a function of k arguments is represented as a $(k+1)$ -ary relation, with the last position representing the result.

2.2 Operational Semantics

A calculator is an program that operates entirely on representational forms. That is, it is an algorithm whose data structures are the abstract syntax of the language, and it takes in an expression, and returns as its answer an expression that represents the value of the input expression.

Let us *specify* a simple calculator using "inference rules". These rules are to be read as "if all the antecedent statements of the rule hold, then the consequent statement holds". Some rules have no antecedents. These are called "axioms". Other rules may have "side conditions" or "provisos", which must hold in order for the rule to apply. In some treatments, the provisos are included as an antecedent. However, we prefer to write them alongside the rules — partly because we usually are defining a recursive relation, and will use induction as a proof technique. Note that induction and recursion are flip sides of the same coin — if a set is defined inductively, many useful functions *from* that set are recursively defined.

In some situations, we read the inference rules in a bottom-to-top manner: for example, the **(CalcPlus)** rule below can be read as "expression $\underline{E_1 + E_2}$ calculates to a numeral $\underline{N}$ if expression $\underline{E_1}$ calculates to some numeral $\underline{N_1}$ and expression $\underline{E_2}$ calculates to some numeral $\underline{N_2}$, provided the addition algorithm *PLUS* adds numerals $\underline{N_1}, \underline{N_2}$ to yield numeral $\underline{N}$".

(CalcNum) $$\frac{}{\underline{N} \Longrightarrow \underline{N}}$$

(CalcPlus) $$\frac{\underline{E_1} \Longrightarrow \underline{N_1} \quad \underline{E_2} \Longrightarrow \underline{N_2}}{\underline{E_1 + E_2} \Longrightarrow \underline{N}} \text{ provided } PLUS(\underline{N_1}, \underline{N_2}, \underline{N})$$

assuming we have an algorithm *PLUS* (a circuit in our hypothetical machine) that takes the numerals $\underline{N_1}, \underline{N_2}$ representing of two numbers, and returns a numeral $\underline{N}$ that represents their sum.

(CalcTimes) $$\frac{\underline{E_1} \Longrightarrow \underline{N_1} \quad \underline{E_2} \Longrightarrow \underline{N_2}}{\underline{E_1 * E_2} \Longrightarrow \underline{N}} \text{ provided } TIMES(\underline{N_1}, \underline{N_2}, \underline{N})$$

assuming we have an algorithm *TIMES* (a circuit in our hypothetical machine) that takes the numerals $\underline{N_1}, \underline{N_2}$ representing of two numbers, and returns a numeral $\underline{N}$ that represents their product.

[Note: This style of specification is called "operational" since it specifies the operational behaviour of the program. Within the operational style, this style is called "big-step" or Kahn-style semantics (after the French computer scientist Gilles Kahn) since the computation is one giant step from inputs to outputs. Note that if for an input, there is no output (e.g., if the program goes into an infinite loop or if it "hangs", no corresponding input-output pair will be specified.]

2.2.1 The Calculator Specification in PROLOG

The calculator can be coded in PROLOG as a predicate `calculate(E, A)`

```
/* A tiny integer calculator */
calculate(numeral(N),numeral(N)) :- integer(N).
calculate(plus(E1,E2), numeral(N)) :-
                          calculate(E1, numeral(N1)),
                          calculate(E2, numeral(N2)),
                          N is N1+N2.
calculate(times(E1,E2), numeral(N)) :-
                          calculate(E1, numeral(N1)),
                          calculate(E2, numeral(N2)),
                          N is N1*N2.
/*  A test case
eval( plus(
            times(numeral(3), numeral(4)),
            times(numeral(5), numeral(6))
          ),
     V).
*/
```

2.3 Correctness of the Operational Specification

Note that the calculator, as specified, is not guaranteed to be a function, whether partial or total. The functional nature of this relation is a property that we need to establish.

If (and indeed it is so) every numeral N corresponds to a unique integer n, and also that the circuits for *PLUS* and *TIMES* are (total) functions (the PROLOG implementations of addition and multiplication are hopefully so, i.e., for every pair of representation of integers $\underline{N_1}, \underline{N_2}$ represented in PROLOG as `N1` and `N2`, `N1+N2` and `N1*N2` both returns exactly one PROLOG integer representation), then the `calculates` relation is indeed a (total) function.

