ソフトウェア基礎論　16－7
プログラミング言語　4－7
（1986. 2. 7）

Experimental Generation of An NBSG/PD Pre-Compiler by the Language Processor Generator MYLANG

Takashi Yamanoue* Hiroyuki Anzai+ Sho Yoshida*
Toshiyuki Sugio@ Atushi Takeuchi@ Tsutomu Shiino@

*Kyushu University +Kyushu Institute of Technology
@OKI Electric Industry Company,Ltd.

Abstract

An NBSG/PD pre-compiler is a translator which transforms one kind of Japanese programs specifically the NBSG/PD programs into C programs. The NBSG/PD has limited vocabulary, plain syntax and a mechanism for using information of the required specification which is defined by the Japanese Based Specification Language NBSG. We have developed a subset of NBSG/PD pre-compiler using the language processor generator MYLANG. MYLANG transforms Extended Attributed Regular Translation Forms(EARTFs) into Extended Attributed Syntax Directed Translators (EASDTs). The EARTF can define entire parts of language processors.

1. Introduction

An NBSG/PD[2] is a Japanese programming language which is developed in order to describe procedures of the functional elements. These functional elements are extracted from the Japanese specification language NBSG[1].

The NBSG/PD has limited vocabulary, plain syntax and a mechanism for using information of the required specification which is defined by the NBSG.

We have developed the subset of NBSG/PD pre-compiler by the language processor generator MYLANG. The NBSG/PD pre-compiler is a translator which transforms NBSG/PD programs into C programs.

MYLANG is a translator which transforms the Extended Attributed Regular Translation Forms(EARTFs) into Extended Attributed Syntax Directed Translator (EASDTs). EARTF is the Attributed Regular Translation Form(ARTF) with the notation of RAM(Random Access Memory) manipulating functions. EARTF can be used not only for defining of language processors but also for defining software models of conventional programs. According to Watt's paper[3], "Attribute grammars and affix grammars are currently the most promising tools available in the design of compiler writing systems." On the other hand, previous compiler writing systems which use attribute grammars, are adopting λ-notation or the

languages such as LISP, Pascal or C etc. in order to describe semantic functions of its attribute grammars. So, users of these compiler writing systems have to acquire plural schema, for example, scheme of lexical analyzer, scheme of syntax analyzer and scheme of semantic functions of the attribute grammar etc..

In this paper, unification of the above-mentioned schema is shown, using the Extended Attributed Regular Translation Form(EARTF). EARTF is a kind of scheme of attributed grammars. The EARTF can define semantic functions of attributed grammars by itself. The Extended Attributed Regular Translation Form(EARTF) is the Attributed Regular Translation Form(ARTF) with the notation of RAM manipulating functions.

The Attributed Regular Translation Form(ARTF) is an augmented form of extended BNF. The vocabulary of ARTF consists of not only terminal and nonterminal symbols but also action symbols and furthermore attributes may be attached to these symbols.

Manna has exploited the regular expressions in order to prove the isomorphism problem of program schema[5]. Jackson has suggested one method of software design, which is known as the Jackson method[6]. In this method, structures of programs are expressed by regular expressions. Shaw has suggested flow expressions[7]. These are regular expressions with shuffle products in order to express concurrent process. Katayama has shown that attribute grammars can be used for calculation models[8].

ARTF is a scheme which employs regular expressions and is augmented with the attribute grammar. These features makes it possible for ARTF to define software models and the calculation systems by ARTF.

A comparison between the Turing machine[10], which can be described as "A finite state automaton + Infinite length's tape" and the idea contained in the title of Wirth's book "Algorithms +Data Structures = Programs"[9], reveals that they are quite similar. In EARTF, ARTF corresponds to Algorithms, and RAM manipulation functions corresponds to Data Structures. Using EARTF, we can define semantic functions of the attribute grammar and software models of conventional programs. The language processor generator MYLANG transforms EARTFs into language processors or object programs. Comparison of MYLANG to YACC[11] is shown by Fig.1.

	Lexical	Syntax	Semantics
YACC	Lex	BNF	C
MYLANG	EARTF	EARTF(EBNF)	EARTF

Fig. 1. Comparison between YACC and MYLANG.

