単純な算術式言語の処理系の試作

2015SE019 池田竜平　2015SE050 三輪峻大　2015SE070 須藤拓也
指導教員：横山哲郎

1　はじめに

計算機の中央処理装置において解釈実行できるのは，機械語のみである．しかし，機械語は人間が理解するのは難しい．高水準言語は，機械語よりも人間にとって理解しやすい．また，プログラミング言語を説明するためには，プログラミング言語の機能を定義し，その意味を述べるための形式的な言語が必要である．プログラミング言語を記述するための言語もまたプログラミング言語で書かれていればそのままプログラムを実行することができる．プログラミング言語を記述するためのプログラミング言語によって形式化をすることができれば，意味を厳密に定めることができる [1]．また，式を構成させる際は，括弧の式や基本の構成を同様に用いるように定義しなければならない．

本研究では，簡単な加減演算式を定義し，語句解析，構文解析，意味解析を行う．また，加減演算式を加減乗除演算式に拡張も行う．さらに，加減乗除演算式に対して別の意味解析を行う．加減演算式の定義には，関数型言語の Haskell 言語を用いる．関数型言語は数学的な扱いに適した性質を持っており，基本的なことを数学的に述べることができる．また，算術式の構文を BNF，代数型を用いて試作する．Haskell と C 言語の 2 言語でプログラムを実装するときの長所と短所の比較を行う．比較方法としては，BNFC のサンプルから生成されたプログラムを比較に用いて，パーサーコンビネータ，構文を比較する．単純な構文をもつ命令型言語の文の語句解析器と構文解析器を試作する．

2　プログラミング言語の定義

プログラムは文字列で表現される．プログラミング言語は表現された文字列をどのように構成し，どのような計算をするのかを定めるものである．プログラミング言語を定義するためには構文と意味を定める必要がある．文字列に解釈を与えることで言語を定めることができる．言語を定めるために構文を記述し，構文を形式的に表すためには構文規則によって定める．構文規則を表記記述するために BNF 記法を用いる．

2.1　文脈自由文法

プログラミング言語の文法構造の定義には，正規表現で定義された語彙をどのように組み合わせたら, プログラミング言語の文になるか規定しなければならない [2]．この定義を文脈自由文法で行う．文脈自由文法 $G = (N, \Sigma, P, S)$ を定義する．ここで，N は非終端記号の有限集合，Σ は終端記号の有限集合，P は $A \to \alpha (A \in N, \alpha \in (\Sigma \cup N)^*)$ の形の生成規則の有限集合，S $(\in N)$ は開始記号である．各記号は非終端か終端のいずれかであり非終端記号は P の規則により別の記号に変わるが，終端記号はそれ以上変化しない．ここで $\alpha = \alpha_1, \alpha_2$ $(\alpha_1, \alpha_2 \in A)$ であり，かつ $A \Rightarrow \gamma$ でもあるなら $\alpha \Rightarrow \alpha_1 \gamma \alpha_2$ と記述する．つまり α の中にある非終端記号の 1 つを生成規則により $\beta \in \alpha$ に書き換えられたとすると，$\alpha \Rightarrow \beta$ と記述する．この 2 項関係の反射的推移閉包を $\Rightarrow^*$ と記述する．文脈自由文法 $G = (N, \Sigma, P, S)$ によって表現される文の集合 $L(G)$ を $L(G) = \{\omega \mid P \Rightarrow^* \omega, \omega \in N^*\}$ と定義できる．このようにして S から P の規則を適用して Σ の集合だけになるまで書き換えることで表現できる文法を文脈自由文法という．

文脈自由文法は，BNF で定義することができる．BNF は，

$$\langle \text{非終端記号} \rangle ::= \text{構成}$$

という形の構文規則の有限集合と開始記号で文脈自由文法を定義する．ここで，構成は記号列である．

2.2　代数型による表現

代数型は構成子をもつ関数型言語で使われるデータの型である．同じ性質を持つデータは 1 つの型として扱われ共通の演算が定義される．代数型は構成子によってデータ型を定義し，新たなデータ型を作り出すこともできる．BNF 記法で書かれた構文規則

$$\langle \text{構文単位名} \rangle ::= \text{構成 } 1 \mid \text{構成 } 2 \mid \cdots \mid \text{構成 } n$$

