ソフトウェア工学 80－9
（1991. 7. 18）

Earley 型パーサに基づく代数仕様記述言語の構文解析

大蒔和仁† 戸村哲† 澤田寿実‡ 二木厚吉†

†電子技術総合研究所 ‡(株) SRA

OBJ のような代数仕様記述言語では、データである項 (term) の表現形式を規定する文脈自由文法を文脈に応じて利用者が指定できる。したがってそうした言語の構文解析には文脈自由文法用の構文解析木を文脈に応じて動的に生成する必要が生じる。我々は一般には実行効率が悪いとされている Earley の構文解析アルゴリズムが代数仕様記述言語の項の構文解析にどの程度利用可能かを検討したので、その結果を報告する。

Earley Parsing for Algebraic Specification Languages

Kazuhito OHMAKI† Satoru TOMURA† Toshimi SAWADA‡ Kokichi FUTATSUGI†

†Electrotechnical Laboratory (ETL)
1-1-4 Umezono, Tsukuba, Ibaraki 305, JAPAN
‡SRA, 1-1-4 Hirakawa-Cho, Chiyoda-ku, Tokyo 102

The syntax of terms in an algebraic specification language can be defined by users. We are experimenting to see whether it is feasible to use the Earley parser for the term parsing. The syntax analyzer for these terms have to make syntax trees using the grammar whose production rules are dynamically changing according to the context in a specification. The Earley parser is the most flexibe one to use for this type of the syntax analyzer. Although the Earley parser is inefficient to parse longer input strings, terms to be parsed are often shorter input strings in algebraic specifications.

1 まえがき

OBJ などに代表される多ソート代数あるいは順序ソート代数に基づく仕様記述言語では、構文解析すべき項 (term) の構文規則がプログラマによって定義される [1, 2, 3]。従って、このような言語の構文解析においては文脈自由文法用の構文解析木を文脈に応じて動的に生成する必要が生じる。

このような目的で用いる構文解析系では、文法規則を文脈により取り替えて解析を実行できなければならない。実行効率を無視すれば、このような目的の構文解析系としては、Earley による構文解析系 [4] が、最も汎用的で柔軟なものとして利用できる。文献 [5] においては、Earley による方法よりも効率が良いとされている LR 構文解析系を利用し、文脈に応じて構文解析表を変更する手法について考察している。

OBJ のような仕様記述言語構文解析すべき項はあまり大きくはない。従って、汎用の構文解析系の用途を OBJ の構文解析に限った場合には、Earley の方法を直接利用しても十分な効率が得られることが期待される。

このような動機に基づいて、我々は一般には実行効率が悪いとされている Earley の構文解析アルゴリズムが代数仕様記述言語の項の構文解析にどの程度利用可能かを検討している。

本稿では、我々が設計している Earley 構文解析系のインターフェースとなる関数を示す。そしてこの構文解析系を、OBJ の構文解析木から抽象構文を生成に用いた例を示す。

なお、本稿で示すシステムは Common LISP を用いて実現されている。なお文脈自由文法の構文解析に関する用語の説明は紙面の都合で省略した。

2 Earley 構文解析系

2.1 インターフェース

図 1に我々が作成した Earley 構文解析系の利用者に対するインターフェースの概念を示す。利用者のために構文解析系が用意しているサービス関数は、文法規則や語彙 (token) を登録削除する関数、構文解析すべき系列と初期記号とを与えて Earley 項集合を計算する関数、および Earley 項の集合から構文解析木を得る関数などからなる。

関数名	説明
RuleEntry	生成規則の登録
RuleDelete	生成規則の削除
RuleList	生成規則の表示
TokenEntry	語彙の登録
TokenDelete	語彙の削除
TokenList	語彙の表示
DelimitterEntry	区切り文字の登録
DelimitterDelete	区切り文字の削除
DelimitterList	区切り文字の表示

表 1: 生成規則操作のための主要関数

上記の「利用者」あるいは「サービス関数」という用語は、OSI(開放型システム間相互接続) の基本参照モデル [6] で用いられたインターフェース定義のための概念である。

表 1および表 2にそれぞれ文法定義および構文解析のための主なサービス関数を示す。

Earley 構文解析系を用いているため、構文解析において生成規則はその時点で系に登録されているものが用いられ、かつ初期記号も任意に選んで構文解析することができる。表 2における関数 ParseIt はそのためのものである。この関数により、構文解析系の内部には Earley 項 (item) の集合ができる。