2. The Extended Attributed Regular Translation Form

The Extended Attributed Regular Translation Form(EARTF) is the Attributed Regular Translation Form(ARTF) with the notation of RAM manipulation functions. ARTF is an augmented form of Extend BNF. The Extended BNF which is the basis of ARTF, is as shown by the following example. This example is the syntax defining simple arithmetic expressions.

```
<e> = <t> ( '+' <t> )*
<t> = <f> ( '*' <f> )*
<f> = <id> + '(' <e> ')'
```

Fig. 2. An example of Extended BNF

Where the objects <e> and '+' are a nonterminal symbol and a terminal symbol, respectively. Namely, for any string S, <S> is regarded as a nonterminal symbol, and 'S' as a terminal symbol. The two meta-symbols "=" and "+" are used in the same context as "::=" and "|" in BNF respectively. Similarly, the meta-symbols "(", ")", "+" and "*" are used in the same context as in regular expressions.

Before explaining ARTF in detail, an example is shown below.

```
<e(/x)> = <t(/x)> ( '+' <t(/x1)> [(x:=x+x1)] )* ;
<t(/x)> = <f(/x)> ( '*' <f(/x1)> [(x:=x*x1)] )* ;
<f(/x)> = <id(/n)>[lookup(n/x)] + '(' <e(/x)> ')' ;
```

Fig. 3 An example of ARTF: definition of arithmetic calculator by ARTF

The example in Fig. 3 is an ARTF defining the syntax and semantics of arithmetic calculator. This illustrates the attributed syntax-directed translation of simple arithmetic expressions. ARTF is a kind of the attributed syntax-directed translation scheme.

As shown by the example in Fig. 3, ARTF consists of one or more equations which are terminated by a meta-symbol ";". For a sequence of strings S, [S] is called an action symbol. [lookup(n/x)] in Fig. 3. is an action symbol and has "lookup" as its name.

The nonterminal symbol <e(/x)> shows that the symbol has an attribute. This is done by the attribute occurrence variable(name) "x". Attributes can be attached to nonterminal and action symbols in the following manner:

$\langle N(i_1, i_2, \ldots, i_m / s_1, s_2, \ldots, s_n) \rangle$

$[A(i_1, i_2, \ldots, i_m / s_1, s_2, \ldots, s_n)]$

where, N is a nonterminal symbol name and A is an action symbol name. Each i_x where $1 \leq x \leq m$, shows an occurrence of an inherited attribute of the symbol. Similarly, each s_y where $1 \leq y \leq n$, shows an occurrence of a synthesized attribute of the symbol. An attributes of the nonterminal symbol or action symbol is identified by the position of its attribute occurrence.

It is possible to describe an assignment of arithmetic expression in an action symbol directly , such as [(x:=x+x1)],[(x:=x*x1)]. Fig. 4 is an ASDT which is defined by Fig. 3.

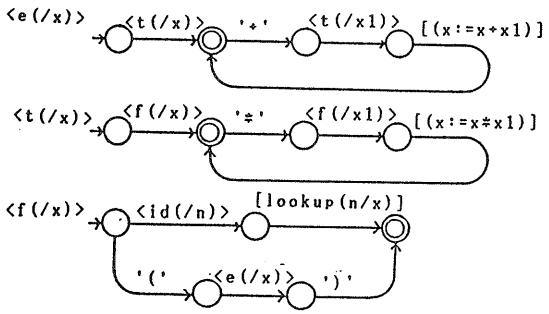

Fig. 4 ASDT which is transformed from Fig. 3

ARTF can also be used to describe predicates. Using the predicates, it is possible to define context-sensitive languages by the ARTF. Fig. 5 is the ARTF which defines a recognizer of the language $A^nB^nC^n$. $A^nB^nC^n$ is a famous example of context-sensitive language. Here, action symbols [?(i=j)] and [?(j=k)] are predicates. A symbol $[?(p(x_1..x_n))]$ means that if $p(x_1..x_n)$ is true then the symbol becomes λ(empty string), and otherwise it is ϕ(empty). In this example, attribute occurrences i,j,k means number of A, number of B, and number of C respectively.

