

属性文法評価法の効率化について

徳田雄洋 藪 正樹
(山梨大学 工学部)

"Methods for Transforming Attribute Grammars into Efficient Action Routines" (in English)
by Takehiro TOKUDA and Masaki YABU (Department of Computer Science, Yamanashi University Takeda, Kofu 400, Japan)

This note gives methods for transforming attribute grammars into efficient action routines. An action routine description is a set of fragments of programs associated with production rules. Those fragments of programs are activated according to the ordering given by a syntax analyzer.

We present transformation methods based on following techniques: asynchronous stack introduction patch operation introduction, bypassed transformation, and initial value introduction. We illustrate the improvement of efficiency using problems such as translation of arithmetic expressions and Boolean expressions into intermediate codes.

METHODS FOR TRANSFORMING ATTRIBUTE GRAMMARS
INTO EFFICIENT ACTION ROUTINES

Takehiro Tokuda and Masaki Yabu
Department of Computer Science
Yamanashi University
Takeda, Kofu 400, Japan

1. Introduction

The purpose of this note is to show that we can systematically transform some type of attribute grammars into efficient action routines. An action routine description is a set of fragments of programs associated with production rules. Those fragments of programs are activated according to the ordering given by a syntax analyzer. Action routines use both local and global memory space efficiently.

Executional disadvantage of attribute grammars comes from the fact that locality of descriptions, freedom of attribute dependency, and absence of the knowledge of the ordering of syntax analysis. We refer to [2-5, 7-12, 14-17] for some of existing implementation methods. Those methods use various assumptions

about attribute dependency and syntax analysis ordering.

Traditional evaluation methods of attribute grammars tend to use redundant memory space or redundant operations. Recent methods try to translate the descriptions into sets of procedures, or minimize memory space using global variables. Those approaches achieve improvement of efficiency by usually treating the subject as an implementation problem of procedure parameter passing, or a combinatorial minimization problem [12, 14-17].

Our goal is to present methods for transforming attribute grammars into efficient action routines naturally and mechanically.

2. Transformation Methods

The simple postfix transformation scheme [1] is a classical example of transformation of attribute grammars into efficient action routines. We give an example of such transformation in Fig. 2.1 and 2.2. This example shows a translation of infix arithmetic expressions into postfix arithmetic expressions. Note that the attribute code is a postfix synthesized attribute and no variable is used in Fig. 2.2 at all.

Fig. 2.1. Infix-to-postfix translation

```

0) E' -> E
1) E -> E + T
   E1.code := E2.code + T.code + "+"
2) E -> E - T
   E1.code := E2.code + T.code + "-"
3) E -> T
   E.code := T.code
4) T -> T * F
   T1.code := T2.code + F.code + "*"
5) T -> T / F
   T1.code := T2.code + F.code + "/"
6) T -> ( E )
   T.code := E.code
7) F -> i
   F.code := lex(i)

```

Fig. 2.2 Infix-to-postfix translation

```

0) E' -> E
1) E -> E + T
   print("+")
2) E -> E - T
   print("-")
3) E -> T
4) T -> T * F
   print("*")
5) T -> T / F
   print("/")
6) T -> ( E )
7) F -> i
   print(lex(i))

```

Transformation methods we present in this note are as follows.

- 1) Asynchronous_stack_introduction deals with cases where attributes are only synthesized, and eliminate redundant copy operations involving unit productions.
- 2) Patch_operation_introduction eliminates some type of inherited attributes, and introduces patch operations.
- 3) Bypass_transformation deals with general non-circular attribute grammar cases, and eliminates redundant copy operations involving unit productions.
- 4) Initial_Value_Introduction reduces the number of variables in action routines.