は，代数型

```
data 代数型名 = 構成子 1 構成要素列 1
              | 構成子 2 構成要素列 2
              ⋮
              | 構成子 n 構成要素列 n
```

で表現をすることができる．このように構成子，構成要素列により代数型を定義する．

構成子は文の構成法の識別のために用いることや，構成要素列の型の値を代数型の値に変換する関数として使われる．上記の構成子はそれぞれ

```
構成子 i :: 構成要素列 i -> 代数型名
```

という型をもつ．また基本記号の情報を構成子に含めることができる．

2.3　構文の記述

基本記号の集合は $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /\}$ であり，加減演算式の構文規則は，

$$\begin{aligned}
\langle \text{Expr} \rangle &::= \langle \text{Numeral} \rangle \\
&\quad \mid \langle \text{Expr} \rangle \texttt{ + } \langle \text{Expr} \rangle \mid \langle \text{Expr} \rangle \texttt{ - } \langle \text{Expr} \rangle \\
\langle \text{Numeral} \rangle &::= \langle \text{Digit} \rangle \mid \langle \text{Numeral} \rangle \langle \text{Digit} \rangle \\
\langle \text{Digit} \rangle &::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}
\end{aligned}$$

となる．この構文規則を代数型で表現すると

```
data Expr = Num Numeral
          | Pexpr Expr Expr
          | Mexpr Expr Expr
data Numeral = Single Digit
             | Composite Numeral Digit
data Digit = Digit_0 | ··· | Digit_9
```

となる．{0, 1, . . . , 9} の情報はそれぞれ構成子 Digit_0, Digit_1, ..., Digit_9 に含まれている．また +, - の情報は + は Pexpr, - は Mexpr に含まれている．

2.4 語句解析

語句解析は，文字列から語句要素の列に変換する．本章では加減演算式に従った文字列の語句解析を行う．

語句要素は構文上の計算対象であり，それらの間の等価性が分かればいい．語句要素を表す Token Tag を定義する．

```
type Token Tag = (Tag, [Char])
```

これは語句の種類を示す Tag と語句要素の文字列で表現されている．語句の種類を表す Tag を定義する．

```
data Tag = T_Num | T_Sym | T_Junk
```

語句の種類は T_Num は数の表記を表し，T_Sym は演算子を表す．T_Junk は語句の意味を持たないものを表し語句要素を表現できる．

```
lexer :: Parser Char [Token Tag]
lexer = lexime [(some (satisfy isSpace), T_Junk),
                (number, T_Num),
                (anyof string ["(",")",
                               "+","-"],
                               T_Sym) ]

lex :: [Char] -> [Token Tag]
lex = (strip T_Junk) . fst . head . lexer
```

関数 lexer は型構成子 Parser により Char のリストを入力として受け取り，(Tag，Char) の形で出力する．関数 lexime により並んでいる文字列を認識している．関数 some (satisfy isSpace) により空白の最長列を認識している．関数 number は数字が 1 つ以上並んでいることを認識している．関数 anyof string により文字列からなるリスト ["(",")","+","-"] からリストのなかのいずれかと一致することを認識している．関数 lex により文字列を解析する際，語句要素が数であるなら，数のまとまりを語句要素として認識し，(T_Num,"数") で出力される．語句要素が演算子であるなら，(T_Sym,"演算子") で出力される．T_Junk 語句として意味を持たないので関数 strip により T_Junk が出力されないようにしている．

2.5 構文解析

構文解析は，語句解析によって変換された語句要素列が定義された文法の構造をしているかを解析し，その構造をもつ構文木に変換する．本章では加減演算式の構文解析を行う．

関数 syn を定義する．

```
syn :: [Token Tag] -> Prog
syn = fst . head . prog
```

syn は語句解析により文字列から変換された語句要素を構文木の型に変換する関数である．関数 parse を定義する．

```
parse :: [Char] -> Prog
parse = syn . lex
```

parse は文字列を構文木の型に変換するための syn と lex の合成関数である．

```
type Prog = Expr
prog = expr
```