Nor is it clear that any answer given by the calculator specification will indeed be a valid representation of the *mathematical value* to which any expression will *eval*. However, if it does not do so, then our calculator is worthless, a junk specification, and we should throw away any implementation of this specification. For the correctness of the specification we require that the circuits $PLUS(\underline{N_1}, \underline{N_2}, \underline{N})$ and $TIMES(\underline{N_1}, \underline{N_2}, \underline{N})$ — and their corresponding PROLOG analogues `N1+N2` and `N1*N2` — are *semantically correct*, i.e., they are (total) functions that return answers that are indeed the mathematical representations of $\underline{N_1} + \underline{N_2}$ and $\underline{N_1} * \underline{N_2}$

Theorem 2.0 (Soundness of $\underline{E} \Longrightarrow \underline{N}$, i.e., of `calculate(E, A)`).
Assume that (i) every numeral $\underline{N}$ corresponds to a unique integer $n \in \mathbb{Z}$, and (ii) for every pair of numerals $\underline{N_1}, \underline{N_2}$, (ii-a) if $PLUS(\underline{N_1}, \underline{N_2}, \underline{N})$ then $eval[\![\underline{N}]\!] = eval[\![\underline{N_1}]\!] + eval[\![\underline{N_2}]\!]$, and (ii-b) if $TIMES(\underline{N_1}, \underline{N_2}, \underline{N})$ then $eval[\![\underline{N}]\!] = eval[\![\underline{N_1}]\!] \times eval[\![\underline{N_2}]\!]$.

Then for all $\underline{E}, \underline{N} \in Exp$, if $\underline{E} \Longrightarrow \underline{N}$ then $eval[\![\ \underline{E}\ ]\!] = eval[\![\ \underline{N}\ ]\!]$.
[In terms of the PROLOG encoding, if `calculate(E, A)` then `E = numeral(N),` where `N` is the representation of $eval[\![\ \underline{E}\ ]\!]$]

The proof is by induction on the structure of the (abstract syntax of) expressions $\underline{E} \in Exp$. This is because Exp is defined by induction, and also because the relation $\underline{E} \Longrightarrow \underline{N}$ is defined by inductive case analysis on $\underline{E} \in Exp$

Proof. By structural induction on $\underline{E} \in Exp$.
Assume $\underline{E} \Longrightarrow \underline{N}$.

Base Case: $ht(\underline{E}) = 0$. By case analysis, $\underline{E} \equiv \underline{N'}$ for some numeral $\underline{N'}$
By (**CalcNum**) $\underline{N'} \Longrightarrow \underline{N'}$. So $\underline{N} \equiv \underline{N'}$.
Trivially, $eval[\![\ \underline{E}\ ]\!] = eval[\![\ \underline{N}\ ]\!]$.

Induction Hypothesis (IH): Assume that for all $\underline{E'}, \underline{N'} \in Exp$ such that $ht(\underline{E'}) \leq k$, if $\underline{E'} \Longrightarrow \underline{N'}$ then $eval[\![\ \underline{E'}]\!] = eval[\![\ \underline{N'}]\!]$.

Induction Step: Consider $\underline{E} \in Exp$ such that $ht(\underline{E}) = k+1$.
By analysis, either (i) $\underline{E} \equiv \underline{E_1} + \underline{E_2}$ or (ii) $\underline{E} \equiv \underline{E_1} * \underline{E_2}$.

(i) $\underline{E} \equiv \underline{E_1} + \underline{E_2}$, where $ht(\underline{E_1}) \leq k$ *and* $ht(\underline{E_2}) \leq k$.
Suppose $\underline{E} \Longrightarrow \underline{N}$.
Then by (**CalcPlus**) $\underline{E_1} \Longrightarrow \underline{N_1}$, $\underline{E_2} \Longrightarrow \underline{N_2}$ for some numerals $\underline{N_1}, \underline{N_2}$.
By IH on $\underline{E_1}$, $eval[\![\ \underline{E_1}]\!] = eval[\![\ \underline{N_1}]\!]$ and by IH on $\underline{E_2}$, $eval[\![\ \underline{E_2}]\!] = eval[\![\ \underline{N_2}]\!]$
Recall that by definition, $eval[\![\underline{E_1} + \underline{E_2}]\!] = eval[\![\underline{E_1}]\!] + eval[\![\underline{E_2}]\!]$
Thus from the IHs, $eval[\![\underline{E_1} + \underline{E_2}]\!] = eval[\![\underline{N_1}]\!] + eval[\![\underline{N_2}]\!]$
By the condition $PLUS(\underline{N_1}, \underline{N_2}, \underline{N})$ and the assumption on $PLUS$ that $eval[\![\underline{N}]\!] = eval[\![\underline{N_1}]\!] + eval[\![\underline{N_2}]\!]$, we have $eval[\![\ \underline{E}\ ]\!] = eval[\![\ \underline{N}\ ]\!]$.

