プログラム言語論 資料

算術演算言語とその意味論

亀山 幸義

算術演算のみができる小さなプログラム言語 Arith を例として、構文、表示的意味、操作的意味、抽象機械がどのようなものであるか理解しよう。

ここでの言語は、足し算、掛け算、自然数定数で構成される「式」がプログラムであるような言語である。

1 構文の定義

1.1 具体構文

言語 Arith の構文を、BNF (Backus Normal Form) で定義する。

$$n ::= 0 \mid 1 \mid 2 \mid \cdots \quad \text{(自然数の定数)}$$
$$e ::= n \mid (e) \mid e + e \mid e * e \quad \text{(式)}$$

この定義はシンプルでよいが、2 つの問題がある。

- 問題 1: 式の構文 e の定義には曖昧さがある。すなわち、1 つの文字列に対して 2 通り以上の方法で、式であることを導くことができる。たとえば、$1 + 2 + 3$ の構文が 2 通りの方法で導出される。これを確認せよ。
- 問題 2: 多くの言語では、10 と ((10)) は、式としての内部表現が同じである。逆にいうと、処理系の内部では「同じ」とおもっているものが、上記の式の構文 e の定義では、2 つ以上の異なる文字列に対応している場合がある。

問題 1 における曖昧さをなくした構文の定義の一例は以下の通りである。

$$n ::= 0 \mid 1 \mid 2 \mid \cdots \quad \text{(自然数定数)}$$
$$e ::= f \mid e + f \text{ (式)}$$
$$f ::= g \mid f * g \text{ (補助的な構文クラス)}$$
$$g ::= n \mid (e) \text{ (補助的な構文クラス)}$$

演習 1.1 $1 + 2 * (3 + 4) * 5$ が上記の e の定義にあてはまることを導出せよ。また、その導出が *1* 通りしかないことを確認せよ。

演習 1.2 言語 *Arith* に引き算と割り算を追加したものに対して、曖昧さがない構文を定義せよ。ただし、かけ算と割り算の結合力は同等で、足し算と引き算も同等で、前者は後者より強いものとする。また、同じ結合力のものが複数ならんでいるときは、左側のものを優先する。なお、単項の引き算 *(-10* のように *1* 引数のマイナス*)* はないものとする。

1.2 抽象構文

前節で述べたのは、具体構文 (concrete syntax) あるいは、表層構文 (surface syntax) である。

問題 2 における冗長さをなくすため、処理系の内部表現となる構文を考え、抽象構文 (abstract syntax) と言う。

「処理系の内部表現となる構文」というのは随分乱暴な (意味不明な) 言葉であるが、式などの言語要素の処理がしやすくなるよう、木の形で一意的に表現したものである。

抽象構文の定義:

$$N ::= \texttt{Int}(n)$$
$$E ::= \texttt{Int}(n) \mid \texttt{Plus}(E, E) \mid \texttt{Times}(E, E)$$

ここで、Plusとか Timesというのは一種の「タグ」であり、木のノードにつけた名前である。異なる種類のノードを区別するため異なる名前をつけている。上記のように書いたが、抽象構文としては曖昧さも冗長さもない木であればよいので、普通に、$e ::= n \mid (e+e) \mid (e*e)$ という定義でもまったく問題ない。これを抽象構文とおもってもよい。

この原稿では、具体構文と抽象構文のどちらの話をしているかを明確に区別したいので、$(e+e)$ と書かず、$\texttt{Plus}(e,e)$ と書くことにする。(以後、ほとんどのケースで抽象構文のみを使う。)

1.3 構文解析

構文解析 (parse) は、具体構文で与えられた表現を、抽象構文の表現に変換することである。

2 意味の定義 (表示的意味論)

式の意味を決めるにあたり、「式 $1+2$ と式 $3*1$ は同じ意味を持つ」ように決めたい。表示的意味論の方法では、すべての式を、適当な (数学的にきちんと定まる) 集合の要素に対応付ける形で意味を与える。今回の小さなプログラム言語では、「適当な集合」として、自然数の集合を取れば良いので、そうしてみる。