ここで型 Prog は加減演算式の代数型の型 Expr としている．補助関数 expr を定義する．

```
expr :: Parser (Token Tag) Expr
expr = factor `seqp`
       many ((lit "+" `x_seq` expr `using`
                   flip Pexpr) `alt`
       (lit "-" `x_seq` expr `using` flip Mexpr))
              `using` uncurry (foldl (flip id))
```

expr は語句要素の (T_Sym, "+"),(T_Sym, "-") を型 Expr の Pexpr, Mexpr に変換をしている．関数 pnumber, valdig を定義する．

```
pnumber = Num . pnumber' . map valdig
    where
    pnumber' (n:ns) = foldl Composite (Single n) ns

valdig :: Char -> Digit
valdig '0' = Digit_0
valdig '1' = Digit_1
           ⋮
valdig '9' = Digit_9
```

valdig は文字列で入力された文字を Digit の型に変換をしている．pnumber により代数型の Numeral に変換をしている．関数 factor を定義する．

```
factor = kind T_Num `using` pnumber `alt`
         lit "(" `x_seq` expr `seq_x` lit ")"
```

factor により語句要素の語句の列を pnumber の型にし，構文解析を行った際構文木の型への変換をしている．expr の具象構文には構成を認識し，解析結果として使う必要がないため括弧を seq により結合している．

2.6 構文の拡張

2.3 節の加減演算式の構文規則を拡張する．加減演算式の構文規則を四則演算ができるように拡張する．構文規則

$$\langle \text{Expr} \rangle \quad ::= \langle \text{Numeral} \rangle \mid \langle \text{Expr} \rangle \ \langle \text{BinOpr} \rangle \ \langle \text{Expr} \rangle$$
$$\langle \text{BinOpr} \rangle ::= + \mid - \mid * \mid /$$

という ⟨Expr⟩ を定義する．ここで，積算演算子，除算演算子を表す *, / を追加した．

$$\langle \text{Expr} \rangle ::= \langle \text{Expr} \rangle + \langle \text{Expr} \rangle \mid \langle \text{Expr} \rangle - \langle \text{Expr} \rangle$$

の Expr の演算式の構成要素を

$$\langle \text{Expr} \rangle \quad ::= \langle \text{Expr} \rangle \langle \text{BinOpr} \rangle \langle \text{Expr} \rangle$$
$$\langle \text{BinOpr} \rangle ::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}$$

とすることで Expr の演算式の構成要素を 1 つにまとめプログラムが短く記述することができる．構文規則を代数型で表現すると

```
data Expr    = Num Numeral
             | Bexpr BinOpr Expr Expr
data Numeral = Single Digit
             | Compoite Numeral Digit
data Digit   = Digit_0 | ··· | Digit_9
data BinOpr  = Plus | Minus | Times | Div
```

のようになる．基本記号は構成子に情報を含めることができるので 0, · · · , 9 を Digit_0, . . ., Digit_9 と表現し，+, -, *, /を Plus, Minus, Times, Div と表現している．

2.7 意味解析

意味論では構文に意味を与える．本稿では，意味の記述は表示的意味論に基づく．すなわち，BNF 記法により定められた構文によりできる算術式を Haskell プログラムによる意味関数で意味を記述する．2.6 節の加減乗除演算式に意味を与える関数 numval，digval，expval を定義する．

```
numval :: Numeral -> Int
numval (Single d)      = digval d
numval (Composite n d) = numval n * 10 + digval d

digval :: Digit -> Int
digval Digit_0 = 0
digval Digit_1 = 1
               ⋮
digval Digit_9 = 9

expval :: Expr -> Int
expval (Num n)             = numval n
expval (Bexpr Plus  e1 e2) = expval e1 + expval e2
expval (Bexpr Minus e1 e2) = expval e1 - expval e2
expval (Bexpr Times e1 e2) = expval e1 * expval e2
expval (Bexpr Div   e1 e2) = expval e1 `div`
                             expval e2
```

意味関数 digval により Digit のそれぞれの構成要素に数学上の 0 ～ 9 の意味を与えることで，1 文字の数字の数を与える．numval は代数型 Numeral の構成要素に意味を与えている．1 文字の数字 Single d の場合，digval d により意味を定める．2 文字以上の数字 Composite n d の場合，再帰的定義を用いることにより，numval n，*，10，+，digval d それぞれの意味から Composite n d の意味を定めている．expval は Expr の構成要素に意味を与えている．expval は ⟨BinOpr⟩ の構成要素に四則演算のそれぞれの意味を与える．これにより expval は四則演算の意味を持つ関数となる．