3. Asynchronous Stack Introduction

Asynchronous stack introduction consists of two phases. We first eliminate postfix synthesized attributes by simple postfix transformation. Then we introduce asynchronous stacks for each attribute, and eliminate redundant copy operations involving unit productions [19]. In Fig. 3.1-3.4 we consider a problem translation of arithmetic expressions into quadruples

Fig. 3.1. Starting attribute grammar description

```

0) E' -> E
1) E -> E + T
   E'.place := E.place
   E'.code := E.code
   E1.place := newtemp()
   E1.code := E2.code + T.code
   + ("+", E2.place, T.place, E1.place)
2) E -> E - T
   E1.place := newtemp()
   E1.code := E2.code + T.code
   + ("-", E2.place, T.place, E1.place)
3) E -> T
   E.place := E.place
   E.code := E.code
   E.place := T.place
   E.code := T.code
4) T -> T * F
   T1.place := newtemp()
   T1.code := T2.code + F.code
   + ("*", T2.place, F.place, T1.place)
5) T -> T / F
   T1.place := newtemp()
   T1.code := T2.code + F.code
   + ("/", T2.place, F.place, T1.place)
6) T -> F
   T.place := F.place
   T.code := F.code
7) F -> ( E )
   F.place := E.place

```

```

F.code := E.code
8) F -> i
   F.place := lex(i)
   F.code := ""

```

Using the simple postfix transformation scheme, we eliminate occurrences of attribute code as in Fig. 3.2.

Fig. 3.2. A description without code

```

0) E' -> E
   E'.place := E.place
   print("")
1) E -> E + T
   E1.place := newtemp()
   print("+", E2.place, T.place, E1.place)
2) E -> E - T
   E1.place := newtemp()
   print("-", E2.place, T.place, E1.place)
3) E -> T
   E.place := T.place
   print("")
4) T -> T * F
   T1.place := newtemp()
   print("*", T2.place, F.place, T1.place)
5) T -> T / F
   T1.place := newtemp()
   print("/", T2.place, F.place, T1.place)
6) T -> F
   T.place := F.place
   print("")
7) F -> ( E )
   F.place := E.place
   print("")
8) F -> i
   F.place := lex(i)
   print("")

```

Using the asynchronous stack introduction, we can replace occurrences of attribute place by pop and push operations with respect to a stack PLACE as in Fig. 3.3.

Fig. 3.3. An action routine description

```

0) E' -> E
    pop(PLACE,op)
    t := op
    push(PLACE,t)
    print("")

1) E -> E + T
    pop(PLACE,op2)
    pop(PLACE,op1)
    t := newtemp()
    push(PLACE,t)
    print("+", op1, op2, t)

2) E -> E - T
    pop(PLACE,op2)
    pop(PLACE,op1)
    t := newtemp()
    push(PLACE,t)
    print("-", op1, op2, t)

3) E -> T
    pop(PLACE,op)
    t := op
    push(PLACE,t)
    print("")

4) T -> T * F
    pop(PLACE,op2)
    pop(PLACE,op1)
    t := newtemp()
    push(PLACE,t)
    print("*", op1, op2, t)

5) T -> T / F
    pop(PLACE,op2)
    pop(PLACE,op1)
    t := newtemp()
    push(PLACE,t)
    print("/", op1, op2, t)

6) T -> F
    pop(PLACE,op)
    t := op
    push(PLACE,t)
    print("")

7) F -> ( E )
    pop(PLACE,op)
    t := op
    push(PLACE,t)
    print("")

8) F -> i
    push(PLACE,lex(i))

```

print("")

By eliminating redundant action sequences, we obtain the final action routine description in Fig. 3.4.