関数 RTree により、この Earley 項の集合から右構文解析木が作成される。

2.2 実行例

次の BNF を Earley 構文解析系に与えた例を説明する。

```
E ::= T "+" E
    | T
T ::= F "*" T
    | F
F ::= "(" E ")"
    | "a"
```

この BNF を RuleEntry を使って次のように登録する。

```
>(RuleEntry '(E T "+" E)
            '(E T)
```

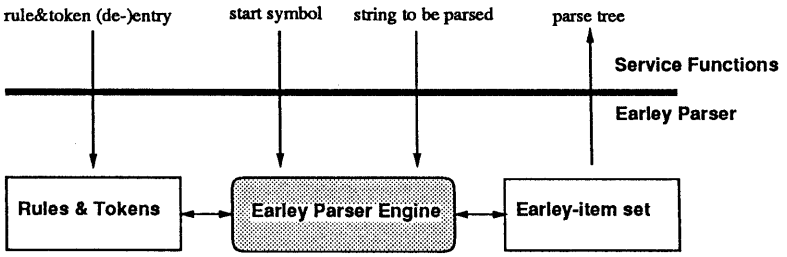

図 1: Earley パーザとそのサービス関数の関係

関数名	説明
ParseIt	文字列と初期記号とを与えて構文解析し Earley- 項集合を計算する。
RightParse	Earlye- 項集合から最右導出の生成規則列を生成する。
LeftParse	Earlye- 項集合から最左導出の生成規則列を生成する。
RTree	Earlye- 項集合から最右導出の構文解析木を S 式で生成する。
FirstSet	非終端記号を与えその First(1) 集合を計算する。
FollowSet	非終端記号を与えその Follow(1) 集合を計算する。
ShowIsland	Earley- 項集合を表示する

表 2: 構文解析のための主要関数

```
'(T F "*" T)
'(T F)
'(F "(" E ")")
'(F "a"))
```

関数 RuleList を用いて、今登録した生成規則を番号と共に表示する。

```
>(RuleList)

    (0) F -> "a"
    (1) F -> "(" E ")"
    (2) T -> F
    (3) T -> F "*" T
    (4) E -> T
    (5) E -> T "+" E
```

次に、初期記号を E として系列 "a*(a+a)" を、関数 ParseIt を用いて構文解析する。このときシステム内部では Earley 項の集合ができる。

```
>(ParseIt "a*(a+a)" 'E)

    Lexicon = (("a" A) ("*" OP)
               ("(" LPAR) ("a" A)
               ("+" OP) ("a" A)
               (")" RPAR))

    Earley Parsing complete Ended
```

変数 *pi* には右構文解析列 (最右導出を与える文法規則の番号列) を逆順に並べた系列が入っている。

```
>*pi*
(1 1 3 1 3 5 6 2 3 4 5 0)
```

次に関数 RTree により右構文解析木を表示する。これは変数 *pi* から生成することができる。

```
>(RTree)
(START (E (T (F "a") "*"
           (T (F "(" (E (T (F "a")) "+"
                      (E (T (F "a"))))
                  ")")))))
```

ここに、Earley 項の集合から右構文解析木を構成する方法および最右導出を求める方法は文献 [4] によっている。

また、Earley 項の集合を直接得る関数 ShowIsland を用意している。この関数は、構文解析エラーが生じた時にも、その時点までに生成された Earley 項の集合を返し、利用者がその情報を元にどのような構文解析エラーが生じたのかを調べることができる。

```
>(ShowIsland)

Island(0) :
    [F -> "" . ("a") ,0]
    [F -> "" . ("(" E ")") ,0]
    [T -> "" . (F) ,0]
    [T -> "" . (F "*" T) ,0]
    [E -> "" . (T) ,0]
    [E -> "" . (T "+" E) ,0]
    [START -> "" . (E) ,0]

Island(1) :
    [START -> (E) . "" ,0]
    [E -> (T) . "" ,0]
    [E -> (T) . ("+" E) ,0]
    [T -> (F) . "" ,0]
    [T -> (F) . ("*" T) ,0]
    [F -> ("a") . "" ,0]
   ...
Island(7) :
    [START -> (E) . "" ,0]
    [E -> (T) . "" ,0]
    [E -> (T) . ("+" E) ,0]
    [T -> (F "*" T) . "" ,0]
    [T -> (F) . "" ,2]
    [T -> (F) . ("*" T) ,2]
    [F -> ("(" E ")") . "" ,2]
```