(ii) $\underline{E} \equiv \underline{E_1} * \underline{E_2}$. Similar to (i). [**Exercise**: Work out this case] □

Note that **Soundness** only says that if the calculator returns an answer, then the answer is semantically correct. However, there is no guarantee that it does indeed return an answer on any given input expression. So a calculator that does not return a result is trivially sound! But probably not what you want. One can take a simplified version of the soundness theorem to show that the calculator specification is indeed a total function.

The converse of soundness, called "*Completeness*", may be somewhat harder to state and prove (especially as the language gets more complicated)... and indeed *does not hold of actual calculators* because of the finiteness limitations of the numerals which we can actually write or with which we can compute. However, for the present specification, can can show that *in principle* (that is if we have enough time and space for the representation of even very large numbers and also circuits that can add or multiply them) the calculator does indeed return an answer that is a correct representation of the meaning to which any express evaluates.

Theorem 2.1 (Completeness of $\underline{E} \Longrightarrow \underline{N}$, i.e., of `calculate(E, A)`).
Assume that (i) every integer $n \in \mathbb{Z}$ can be represented as a numeral $\underline{N}$;
(ii) for every pair of numbers n_1, n_2 represented by numerals $\underline{N_1}, \underline{N_2}$, (ii-a) the mathematical integer $n_1 + n_2$ is represented by some numeral $\underline{N}$ such that $PLUS(\underline{N_1}, \underline{N_2}, \underline{N})$, and (ii-b) the integer $n_1 \times n_2$ is represented by some numeral $\underline{N}$ such that $TIMES(\underline{N_1}, \underline{N_2}, \underline{N})$.

Then for all $\underline{E} \in Exp$, and $n \in \mathbb{Z}$, if $eval[\![\ \underline{E}\ ]\!] = n$ then $\underline{E} \Longrightarrow \underline{N}$ for some numeral $\underline{N}$ such that $eval[\![\ \underline{N}\ ]\!] = n$

> [In terms of the PROLOG encoding, if $eval[\![\ \underline{E}\ ]\!] = n$ then `calculate(E, numeral(N))` where $eval[\![\ \underline{N}\ ]\!] = n.$]

The proof is by induction on the structure of the (abstract syntax of) expressions $\underline{E} \in Exp$. This is because Exp is defined by induction, and also because the recursive function $eval[\![\ \underline{E}\ ]\!]$ is defined by inductive case analysis on $\underline{E} \in Exp$

Proof. By structural induction on $\underline{E} \in Exp$.
Assume $eval[\![\ \underline{E}\ ]\!] = n$

Base Case: $ht(\underline{E}) = 0$. By case analysis, $\underline{E} \equiv \underline{N'}$ for some numeral $\underline{N'}$.
By (**CalcNum**) $\underline{N'} \Longrightarrow \underline{N'}$. We can take $\underline{N} \equiv \underline{N'} \equiv \underline{E}$.
Trivially, $eval[\![\ \underline{E}\ ]\!] = eval[\![\ \underline{N}\ ]\!] = n$.

Induction Hypothesis (IH): Assume that for all $\underline{E'} \in Exp$ such that $ht(\underline{E'}) \leq k$, if $eval[\![\ \underline{E'}\ ]\!] = n'$ then $\underline{E'} \Longrightarrow \underline{N'}$ for some numeral $\underline{N'}$ such that $eval[\![\ \underline{N'}\ ]\!] = n'$

Induction Step: Consider $\underline{E} \in Exp$ such that $ht(\underline{E'}) = k+1$.
By analysis, either (i) $\underline{E} \equiv \underline{E_1 + E_2}$ or (ii) $\underline{E} \equiv \underline{E_1 * E_2}$.