Fig. 3.4. Final action routine description

```

0) E' -> E
1) E -> E + T
    pop(PLACE,op2)
    pop(PLACE,op1)
    t := newtemp()
    push(PLACE,t)
    print("+", op1, op2, t)

2) E -> E - T
    pop(PLACE,op2)
    pop(PLACE,op1)
    t := newtemp()
    push(PLACE,t)
    print("-", op1, op2, t)

3) E -> T
4) T -> T * F
    pop(PLACE,op2)
    pop(PLACE,op1)
    t := newtemp()
    push(PLACE,t)
    print("*", op1, op2, t)

5) T -> T / F
    pop(PLACE,op2)
    pop(PLACE,op1)
    t := newtemp()
    push(PLACE,t)
    print("/", op1, op2, t)

6) T -> F
7) F -> ( E )
8) F -> i
    push(PLACE,lex(i))

```

4. Patch Operation Introduction

Patch operation introduction eliminates some type

of inherited attributes. Patch operation introduction consists of two phases. First we reverse the flow of inherited attributes, and treat those inherited attributes as synthesized. Then we introduce patch operations at the point where the value is determined.

We consider a problem of translating a Boolean expression into quadruples of short circuit evaluation form. We, for example, translate a Boolean expression below

(A or B) and (C or D)

into the corresponding quadruples as follows. A quadruple (:=, val, Z) stands for the assignment of a value to a variable Z. Values 1 and 0 respectively stand for true and false. A quadruple (Br, VAR, i, j) stands for branch to address i, if VAR is true, and branch to address j, if VAR is false.

```

-----
1 | := 1      Z |
2 | Br A     4 3 |
3 | Br B     4 6 |
4 | Br C     7 5 |
5 | Br D     7 6 |
6 | := 0      Z |
7 | -----

```

We first describe the specification of the problem in terms of an attribute assignment system [20]. Here the value of attributes start, next, true, and false is an integer, and the value of attribute code is a

string.

Fig. 4.1. An attribute assignment description

```

0) Z -> E
    Z.code = (:=,1,Z) + E.code + (:=,0,Z)
    E.start = Z
    E.true = E.next+1
    E.false = E.next

1) E -> T
    E.code = T.code
    E.next = T.next
    E.start = T.start
    E.true = T.true
    E.false = T.false

2) E1 -> TOR E2
    E1.code = TOR.code+E2.code
    E1.next = E2.next
    E1.start = TOR.start
    E2.start = TOR.next
    E1.true = TOR.true = E2.true
    E1.false = E2.false
    TOR.false = TOR.next

3) TOR -> T or
    TOR.code = T.code
    TOR.next = T.next
    TOR.start = T.start
    TOR.true = T.true
    TOR.false = T.false = T.next

4) T -> F
    T.code = F.code
    T.next = F.next
    T.start = F.start
    T.true = F.true
    T.false = F.false

5) T1 -> FAND T2
    T1.code = FAND.code+T2.code
    T1.next = T2.next
    T1.start = FAND.start
    T2.start = FAND.next
    T1.true = T2.true
    T1.false = FAND.false = T2.false
    FAND.true = FAND.next

6) FAND -> F and
    FAND.code = F.code
    FAND.next = F.next
    FAND.start = F.start
    FAND.true = F.true = F.next

```

```

FAND.false = F.false
F.code = (Br, lex(), F.true, F.false)
F.next = F.start+1
F.code = E.code
F.next = E.next
F.start = E.start
F.true = E.true
F.false = E.false
F1 -> not F2
F1.code = SWAP(F2.code)
F1.true = F2.false
F1.false = F2.true
F1.next = F2.next
F1.start = F2.start
-----
By treating attributes start, true, and false as inherited
attributes, and attributes code and next as synthesized
attributes, we obtain a natural attribute
grammar description in Fig. 4.2.
-----
Fig. 4.2. An attribute grammar description
0) Z -> E
E.start := 2
E.true := E.next+1
E.false := E.next
Z.code := (:,1,Z) + E.code + (:,0,Z)
1) E -> T
T.start := E.start
E.next := T.next
T.true := E.true
T.false := E.false
E.code := T.code
2) E1 -> TOR E2
TOR.start := E1.start
E2.start := TOR.next
E1.next := E2.next
TOR.true := E2.true
E2.false := E1.false
TOR.false := TOR.next
E1.code := TOR.code + E2.code
-----
By assuming the canonical bottom-up parsing, we can eliminate
occurrences of start and next, and we can also
eliminate occurrences of code by simple postfix