式 e を自然数に対応付ける。対応付けられた自然数を $[\![e]\!]$ と書くことにする。

$$[\![\texttt{Int}(n)]\!] = n$$
$$[\![\texttt{Plus}(e_1,e_2)]\!] = [\![e_1]\!] \dot{+} [\![e_2]\!]$$
$$[\![\texttt{Times}(e_1,e_2)]\!] = [\![e_1]\!] \dot{*} [\![e_2]\!]$$

ただし、$\dot{+}$ は、式の構文にでてくる $+$ の記号ではなく、2 つの自然数の足し算をあらわす。$\dot{*}$ も同様

例. $[\![\texttt{Plus}(\texttt{Int}(1),\texttt{Times}(\texttt{Int}(2),\texttt{Int}(3)))]\!] = [\![\texttt{Int}(1)]\!] \dot{+} [\![\texttt{TimesInt}(2),\texttt{Int}(3)]\!] = 1 \dot{+} ([\![\texttt{Int}(2)]\!] \dot{*} [\![\texttt{Int}(3)]\!]) = 7$

演習 2.1 $[\![\mathit{Times}(\mathit{Plus}(\mathit{Int}(2),\mathit{Int}(3)),\mathit{Plus}(\mathit{Int}(4),\mathit{Int}(5)))]\!]$ を求めよ。

3 意味の定義 (操作的意味論の 1 つ)

表示的意味論は、式が「結局、何を表しているか」だけを示すものであり、数学的にはこれでよいが、コンピュータ屋としては、式が「どういう風に計算されていって、最終的な答えになるか」という過程も大事にしたい。(また、今回のように単純な言語では、どういう順番に計算しても同じ結果になるが、print 文などの「副作用」がある言語では、式を計算する順番が大事になるので、上記のような単純な形で表示的意味論を与えることはできなくなる。)

ということで、操作的意味論の出番である。ここでは、操作的意味論のなかでも代表的な自然意味論 (natural semantics) を与える。(他の代表的な操作的意味論は、構造的操作的意味論 (structural operational semantics) とよばれ、SOS と略記される。)

「式 e を計算すると自然数 n を得る」ことを意味する関係 $e \downarrow n$ の帰納的定義:

$$\frac{}{\texttt{Int}(n) \downarrow n} \qquad \frac{e_1 \downarrow n_1 \quad e_2 \downarrow n_2 \quad (n_1 \dot{+} n_2 = n)}{\texttt{Plus}(e_1,e_2) \downarrow n} \qquad \frac{e_1 \downarrow n_1 \quad e_2 \downarrow n_2 \quad (n_1 \dot{*} n_2 = n)}{\texttt{Times}(e_1,e_2) \downarrow n}$$

例題。$\texttt{Plus}(\texttt{Int}(1),\texttt{Times}(\texttt{Int}(2),\texttt{Int}(3))) \downarrow 7$ の導出は以下の通り。

$$\dfrac{\dfrac{}{\texttt{Int}(1)\downarrow 1} \quad \dfrac{\dfrac{}{\texttt{Int}(2)\downarrow 2} \quad \dfrac{}{\texttt{Int}(3)\downarrow 3}}{\texttt{Times}(\texttt{Int}(2),\texttt{Int}(3))\downarrow 6}}{\texttt{Plus}(\texttt{Int}(1),\texttt{Times}(\texttt{Int}(2),\texttt{Int}(3)))\downarrow 7}$$

4 抽象機械

抽象機械 (abstract machine) は、その名前の通り、機械 (コンピュータ) を抽象化したものである。抽象化 (abstraction) とは、ものごとを難しくすることと思われがちであるが、実際はその逆であり、「何らかのデータを忘れて (捨てて) 簡単にすること」である。たとえば、名前と身長と年齢の 3 つのデータからなるデータベースで、身長のデータを捨てて、名前と年齢の 2 つのデータかなるデータベースにすることは抽象化の一種である。具体構文では、10 と (10) と ((10)) が違っていたのに、その括弧の違いをわすれて同じとおもうことも、一種の抽象化である。同様に、実在のコンピュータ (CPU のほか、キャッシュ、メインメモリ、ハードディスクなどをもつ) において、「キャッシュもメインメモリもハードウェアも同じ (その差を忘れる)」ことにより、「CPU と、一種類のメモリだけ」からなる仮想的なコンピュータが得られる。これが抽象機械の一種である。