3 実装言語の比較

3.1 BNFC

BNF(Backus-Naur-Form) は，プログラミング言語の構文を記述する時に使われる．BNFC は BNF に従った構文規則がほぼそのまま記述されたテキストファイルを入力として, Lex や YACC といった従来のツールを用いて語句解析器，構文解析器，及びプリティプリンタを生成するための変換器である．BNFC を用いると，典型的なコンパイラの語句解析器や構文解析の実装において，ソースコードの行数を 9 割ほど削減できる．BNFC を用いて，ANSI C と Java などの実用的な言語の実装を行うことができる [3].

3.2 Haskell と C 言語での算術式言語の実装方法の比較

BNFC で生成された Haskell と C 言語のプログラムを比較する．Haskell で算術式のプログラムを定義する．

```
data Exp =
       EAdd Exp Exp
     | ESub Exp Exp
     | EMul Exp Exp
     | EDiv Exp Exp
     | EInt Integer
      deriving (Eq,Ord,Show)
```

Haskell で算術式のプログラムは代数型を用いる．代数データ型は data で新しい型 Exp を定義する．型 Exp は EAdd，ESub，EMul,EDiv のいずれかの構成子と Exp Exp の構成か EInt Integer の 5 つに定義する．deriving(Eq,Ord,Show) は新しく定義した型 Exp で型クラス Eq，Ord，Show を定義している．Eq は 2 つの引数が同じであるかを評価できる型を，Ord は順序付けを可能にする型を，Show は出力の際，文字列化可能にする型を表す.

```
interpret :: Exp -> Integer
interpret x = case x of
 EAdd e0 e1 -> interpret e0 + interpret e1
 ESub e0 e1 -> interpret e0 - interpret e1
 EMul e0 e1 -> interpret e0 * interpret e1
 EDiv e0 e1 -> interpret e0 `div` interpret e1
 EInt n     -> n
```

interpret は型 Exp を整数の型 Integer に変換している．interpret は case 式とパターンマッチを用いて場合分けをしている．型 Exp の構成子が EAdd の場合は足し算，ESub の場合は引き算，EMul の場合はかけ算，EDiv の場合は割り算を表す．Haskell では型推論と呼ばれるものがあるため interpret がどのような型であるかを記述しなくても適正な型に決定してくれる．C では型推論はないので struct や int のようにそれぞれ明確に定義しなければならない．Haskell の case 式ではパターンマッチングを使用している．パターンマッチングでは，C 言語の if 文や switch 文のようなことができる．パターンマッチングは再帰的に書くときに if 文や switch 文よりも簡潔に書くことが出来る．switch 文は整数値でしか場合分けできないがパターンマッチングなら変数や構成子で場合分けできるため，具体的に場合分けをしやすい．パ

ターンマッチングは上から順番に評価していくため一番最初にほとんどの条件で通ってしまうものをもってきたりするとそれより後にあるパターンを評価せずに終わってしまうため順番を間違えてしまうと適切な結果を出せない場合がある．

次に，Haskell で実装したプログラムを C 言語で定義する．

```
struct Exp_ {
   enum {is_EAdd, is_ESub, is_EMul,
                is_EDiv, is_EInt} kind;
      union {
        struct { Exp exp_1, exp_2; } eadd_;
        struct { Exp exp_1, exp_2; } esub_;
        struct { Exp exp_1, exp_2; } emul_;
        struct { Exp exp_1, exp_2; } ediv_;
        struct { Integer integer_; } eint_;
      } u;
    };

typedef struct Exp_ *Exp;

Exp make_EAdd(Exp p0, Exp p1);
Exp make_ESub(Exp p0, Exp p1);
Exp make_EMul(Exp p0, Exp p1);
Exp make_EDiv(Exp p0, Exp p1);
Exp make_EInt(Integer p0);
```