```

```

T.start = TOR.start
TOR.next := T.next
T.true := TOR.true
TOR.false := T.false := T.next
TOR.code := T.code
4) T -> F
F.start := T.start
T.next := F.next
F.true := T.true
F.false := T.false
T.code := F.code
5) T1 -> FAND T2
FAND.start := T1.start
T2.start := FAND.next
T1.next := T2.next
T2.true := T1.true
FAND.false := T2.false := T1.false
FAND.true := FAND.next
T1.code := FAND.code+T2.code
6) FAND -> F and
F.start := FAND.start
FAND.next := F.next
FAND.true := F.true := F.next
F.false := FAND.false
FAND.code := F.code
7) F -> i
F.next := F.start+1
F.code := (B, lex(), F.true, F.false)
8) F -> ( E )
E.start := F.start
F.next := E.next
E.true := F.true
E.false := F.false
F.code := E.code
9) F1 -> not F2
F2.start := F1.start
F1.next := F2.next
F2.true := F1.false
F2.false := F1.true
F1.code := SWAP(F2.code)
-----

```

By assuming the canonical bottom-up parsing, we can eliminate occurrences of start and next, and we can also eliminate occurrences of code by simple postfix

}
 }
 }

3) TOR -> T or

transformation scheme. (We assume that generate(:=,1,Z) is already done) Also because of identity relations, we can eliminate occurrences of TOR.false and FAND.true. Now we obtain the following description in Fig. 4.3.

Fig. 4.3. A description without start, next and code

```

0) Z -> E
-----
1) E -> I
   E.true := E.next+1
   E.false := E.next
   generate(:=,0,Z)
2) E1 -> TOR E2
   T.true := E.true
   T.false := E.false
   TOR.true := E2.true := E1.true
   E2.false := E1.false
3) TOR -> T or
   T.true := TOR.true
   T.false := T.next
4) T -> F
   F.true := T.true
   F.false := T.false
5) T1 -> FAND T2
   T2.true := T1.true
   FAND.false := T2.false := T1.false
6) FAND -> F and
   F.true := F.next
   F.false := FAND.false
7) F -> i
   generate(Br, lex(), F.true, F.false)
8) F -> ( E )
   E.true := F.true
   E.false := F.false
9) F1 -> not F2
   F2.true := F1.false
   F2.false := F1.true
-----

```

We reverse the flow of inherited attributes true and false. Attributes true and false are now synthesized

attributes. Then we introduce patch operations for those attribute values. We obtain the following Fig. 4.4.

Fig. 4.4. An invalid description

```

0) Z -> E
-----
1) E -> T
   patch(E.true, true, E.next+1)
   patch(E.false, false, E.next)
   generate(:=,0,Z)
2) E1 -> TOR E2
   E.true := T.true
   E.false := T.false
   * E1.true := E2.true
   * E1.true := TOR.true
   E1.false := E2.false
3) TOR -> T or
   TOR.true := T.true
   patch(T.false, false, T.next)
4) T -> F
   T.true := F.true
   T.false := F.false
5) T1 -> FAND T2
   * T1.false := T2.false
   * T1.false := FAND.false
   T1.true := T2.true
6) FAND -> F and
   patch(F.true, true, F.next)
   FAND.false := F.false
7) F -> i
   F.true := F.false := nextlocation()
   generate(Br, lex(), 0, 0)
8) F -> ( E )
   F.true := E.true
   F.false := E.false
9) F1 -> not F2
   F1.true := F2.false
   F1.false := F2.true
-----