(i) $\underline{E} \equiv \underline{E_1 + E_2}$, where $ht(\underline{E_1}) \leq k$ *and* $ht(\underline{E_2}) \leq k$.
Suppose $eval[\![\ \underline{E}\ ]\!] = n$
Then by the recursive definition of $eval[\![\ \underline{E}\ ]\!]$,
$eval[\![\underline{E_1 + E_2}]\!] = eval[\![\underline{E_1}]\!] + eval[\![\underline{E_2}]\!]$
Let $eval[\![\ \underline{E_1}\ ]\!] = n_1$ and $eval[\![\ \underline{E_2}\ ]\!] = n_2$, i.e., $n_1 + n_2 = n$
By IH on $\underline{E_1}, \underline{E_2}$ respectively, $\underline{E_1} \Longrightarrow \underline{N_1}$, $\underline{E_2} \Longrightarrow \underline{N_2}$ for some numerals $\underline{N_1}, \underline{N_2}$, such that $eval[\![\ \underline{N_1}\ ]\!] = n_1$ and $eval[\![\ \underline{N_2}\ ]\!] = n_2$

Now by assumption about $PLUS$ that the integer $n_1 + n_2$ is represented by some numeral $\underline{N}$ such that $PLUS(\underline{N_1}, \underline{N_2}, \underline{N})$, we have by the rule (**CalcPlus**) (read top to down) that $\underline{E} \Longrightarrow \underline{N}$ where $eval[\![\ \underline{N}\ ]\!] = n_1 + n_2 = n$.

(ii) $\underline{E} \equiv \underline{E_1 * E_2}$. Similar to (i). [**Exercise**: Work out this case] □

2.4 A Compiler for the Toy Language

The calculator specification is an input-output relational specification of an interpreter. An alternative model of program execution is to compile the program into a sequence of "op codes" for an abstract machine. The abstract machine that we have in mind is a simple stack-based evaluator, where the main data structure is a stack on which values are pushed, or from which the operands of an operator are popped and replaced by the result of the operation.

Let us first define the op-codes of the machine and their intended operation:

1. ldop (N) - which loads the specified constant value N onto the stack;
2. plusop - which pops the top two values from the stack, adds them and pushes the resulting value back onto the stack;
3. timesop - which pops the top two values from the stack, multiplies them and pushes the resulting value back onto the stack;

First some preliminaries about lists and a relational encoding in PROLOG of the append function:

```
append([], L, L).
append([X|R], L, [X|Z]):- append(R, L, Z).
```

This code may be read as follows: (1) appending an empty list to any list `L` results in the list `L`. (2) appending a non-empty list, whose first element is `X` and the rest of the list is `R`, to a list `L` results in a non-empty list whose first element is `X`, and the rest of the list is `Z`, which is the result of appending `R` to `L`.
Note that PROLOG does not allow nested recursive calls, but instead employs recursive calls on the right side of the `:-` symbol, and the chaining of outputs to inputs by having the named output of one predicate to be an input to another.

The *compile* function takes an expression `(E)` and returns a *list* or sequence `(C)` of op-codes. The relational encoding `compile(E,C)` of the *compile* function is quite similar in structure to the relational specification for `calculate(E,A)`.

```
/* A simple compiler for the toy language */

compile(numeral(N),[ldop(N)]) :- integer(N).
compile(plus(E1,E2),C ) :-
                          compile(E1, C1),
                          compile(E2, C2),
                          append(C1, C2, C3),
                          append(C3, [plusop], C).
compile(times(E1,E2), C) :-
                          compile(E1, C1),
```

```
                                compile(E2, C2),
                                append(C1, C2, C3),
                                append(C3, [timesop], C).
```

Compiling a numeral $\underline{N}$ results in a *singleton* list consisting of the op-code ldop (N) Compiling an expression $\underline{E_1} + \underline{E_2}$ results in a sequence of op-codes that first has the op-codes sequence `C1` obtained by compiling expression $\underline{E_1}$ followed by the sequence `C2` obtained from compiling $\underline{E_2}$, followed by the singleton list containing the op-code plusop . The code for $\underline{E_1} * \underline{E_2}$ is similar, but with timesop instead. Note the similarity in structure between the predicates `compile(E,C)` and `calculate(E, A).`

2.4.1 The Abstract Machine

Let us first specify the one-step transitions of the stack machine, writing [ ] for the empty list and $x :: l$ for a non-empty list with initial element x and remaining list l. Note that the same notation works for a stack.