Haskell の代数データ型で実装していたものを C 言語では列挙型，構造体，共用体を組み合わせて実装している．構造体のタグ名を Exp_と名付け列挙型 enum でメンバの is_EAdd，is_ESub，is_EMul，is_EDiv，is_EInt にそれぞれ順番に 0 から整数値を振り分けている．これにより整数値が何を表しているかをわかるように名前を付けることができるため可読性が上がる．列挙型は定義した以外の値になることが無いため，これ以外の可能性を考える必要がない．また，switch 文ですべての条件を網羅する必要があるが，列挙型はとりうる値が決まっている為，case 文で定義したものがない，または定義した以外のものがある場合にエラーとなる．これにより switch 文でそれ以外を表す default を使わなくても良い．default を使った場合，enum に追加した際に，エラーを検出できないが，default を使わないことで列挙型に追加した際に，switch 文で追加したものが書き忘れていた場合などに，エラーが検出される．共用体 union は構造体と似たような構造をしているが，構造体のメンバはメモリを別々で使用するのに対し，共用体は 1 番最初のメンバの大きさのメモリを確保し，そのメモリを共有している．メモリを共有することでメモリ使用率を下げることができる．しかし，メンバごとにメモリサイズが大きく異なると，メモリサイズが小さいメンバのメモリ効率が悪くなる．C 言語は，構成子の役割を持つものを自動で生成できないため，構造体とは別で生成しなければならない．これにより，プログラムを 2 段階で記述しなければならないため C 言語のプログラムは Haskell よりも長くなってしまう．まず，typedef を用いて struct Exp_ *を Exp に別名を与えている．また，Exp make_EAdd(Exp p0,Exp p1) は Haskell の構成子の部分に対応する部分を生成している．

```
int interpret(Exp _p_)
    {
     switch(_p_->kind)
     {
     case is_EAdd:
       return interpret(_p_->u.eadd_.exp_1) +
              interpret(_p_->u.eadd_.exp_2) ;
     case is_ESub:
       return interpret(_p_->u.eadd_.exp_1) -
              interpret(_p_->u.eadd_.exp_2) ;
     case is_EMul:
       return interpret(_p_->u.eadd_.exp_1) *
              interpret(_p_->u.eadd_.exp_2) ;
     case is_EDiv:
       return interpret(_p_->u.eadd_.exp_1) /
              interpret(_p_->u.eadd_.exp_2) ;
     case is_EInt:
       return _p_->u.eint_.integer_ ;
     }
   }
```

Haskell では case 式を用いていたが，C 言語では switch 文を用いている．if 文でも記述はできるが switch 文はこのような多分岐の条件による処理の場合に見やすく書くことが出来る．if 文は条件分岐を基本的に True，False で書いていくため，True，False の二分岐で条件分岐をしていくとき，使用するのに向いている．基本的に if 文と switch 文では if 文のほうが処理時間が短いことが多い．switch 文は処理回数が多いと処理時間が長くなるというデメリットがあるので処理時間を速くしたい場合などは if 文で記述するほうが良い．switch 文は enum のメンバで場合分けをしている．返り値は is_EAdd は足し算，is_ESub は引き算，is_EMul はかけ算，is_EDiv は割り算，is_EInt は整数値を返す．C 言語での switch 文でポインタを使用している．ポインタを用いることにより，アドレスを渡しているのでコピーをすることがなくメモリを節約できる．またコピーしないことにより実行時間を下げることが出来る．ポインタはプログラムが長くなればなるほど便利であるが，プログラムが短ければポインタを使わないほうが可読性が上がる場合もある．

4 おわりに

本研究では，BNF，代数型を用いて加減演算式を定義し，語句解析，構文解析，意味解析を行った．また，加減演算式を加減乗除演算式に拡張も行った．そして，加減乗除演算式に対して，別の意味解析を行った．さらに，Haskell と C 言語で算術式言語を実装するときの比較を行った．パーサーコンビネータ，構文の比較を行った．

参考文献

[1] 武市正人：プログラミング言語，岩波書店，1994.

[2] 大堀淳：アルゴリズムとプログラミング言語，岩波書店，1999.

[3] Almstrm Duregrd et al.：The BNF Converter, ⟨https://bnfc.digitalgrammars.com⟩ (accessed 2018-10-12) .