```

Because of one-to-many mapping of reverse flow, The description in Fig. 4.4 is invalid. By treating attributes true and false as a set of integers instead of an

integer, we obtain the final action routine in Fig. 4.5.

```

Fig. 4.5. Final action routine
-----
0) Z -> E
    patch(E,false, false, E.next)
    patch(E,true, true, E.next+1)
    generate(=,0,,Z)
1) E -> T
    E.true := T.true
    E.false := T.false
2) E1 -> TOR E2
    E1.true := TOR.true + E2.true
    E1.false := E2.false
3) TOR -> T or
    TOR.true := T.true
    patch(T,false, false, T.next)
4) T -> F
    T.true := F.true
    T.false := F.false
5) T1 -> FAND T2
    T1.false := FAND.false + T2.false
    T1.true := T2.true
6) FAND -> F and
    patch(F,true, true, F.next)
    FAND.false := F.false
7) F -> i
    F.true := F.false := {nextlocation()}
    generate(Br, lex(), 0, 0)
8) F -> ( E )
    F.true := E.true
    F.false := E.false
9) F1 -> not F2
    F1.true := F2.false
    F1.false := F2.true
-----

```

5. Bypassed Transformation
 Bypassed transformation deals with general non-circular attribute grammar cases [19]. This method first constructs bypassed parse tree based on bypassed LR parsing methods [18]. Then the method constructs a

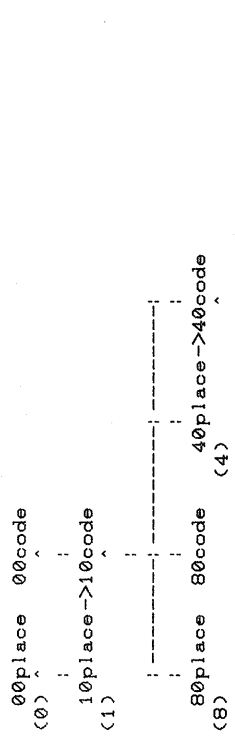
bypassed dependency graph.

In case of the example in Fig. 3.1, we obtain the LR(1) parsing table in Fig. 5.1 and the dependency graph for $i+i$ in Fig. 5.2. The rest of the evaluation process is same as that of Knuth [10]. Hence we can eliminate redundant copy operations involving unit productions efficiently.

Fig. 5.1. Bypassed LR(1) parsing table

	:	:	i)	(/	*	-	+	\$	E	T	F
0	:	:	4	:	3	:	:	:	:	:	1	2	2
1	:	:	:	:	:	:	6	5	a	:	:	:	:
2	:	:	4	:	3	:	8	7	6	5	a	:	:
3	:	:	4	:	3	:	-8	-8	-8	-8	-8	:	10
4	:	:	4	:	3	:	:	:	:	:	:	11	11
5	:	:	4	:	3	:	:	:	:	:	:	12	12
6	:	:	4	:	3	:	:	:	:	:	:	13	13
7	:	:	4	:	3	:	:	:	:	:	:	14	14
8	:	:	4	:	3	:	:	:	:	:	:	:	:
9	:	:	15	:	15	:	8	7	6	5	:	:	:
10	:	:	15	:	15	:	8	7	6	5	:	:	:
11	:	:	-1	:	-1	:	8	7	-1	-1	:	:	:
12	:	:	-2	:	-2	:	8	7	-2	-2	:	:	:
13	:	:	-4	:	-4	:	-4	-4	-4	-4	:	:	:
14	:	:	-5	:	-5	:	-5	-5	-5	-5	:	:	:
15	:	:	-7	:	-7	:	-7	-7	-7	-7	:	:	:

Fig. 5.2. A dependency graph




```

-----
:
:
:
80place 80code 80place 80code
(8)
-----

```

6. Initial Value Introduction

This method is not yet a systematic method. We consider a problem of checking whether or not a given string of parentheses is balanced. Our starting attribute grammar description is given in Fig. 6.1.

Fig. 6.1 Checking parentheses

```

-----
1) N -> L
2) L1 -> L2 B
   N.test := (L.test) and (L.count = 0)
   L1.count := L2.count + B.count
   L1.test := (L2.test) and (L1.count >= 0)
3) L -> B
   L.count := B.count
   L.test := (L.count >= 0)
4) B -> (
   B.count := +1
5) B -> )
   B.count := -1
-----

```

Because of dependency relations, we need two global variables such as var0 and var1 in Fig. 6.2, if we transform it into action routines.