[?(i=j)][?(j=k)] is λ only if i=j=k and otherwise the expression becomes ϕ. Thus the language $A^iB^jC^k=A^nB^nC^n$ where $n \geq 0$ is defined in Fig.5.

```
<s>=[(i:=0)]( 'A'[(i:=i+1)] )*
    [(j:=0)]( 'B'[(j:=j+1)] )*
    [(k:=0)]( 'C'[(k:=k+1)] )*
    <t(i,j,k/)> ;
<t(i,j,k/)>=[?(i=j)][?(j=k)] ;
```

Fig. 5 ARTF which defines the context-sensitive language $A^nB^nC^n$.

The Extended Attributed Regular Translation Form(EARTF) is ARTF with the notation of RAM (Random Access Memory) manipulating functions. RAM is manipulated

by semantic functions which are denoted by action symbols. A RAM manipulating function is a load or a store instruction.

A load instruction is denoted by [(x:=[arithmetic-expression])]. A store instruction is denoted by [([arithmetic-expression]:=x)].

We have defined list manipulation functions and string manipulation functions by EARTF as shown in Fig. 6,7. These functions play an important role in developing language processors.

```
<cons(x,y/z)>=[([#list]:=x,[[#list]+1]:=y,z:=[#list],
               [#list]:=[#list]+2)] ;
<car(x/y)>=[?(x>#list)][(y:=[x])] ;
<cdr(x/y)>=[?(x>#list)][(y:=[x+1])] ;
<atom(x/)>=[?(x<#list)] ;
<null(x/)>=[?(x='nil')] ;
```

Fig. 6 Basic list manipulation functions defined by EARTF.

```
<strcpy(i,j/)>=[([j]:=[i])]
               ( [?([i]<>0)][(i:=i+1,j:=j+1,[j]:=[i])] )* ;
<eqstr(i,j/)>=[?([i]=[j])]
              ( [?([i]<>0)][(i:=i+1,j:=j+1)][?([i]=[j])] )* ;
```

Fig. 7 String manipulation functions defined by EARTF.

In Fig.7, <strcpy(i,j/)> is a function which copy a string from i to j. <eqstr(i,j/)> is a predicate which is $\perp$ if string i and j are the same or else ϕ.(i and j are initial addresses of strings in RAM.)

3. Standard Action Routines and A Tracer in MYLANG(MS-DOS version)

MYLANG is equipped with standard action routines which look ahead more than one character, backtrack input strings and manipulate sequential files etc.. These routines can be called by reserved action symbols which we call standard action symbols. Standard action symbols are denoted by the symbol [.a]. MYLANG is also equipped with a tracer as debugging environment.

3.1 Looking ahead more than 1 character

The standard action symbol [.tch(x/y)] means get the character y which is ahead of x characters. The standard action symbol [.gch(x/)] means abandon x characters without reading.

3.2 Backtracking

The standard action symbol [.gbp(/w)] means memorize the input string in w, and the standard action symbol [.back(w/)] means let the input string as w. w is a pointer to the input buffer. Combining these two action symbols, users

can backtrack input strings explicitly(not automatically). Some other versions of MYLANG are able to backtrack automaticaly.

3.3 File input/output

The standard action symbol [.open(fid,rw/fn)] means open the read or the write(rw) file with file number fn and with file name fid. The standard action symbol [.close(fn/)] means close the file fn. The standard action symbol [.cinf(fn/)] means let the current input file as fn. The standard action symbol [.coutf(fn/)] means let the current output file as fn. The standard action symbol [.tapeout(x/)] means output the string x to the current output file. fid, rw and x are initial addresses of strings in RAM.

3.4 Tracer

Fig. 8 is an example of tracing LISP interpreter which is defined by EARTF. Two kinds of brake points(nonterminal symbols and an address of RAM) can be given by users.

```
lisp
n)ormal or t)race ?   n
 b)reak1 ? b

  non_terminal symbol name ? cons
 cons ... y)es ? y
..s
..iniatmtab
..inilist
..lisp
..reads
..atomeval
..defun
..eval
..prints
..atom
..car
..cadr
..cddr
..ok  brk_no.=13
 b)reak2 ?
 b)reak(ram) ?
c)onsole or f)ile ? c
```

```
>(car '(a b c))
<cons>
n)ormal or t)race ?   t

begin cons : '(a b c))
  < pop (  1 :0 )
  < pop (  0 :6 )
  2001=[2000]
  > push ( 0 :6 )
  [2001]:=6
  2001=[2000]
  > push ( 1 :0 )
  [2002]:=0
  2001=[2000]
  < pop (  2 :2001 )
  2001=[2000]
  [2000]:=2003
  > push ( 2 :2001 )
<cons> end
n)ormal or t)race ?   n
<cons>
n)ormal or t)race ?   n
<cons> end
```