(Load) $\langle S,\ \text{ldop}(\underline{N}) :: C\rangle \gg\!\!\Rightarrow \langle \underline{N} :: S, C\rangle$
(Plus) $\langle \underline{N_2} :: \underline{N_1} :: S,\ \text{plusop} :: C\rangle \gg\!\!\Rightarrow \langle \underline{N} :: S, C\rangle$ where $PLUS(\underline{N_1}, \underline{N_2}, \underline{N})$
(Times) $\langle \underline{N_2} :: \underline{N_1} :: S,\ \text{timesop} :: C\rangle \gg\!\!\Rightarrow \langle \underline{N} :: S, C\rangle$ where $TIMES(\underline{N_1}, \underline{N_2}, \underline{N})$

> [Note: Strictly speaking, we should be pushing data-structure representations of values on the stack but to make the PROLOG code easier to read, we fudge this syntax-*vs*-semantics distinction below.]

Note that the machine cannot make a transition if there are no op-codes that remain to be executed. In this case, the machine halts, and we can "unload" the value at the top of the stack (assuming the stack is not empty; otherwise it is an error).

The machine gets "stuck" (also an error) if the op-code to be executed is either plusop or timesop, but there aren't at least two values on the stack for the arithmetic operation.

The encoding in PROLOG combines the single transitions indicated above with a *tail-recursive call* to execute from the resulting state. The predicate `stackmc(S, C, A)` is to be read as "executing the stack machine from a state $\langle S, C\rangle$ on the LHS of a single transition results in unloading an answer `A` if executing the machine from the state $\langle S', C'\rangle$ on the RHS of that transition results in unloading that answer `A`.

> [Note the use in PROLOG of the idiom "`N is N1 + N2`" to force the computation of the arithmetic expression. See what happens if you omit this PROLOG phrase and instead write `N1 + N2` in place of `N` on the stack.]

```
/* A stack machine: A simple abstract machine */


stackmc( [A|S1], [ ], A).
stackmc( S, [ldop(N)|C], A) :-
                              stackmc([N|S], C, A).
```

```
stackmc( [N2|[N1|S]], [plusop|C], A ) :-
                          N is N1+N2, stackmc([N|S], C, A).
stackmc( [N2|[N1|S]], [timesop|C], A ) :-
                          N is N1*N2, stackmc([N|S], C, A).


/*  Suggested test query:
   compile(
            plus( times(numeral(3), numeral(4)),
                  times(numeral(5), numeral(6)) ),
             C),
  stackmc( [ ], C, A).

*/
```

The suggested test first compiles an expression into an op-code sequence, and then executes this op-code sequence starting with the empty stack.

2.4.2 Correctness of the Abstract Machine model

Proposition 2.2 (Correctness of compile-execute)
For all $\underline{E}, \underline{N} \in Exp$, $\langle [], compile(\underline{E})\rangle >\!\!\Longrightarrow^* \langle \underline{N} :: S, []\rangle$ if and only if $\underline{E} \Longrightarrow \underline{N}$ (for some stack S)

Ideally we should be able to prove this result using induction on the structure of $\underline{E}$, since both $\underline{E} \Longrightarrow \underline{N}$ and the function *compile* are defined accordingly. One technical issue is that the execution rules of the abstract machine are not. Instead, the machine execution rules are by case analysis on the first op-code in the sequence.

Moreover, it is also hard to apply the induction hypothesis (IH) if in the statement of the result, the stack in the LHS state and the op-code list in the RHS state are to be empty. (**Exercise**: Why?)

Fortunately, there is a very strong isolation (modularity) in how the abstract machine executes: the execution of the code obtained from *compile*($\underline{E}$) *never accesses the stack below what it places there,* and execution consumes the op-codes *serially*.

Hence we need to generalise the statement of the result to the (much stronger) result:

Theorem 2.3 (Correctness of compile-execute)
For all $\underline{E}, \underline{N} \in Exp$, for *all* stacks S' and op-code lists C':
$\langle S', compile(\underline{E})@C'\rangle >\!\!\Longrightarrow^* \langle \underline{N} :: S', C'\rangle$ *if and only if* $\underline{E} \Longrightarrow \underline{N}$

Solution: A possible PROLOG coding of the *eval* function. Note the similarity and yet a major difference from the `calculate(E, A)` predicate — the result of evaluation is not a numeral but a (PROLOG) integer value.

```
/* A definitional interpreter for the toy language */
eval(numeral(N),N) :- integer(N).
eval(plus(E1,E2), N) :-
                          eval(E1, N1),
                          eval(E2, N2),
                          N is N1+N2.
eval(times(E1,E2), N) :-
                          eval(E1, N1),
                          eval(E2, N2),
                          N is N1*N2.



/*  A test case
calculate( plus(
            times(numeral(3), numeral(4)),
            times(numeral(5), numeral(6))
                ),
          N).
*/
```