3 OBJ の抽象構文木生成

前節で示した Earley 構文解析系を核として用いて OBJ のプログラムを抽象構文木に変換するシステムを作成した。

3.1 OBJ の説明

入力とする文字列は図 2の BNF で定義される OBJ のプログラムである。図 3に OBJ によるプログラム例として STACK を示す。

このプログラムの中で `subsort NeStack < Stack .` によりソート `NeStack` がソート `Stack` の部分ソートであることが宣言されている。このれにより、ソート `NeStack` の項はまたソート `Stack` の項でもあることが宣言されている。

```
<object> ::="obj" <obj name> "is"
            {<module element>}+ "endo"
<module element> ::= <sort decl>
                   | <subsort decl>
                   | <st op decl>
                   | <dist fix op decl>
                   | <var decl>
                   | <eq decl>
<sort decl> ::= "sort" {<sort name>}+ "."
<sub sort decl> ::= "subsort" <sort name>
                    {"<" <sort name>}+ "."
<st op decl> ::="op" {<op name>}+ ":"
              {<sort name>} "->" <sort name> "."
<dist fix op decl> ::="op" {<dist op form>}+ ":"
                     {<sort name>} "->" <sort name> "."
<dist of form> ::= [<op name>] "_" [ <op name>]
<var decl> ::= {<var name>}+ ":" <sort name> "."
<eq decl> ::= "eq" <term> "=" <term> "."
```

図 2: OBJ の BNF

また、op の宣言により、オペレータ `empty`、`push`、`pop`、および `top` の構文が定義されている。

抽象構文木の形は Lisp の関数呼び出しと同様のフォームである。例えば "1+1"、"(1+-1)*1" および "`if (1=1) then 1 else -1 fi`" を構文解析して得られる抽象構文木はそれぞれ

```
(+ 1 1)
(* (+ 1 -1) 1)
(if_then_else_fi (= 1 1) 1 -1)
```

となる。

3.2 システムの機能

図 4は、OBJ の BNF および OBJ プログラムを入力とし、OBJ プログラムの抽象構文木を出力とするシステムのブロック図を示す。破線より上がこのシステムの入出力である。

図で Rule Generator と書かれている部分が、OBJ の BNF 定義及び OBJ プログラムの `sort`、`subsort`、および `op` の各宣言を読み込んで文脈自由文法の生成規則

```
obj STACK is
  sort Elt .
  sort NeStack Stack .
  subsort NeStack < Stack .
  op empty : -> Stack .
  op push : Elt Stack -> NeStack .
  op pop : NeStack -> Stack .
  op top : NeStack -> Elt .
  var E : Elt .
  var S : Stack .
  eq pop(push(E, S)) = S .
  eq top(push(E, S)) = E .
endo
```

図 3: OBJ のプログラム例 STACK

を作り、Earley 構文解析系に規則の登録をする。各宣言から規則を生成する手順は次説で示す。

また Start Symbol Selector と書かれている部分は、OBJ のプログラムから構文解析の対象とする文字列及び初期記号を決定し、関数 `ParseIt` を用いて Earley 構文解析系に構文解析を行なわせる。

関数 `RTree` によって得られた構文解析木が次節の方法を用いた Abs Tree Generator により抽象構文解析木に変換される。

3.3 変換手順

3.3.1 Rule Generator の動作

図 4の Rule Generator における生成規則の登録方法は次の通りである。

まず、図 4における OBJ Original BNF を読み込んで Earley Parser に生成規則の登録を行なう。この BNF は具体的には図 5に示したように OBJ 自身で書かれている。これは、Rule Generator において BNF の読み込みと OBJ プログラムの読み込みとを共通のルーティンで行なわせるためである。また、この BNF の記述を用いることにより、BNF 自体の拡張も容易になるという副次効果もある。。

次に Rule Generator は、OBJ プログラムを読み込み、その中から `var` と `eq` と `cq` とを除く部分を用いて表 3の方法に従って生成規則を追加していく。ここに、