通常、「抽象機械」というと、実在の機械から大幅に抽象化をして、簡素にしたモデルを指すことが多い。歴史上、有名な抽象機械には Landin の SECD 機械, Felleisen の CEK 機械があり、これ以外にも、CAM, ZAM などがある。コンピュータのハードウェアに、より近いものとしては (通常は「仮想機械」とよばれる) JVM, LLVM などがある。

ここでは、CEK 機械を単純化した CK 機械を与えよう。この機械では、C と K が主役である。

- C ... control string (その抽象機械の機械語で書かれたプログラム; ここでは「(抽象構文の) 式」)
- K ... continuation (一般には「継続」と呼ばれるものだが、ここでは単にスタックを表す)

抽象機械の状態: $\texttt{eval}\langle C \mid K\rangle$ と $\texttt{apply}\langle K \mid n\rangle$ の 2 種類の状態がある。(n は自然数)
抽象機械の動作 (状態遷移):

$$e \to \texttt{eval}\langle e \mid \texttt{init}\rangle \quad \text{(初期状態)}$$

$$\begin{aligned}
\texttt{eval}\langle \texttt{Int}(n) \mid K\rangle &\to \texttt{apply}\langle K \mid n\rangle \\
\texttt{eval}\langle \texttt{Plus}(e_1, e_2) \mid K\rangle &\to \texttt{eval}\langle e_1 \mid \mathit{plus}1(e_2)\texttt{::}K\rangle \\
\texttt{eval}\langle \texttt{Times}(e_1, e_2) \mid K\rangle &\to \texttt{eval}\langle e_1 \mid \mathit{times}1(e_2)\texttt{::}K\rangle
\end{aligned}$$

$$\begin{aligned}
\texttt{apply}\langle \mathit{plus}1(e)\texttt{::}K \mid n\rangle &\to \texttt{eval}\langle e \mid \mathit{plus}2(n)\texttt{::}K\rangle \\
\texttt{apply}\langle \mathit{plus}2(n_2)\texttt{::}K \mid n_1\rangle &\to \texttt{apply}\langle K \mid n\rangle \quad (n_1 \dot{+} n_2 = n) \\
\texttt{apply}\langle \mathit{times}1(e)\texttt{::}K \mid n\rangle &\to \texttt{eval}\langle e \mid \mathit{times}2(n)\texttt{::}K\rangle \\
\texttt{apply}\langle \mathit{times}2(n_2)\texttt{::}K \mid n_1\rangle &\to \texttt{apply}\langle K \mid n\rangle \quad (n_1 \dot{*} n_2 = n)
\end{aligned}$$

$$\texttt{apply}\langle \texttt{init} \mid n\rangle \to n \quad \text{(最終状態)}$$

式 e が与えられると、$\texttt{eval}\langle e \mid \texttt{init}\rangle$ という初期状態に移動し、2 行目から下から 2 行目の規則をつかって状態遷移を繰返し、もし、最後の規則をつかって最終状態になったら、計算は終了する。

この抽象機械は、スタック 1 つを利用して、上記の式の計算を実行する形となっている。$\texttt{Plus}(e_1, e_2)$ の形の式に対して、e_1 を評価するときは e_2 をスタックに積み、e_2 を評価するときは、e_1 の評価結果 n をスタックに積んでいる。

演習 4.1 以下の式を上記抽象機械で実行せよ。

- *Int(10)*
- *Times(Plus(Int(2),Int(3)),Int(4))*
- *Times(Int(2),Plus(Int(3),Int(4)))*

演習 4.2 上記の抽象機械の実行は決定的 *(determistic)* である。すなわち、どの状態からも、その次の状態はたかだか *1* つである。その理由を述べよ。

演習 4.3 式 e が与えられると、その抽象機械での計算は必ず停止する。その理由を述べよ。

演習 4.4 式 e の長さ *(*足し算とかけ算の個数*)* が N のとき、その式の抽象機械での計算ステップ数 *(*状態遷移の回数*)* を求めよ。

演習 4.5 *Arith* に引き算と割り算を加える。ただし、負の数になるときは *0* とし、割り切れないときは小数点以下を切り捨てる。このとき、上記の抽象機械をどう拡張したらよいか述べよ。