Fig.8 Tracing Lisp interpreter

4. An NBSG/PD Pre-Compiler

An NBSG/PD is a Japanese programming language which is developed in order to describe procedures of the functional elements. These functional elements are extracted from the Japanese specification language NBSG.

An NBSG/PD pre-compiler is a translator which transforms NBSG/PD programs into C programs. The NBSG/PD pre-compiler consists of two processors (a lexical analyzer and a syntax-semantics analyzer).

4.1 Examples of NBSG/PD programs

Fig. 9 is an NBSG/PD program which performs sorting. NBSG/PD programs consist of three kinds of control expressions: sequences, selections and iterations. We will explain these control expressions using regular expressions.

If a and b are statements of NBSG/PD program, "a b" is also a statement of an NBSG/PD program which expresses "ab" in the regular expression.

If $a_i (1 \leq i \leq n-1)$ is a conditional expression and $b_j (1 \leq j \leq n)$ is a statement then the statement

```
"ここで，a1 のとき，b1
        a2 のとき，b2
          .........
        その他のとき，bn
 以上。"
```

is also a statement of the NBSG/PD program which expresses

$$"a_1b_1+a_2b_2+\ldots+\text{else } b_n"$$

in the regular expression, i.e.,

$$"\text{if } a_1 \text{ then } b_1, \text{ else if } a_2 \text{ then } b_2, \ldots \text{ else } b_n".$$

If a is a conditional expression and b is a statement,

```
"a の間、次のことを繰り返す。 b 繰り返し終り。" and
"a となるまで、次のことを繰り返す。 b 繰り返し終り。"
```

are statements of the NBSG/PD program which express

$$"(ab)^*" \quad \text{and} \quad "(\neg ab)^*"$$

in regular expressions respectively, i.e.,

"while a do b" and "while not a do b".

```
type sort.ntb
ﾓｼﾞｭｰﾙ main()
{
    INT ﾎﾞｯｸｽ ;
    INT 配列ポインタ，比較ポインタ ;
    STATIC INT 配列[20]={100.43.60.37.60.93.20000.22.
                         20.12453.9870.2.77.221.546.37.
                         9999.42.38.711} ;
    配列ポインタ = 0.
    (配列ポインタ != 20)の間，次のことを繰り返す。
    {
        比較ポインタ = 配列ポインタ.
        (比較ポインタ != 20)の間，次のことを繰り返す。
        {
            比較ポインタ = 比較ポインタ + 1.
            ここで，
                配列[配列ポインタ] > 配列[比較ポインタ]のとき，
                {
                    ﾎﾞｯｸｽ = 配列[配列ポインタ].
                    配列[配列ポインタ] = 配列[比較ポインタ].
                    配列[比較ポインタ] = ﾎﾞｯｸｽ.
                }
                その他のとき，
            以上.
        }
        繰り返し終り.
        書式出力("%dYn".配列[配列ポインタ])を実行する。
        配列ポインタ = 配列ポインタ + 1.
    }
    繰り返し終り.
}
//
```