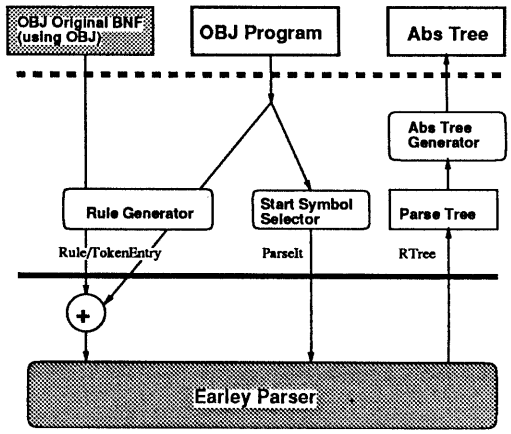

図 4: Earley 構文解析系を用いて OBJ の抽象構文木を作成する手順

```
obj _Native_Rule is
  op sort_\. : _Args -> _Exp .
  op subsort_<_\. : _SortName _SortName -> _Exp .
  op protecting_\. : _ObjName -> _Exp .
  op op_:_->_\. : _OpName _Args _SortName -> _Exp .
  op op_:->_\. : _OpName _SortName -> _Exp .
  op ops_:->_\. : _OpName _SortName -> _Exp .
  op var_:_\. : _VarNames _SortName -> _Exp .
  op eq_=_\. : _Exp _Exp -> _Exp.
  op cq_=_if_\. : _Exp _Exp _Exp -> _Exp .
  op obj_is_endo : _ObjName _Exps -> _ObjSrc .
endo
```

図 5: OBJ の構文を OBJ で書いたもの

```
>(objparse "stack.obj")
Loading native.rule
Finished loading native.rule
Earley parsing successfully finished.
((START
   ("obj_is_endo" "STACK" ("sort_." "Elt")
      ("sort_." "NeStack" "Stack")
      ("subsort_<_." "NeStack" "Stack")
      ("op_:_->_." "empty" "Stack")
      ("op_:_->_." "push"
                   "Elt" "Stack" "NeStack")
      ("op_:_->_." "pop" "NeStack" "Stack")
      ("op_:_->_." "top"  "NeStack" "Elt")
      ("var_:_." "E" "Elt")
      ("var_:_." "S" "Stack")
      ("eq_=_." ("pop" ("push" "E" "S")) "S")
      ("eq_=_." ("top" ("push" "E" "S")) "E"))))
```

図 6: STACK の抽象構文木

<term> は図 2の BNF の非終端記号であり、op 部分で構文が定義される項を構文解析するための初期記号となる。

この「生成規則の追加」の操作が図 4における + の部分に対応している。

この方法により追加された生成規則により <term> を初期記号としてある項を構文解析すると、各節点にソート名が付けられた（すなわちソート名を非終端記号とする）構文解析木が得られる。

3.3.2 Abs Tree Generator の動作

図 4における Abs Tree Generator は、前節のようにして得られた構文解析木を辿り、表 4に示す手順で抽象構文木を作成する。

3.4 実行例

前節までに示したシステムに、図 3で示した STACK の記述を入力した。その結果として図 6の抽象構文木を得た。

また、このとき構文解析に関与した生成規則や構文解析木および右構文解析は図 7のようになっている。

```
>(RuleList)

    (0) STACK -> "S"
    (1) ELT -> "E"
    (2) _OPNAME -> "top"
    (3) _OPNAME -> "pop"
    (4) _OPNAME -> "push"
    (5) _SORTNAME -> "Stack"
    (6) _SORTNAME -> "NeStack"
    (7) _SORTNAME -> "Elt"
    (8) _OBJNAME -> "STACK"
     ...
    (49) _OBJSRC -> "obj" _OBJNAME "is"
                          _EXPS "endo"

NIL

>(RTree)
(START (_OBJSRC
        "obj" (_OBJNAME "STACK") "is"
        (_EXPS (_EXP* (_EXP "sort"
                            (_ARGS
                            (_SORTNAME*
                            (_SORTNAME "Elt")
                            (_SORTNAME*)))
                           ".")
                      ...
        "endo"))

>*pi*
(8 7 34 35 36 40 6 5 34 35 35 36 40 6 5 41 24
 13 28 29 30 34 36 5 43 4 28 29 30 7 5 34 35
 35 36 6 43 3 28 29 30 6 34 35 36 5 43 2 28
 29 30 6 34 35 36 7 43 17 19 37 38 39 7 46 20
 22 37 38 39 5 46 1 0 25 26 14 0 14 47 1 0 25
 27 16 1 16 47 31 32 32 32 32 32 32 32 32 32
 32 32 33 49 9)
```