Fig.9 A Sorting program in NBSG/PD

4.3 A Syntax-Semantics Analyzer

A syntax-semantics analyzer of the NBSG/PD pre-compiler transforms the intermediate programs into C programs. Fig.12 is a part of syntax-semantics analyzer defined by EARTF. The nonterminal symbol <branch(/x)> defines translation of branch statements in NBSG/PD into "if" statements in C language. The nonterminal symbol <iter(x/y)> defines translation of iterate-statements in NBSG/PD into "while" statements in C language. Fig.13 is the C program which is transformed from Fig.11 by the syntax-semantics analyzer.

```
<branch(/x)> =<ここで><ten><notokilis(/x)>
                 'その他のとき'<sonota(x/)>
               '以上'<maru> :
    <ここで>='ここて゛'+'ここで':
  <notokilis(/x)>=<notoki(/x)>(<notoki2(/x1)><nconc(x.x1/)>)* :
  <notoki(/x)>=  <eexpg(/a)>'のとき'<ten><stmt(/b)><pea(a.b/x)> :
  <notoki2(/x)>= <notoki(/y)><list('else'.' '/x)><nconc(x.y/)> :
  <sonota(x/)>=  <ten>
       {<stmt(/y)><list('else'.' '/z)><nconc(z.y/)><nconc(x.z/)>)/ :

  <pea(x.y/z)>=<sand('('.x.')'/x)><list('if'.' '/z)>
                                  <nconc(z.x/)><nconc(z.y/)> :

  <iter(x/y)>=            <iter2(x/y)> :

  <iter2(x/y)>=<noaida(/k)>'次のことを''繰り返す'<maru><stmts(/y)>
                 '繰り返し''終り'<aiter2(k.x.y/y)> :
  <noaida(/k)>='の間'<ten>[(k:=1)] + <となるまで><ten>[(k:=2)] :
    <となるまで>='となるまて゛'+'となるまで' :

  <aiter2(k.x.y/x)>=
       ([?(k<>1)]<sand('('.x.')'/x)><cons('!'.x/x)>)/
       <forhead(x/x)><sand('{'.y.'}'/y)><nconc(x.y/)> :

  <forhead(x/x)>=<sand('('.x.')'/x)><cons('while'.x/x)> :

  <sand(a.x.b/x)>=
      <cons(a.x/x)><cons(b.'nil'/b)><nconc(x.b/)> :
```

Fig.12 A part of the syntax-semantics analyzer which is defined by EARTF

```
main(){int n43.
int n45.n47:
static int n49[20]={100.43.60.37.60.93.20000.22.20.12453.9870.2.77.221.546.37.99
99.42.38.711}
:
n45=0:
while((n45!=20)){n47=n45:
while((n47!=20)){n47=n47+1:
if (n49[n45]>n49[n47]){n43=n49[n45]:
n49[n45]=n49[n47]:
n49[n47]=n43:
}
}
}
printf("%d¥n".n49[n45]):
n45=n45+1:
}
}
}
```

Fig.13 The C program which is transformed from Fig.11

4.2 A Lexical Analyzer

An lexical analyzer of the NBSG/PD pre-compiler transforms NBSG/PD programs into intermediate programs. The lexical analyzer just distinguishes reserved symbols and other names. Reserved symbols are passed through the analyzer without change. Other names are sandwiched between "n(" and ")". Fig.10 is a part of the lexical analyzer which is defined by EARTF. Fig.11 is the intermediate program which is transformed from Fig.9 by the lexical analyzer.

```
<lex>=<inilex><tokens><wln><wln>  :

<tokens>=[(b:=#eof)]  ( <neof(b/)><read(/a.b)><wtokena(a/)><tapeoutb(b/)> )* :

<read(/#w.#b)>=<inistr(#w/)><inistr(#b/)> (<strconst>+else<nx>*) <re> :

<nx>=[.tch(0/x)]<nresc><chrcat(#w.x/)>[.gch(1/)] :

<nresc>=[.gbp(/x)] (<resc>[.back(x/)][?(1=2)])/ :

<re>=<cln><resc><con>[.nametape(#b/)] :

<resc>=<symx>+else(<kstr>+<astr>) :
<symx>=<symx1>+else<symx2> ;
<symx1>=<symx12>+<symx13>+<symx14>+<symx15> :
    <symx12>=´ ´ + ´.´ + ´.´ + ´.´ :
    <symx13>=´:´ + ´:´ + ´~´ + ´^´ :
```