図 7: STACK の構文解析で用いられた生成規則や右構文解析

番号	OBJ のプログラム	対応する追加生成規則
1	無条件に	<term> ::= "(" <term> ")"
2	sort S .	<term> ::= S
3	subsort S_1 < S_2 .	S_2 ::= S_1
4	op f : S_1 S_2 ... S_{n-1} -> S_n .	S_n ::= "f" "(" S_1 "," S_2 ... "," S_{n-1} ")"
5	op f1_f2_...fn : S_1 S_2 ... S_{n-1} -> S_n .	S_n ::= "f1" S_1 "f2" S_2 ... S_{n-1} "fn"
6	var X1 X2 ... Xn : S .	S ::= "X1" | "X2" | ... | "Xn"

表 3: 生成規則の追加方法

番号	還元の生成規則	対応する抽象構文木の作り方
1	S_n ::= "f" "(" S_1 "," S_2 ... "," S_{n-1} ")"	("f" S_1 S_2 ... S_{n-1}) を S_n の抽象構文木とする。
2	S_n ::= "f1" S_1 "f2" S_2 ... S_{n-1} fn	("f1_f2_... _fn" S_1 S_2 ... S_{n-1}) を S_n の抽象構文木とする。
3	S_2 ::= S_1	S_1 の抽象構文をそのまま S_2 の抽象構文木とする。
4	S_n ::= S_1 S_2 ... S_{n-1}	("_" S_1 S_2 ... S_{n-1}) を S_n の抽象構文木とする。

表 4: 抽象構文木の作り方

4 むすび

OBJ のような言語に限った場合、構文解析すべき一つ一つの単位があまり大きくないので、項の構文解析系として Earley による方法でもあまり遅くはならないのではないかと考えている。今後はもっと大きな例題について本稿で示した変換を行ってみて妥当性を検討したい。

構文解析系に関する研究は、意味に関する研究に比べはるかに進んでいる。従って、言語処理系は出来るかぎり構文解析系の論理に乗せたシステムとし、意味に関する部分は言語に本質的な処理だけにしたい。

我々は従来から構文解析系を基本とするユーザインタフェース記述などを行なってきており、その有用性を確認してきた[7, 8]。

構造指向型の形式意味定義を用いた言語の定義が、実用規模の言語に対してもなされるようになってきている[9, 10]。この場合にもより柔軟な構文解析系が必要とされる。

参考文献

[1] K.Futatsugi, J.Goguen, J.-P.Jouannaud, and J.Meseguer, "*Principles of OBJ2,*" Proc. of 12th ACM Sympo. on POPL, pp.52-66, 1985.

[2] J.Goguen and T.Winkler, "*Introducing OBJ3,*" Technical Report, SRI-CSL-88-9, August 1988.

[3] H.Ehrich and B.Mahr, "*Fundamentals of Algebraic Specification 1,*" Springer Verlag, Berlin, 1985.

[4] A.V.Aho and J.D.Ullman, "*The Theory of Parsing, Translation, and Compiling,*" Prentice-Hall, Inc., Vol.1, 1972.

[5] J.Heering, P.Klint, and J.Rekers, "*Incremental Generation of Parsers,*" Proc. of the SIGPLAN '89 Conference on Programming Language Design and Impelemetation, SIGPLAN Notices, Vol.24, No.7, pp. 179-191, 1989.

[6] ISO 7498, "*Information Processing Systems - Basic Reference Model for Open System Interconnection,*" 1983.

[7] 戸村, 保科, 大蒔, 二木, "拡張文法に基づくユーザインタフェース記述法", 電子情報通信学会論文誌, Vol.J70-D, No.11, pp.2259-2266, 1987.

[8] K.Ohmaki and J.Stelovsky, "*An Implementation of Scrolling and Partial REfreshing Operations of a Program Editor for Modula-2,*" Trans. IEICE, Vol.E71, No.11, pp.1146-1156, 1988.

[9] ISO 8807, "*Information Processing Systems - Open System Interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour,*" 1989-02-15.

[10] R.Milner, M.Tofte, and R.Harper, "*The Definition of Standard ML,*" The MIT Press, 1990.