Fig.10 A part of the lexical analyzer which is defined by EARTF

```
E>type sort.trm
ﾓｼﾞｭｰﾙ n(main)()「      INT  n(ﾎﾟｯｸｽ) :
      INT  n(配列ポインタ), n(比較ポインタ) :
      STATIC INT  n(配列)[20]={100.43.60.37.60.93.20000.22.
      20.12453.9870.2.77.221.546.37.                    9999.42.38.711} :
      n(配列ポインタ) = 0.
      (n(配列ポインタ) != 20)の間，次のことを繰り返す。
      「        n(比較ポインタ) = n(配列ポインタ).
           (n(比較ポインタ) != 20)の間，次のことを繰り返す。
           「          n(比較ポインタ) = n(比較ポインタ) + 1.
                ここで，                    n(配列)[n(配列ポインタ)] > n(配列)[n(
比較ポインタ)]のとき，                    「                  n(ﾎﾟｯｸｽ) =
 n(配列)[n(配列ポインタ)].
                          n(配列)[n(配列ポインタ)] = n(配列)[n(比較ポインタ)].
                          n(配列)[n(比較ポインタ)] = n(ﾎﾟｯｸｽ).
                     」
                     その他のとき，          以上.
           」
           繰り返し終り.
           n(書式出力)("%d¥n", n(配列)[n(配列ポインタ)])を実行する。
           n(配列ポインタ) = n(配列ポインタ) + 1.
      」
      繰り返し終り.
」
//
```

Fig.11 A intermediate program which is transformed from Fig.9

5. Conclusion

We have developed a subset of the NBSG/PD pre-compiler by the language processor generator MYLANG. Entire parts of the pre-compiler are defined by EARTF. Both the NBSG/PD pre-compiler and MYLANG are implemented on the personal computer IF-800 model 50/60.We are presently developing the full set of the NBSG/PD pre-compiler. The full set of the NBSG/PD includes enumeral types, subrange types, block transference mechanisms, object oriented style and a concurrent mechanism etc.. And we are planning to develop software development assistance environment of NBSG and NBSG/PD.

In order to develop language processors and other softwares much easier, we are planning to develop much better programming environments of MYLANG and to extending EARTF to abstract data types and concurrent descriptions etc.. These softwares can be defined by EARTF and generated by MYLANG.

References

[1] Shiino,T.,Takeuchi,A.,Sugio,T.:"A Specification-Describing Language NBSG which is based on Japanese",WGSE,30-2(1983)(In Japanese).

[2] Shiino,T.,Takeuchi,A.,Sugio,T.:"Procedure Description on a Specification-Describing Language NBSG which is based on Japanese",WGSE,43-13(1984) (In Japanese).

[3] Watt,D.A. :"Rule Splitting and Attribute Directed Parsing,"In:Semantic-Directed Compiler Generation, Lecture Notes in Computer Science 94, Spring-Verlag,pp.363-392(1980).

[4] Knuth. D.E.:"Semantics of context-free languages," Math.System Theory, Vol.2, pp.127-145(1968).

[5] Manna,Z.:"Program Schema," In:Aho(ed.) Currents in the Theory of Computing, Prentice Hall(1973).

[6] Jackson,M.A.:"Principles of Program Design," Academic Press (1975).

[7] Shaw,A.C.:"Software Specification Languages Based on Regular Expressions," Technical Report, ETH Zurich(1979).

[8] Katayama,T.:"HFP:A Hierachical and Functional Programming Based on Attribute Grammar," 5th International Conference on Software Engineering, 343 (1981).

[9] Wirth,N.:"Algorithms+Data structures=Programs," Prentice-Hall (1976).

[10] Turing,A.M.:"On Computable Numbers, with an Application to the Entsheidungsproblem," Proc. London Math. Soc.(2),42,pp.230-265 (1937).

[11] Johnson,S.C.:"YACC-Yet Another Compiler-Compiler," CSTR 32, Bell Laboratories(1975).

[12] Anzai,H.:"A theory of recursive descent translator generator," Proc. of the Int'l Comp. Symp. ICS80-II,pp.1171- 1182 (1980).

[11] Lewis,P.M.,Rosenkrantz,K.J. and Sterns,R.E.:"Attributed Translations," J. Computer and System Sciences 9,pp.279-307 (1974).

[12] Anzai,H. Sugibayashi,N. and Yamanoue,T.:"Experimental Generation of A Prolog Interpreter by MYLANG," Proc. of the Logic Programming Symposium '84,5-2,Tokyo(1984).