Fig. 6.2. An action routine with two variables

```

-----
1) N -> L
2) L -> L B
   if var0 = 0 then yes else error
   var0 := var0 + var1
   if var0 < 0 then error
-----

```

```

3) L -> B
   var0 := var1; if var0 < 0 then error
4) B -> (
   var1 := +1;
5) B -> )
   var1 := -1;
-----

```

We may also use three global variables as in Fig. 6.3.

Fig. 6.3 An action routine with three variables

```

-----
1) N -> L
   if var0 = 0 then yes else error
2) L -> L B
   var0 := var0 + var1 + var2
   if var0 < 0 then error
3) L -> B
   var0 := var1 + var2
   if var0 < 0 then error
4) B -> (
   var1 := +1; var2 := 0
5) B -> )
   var2 := -1; var1 := 0
-----

```

By introducing initialization mechanism, we can reduce the number of variables to one as in Fig. 6.4.

Fig. 6.4. An action routine with one variable

```

-----
initially, count:=0
1. N -> L
2. L -> L B
   if count = 0 then yes else error
3. L -> B
   if count < 0 then error
4. B -> (
   count := count + 1
5. B -> )
   count := count - 1
-----

```

7. Conclusion

We have shown some methods for transforming attribute grammars into efficient action routines. Integration of various efficient evaluation techniques of attribute grammars should be explored.

REFERENCES

- [1] Aho, A.V. and Ullman, J.D. Principles of Compiler Design, Addison-Wesley (1977).
- [2] Bochmann, G.V. Semantic evaluation from left to right, Comm. ACM, 19, 2 (Feb. 1976), 55-62.
- [3] Demers, A., Reps, T. and Teitelbaum, T. Incremental evaluation for attribute grammars with applications to syntax-directed editors, Proc. of 8th ACM POPL (1981), 105-116.
- [4] Farrow, R. Generating a production compiler from an attribute grammar, IEEE Software 1,4 (Oct. 1984), 77-83.
- [5] Jazayeri, M., Ogdan, W.F. and Rounds, W.C. The intrinsically exponential complexity of the circularity problem for attribute grammars, Comm. ACM 18, 12 (Dec. 1975), 697-706.
- [6] Joliat, M.L. A simple technique for partial elimination of unit productions from LR(k) parsers, IEEE Trans. Comp. C-25, 7 (July 1976), 763-764.
- [7] Katayama, T. Translation of attribute grammars into procedures, ACM Trans. Prog. Lang. and Systems 6,3 (1984), 345-369.
- [8] Kennedy, K. and Warren, S.K. Automatic generation of efficient evaluators for attribute grammars, Proc. of 3rd ACM POPL (1976), 32-49.
- [9] Kennedy, K. and Ramanathan A deterministic attribute grammar evaluator based on dynamic sequencing, ACM Trans. Prog. Lang. and Systems 1, 1 (1979), 142-160.
- [10] Knuth, D.E. Semantics of context-free languages, Math. Systems Theory 2,2 (1968) 127-145; Correction 5,1 (1971), 95-96.
- [11] Lewis, P.M., Rosenkrantz, D.J., and Stearns, R.E. Compiler Design Theory, Addison-Wesley (1976).
- [12] Lorho, B. Semantic attribute processing in the system DELTA, Lecture Notes in Computer Science 47 (1977), 21-40.
- [13] Marcotty, M., Leggard, H.F., and Bochmann, G.V. A sampler of formal definitions, Computing Surveys 8,2 (1976), 189-276.
- [14] Raiha, K.J. A space management technique for multi-pass attribute evaluators, Technical report A-1981-4, Dept. of Comp. Science, University of Helsinki (1981).
- [15] Reps, T. Generating Language-Based Environments, Technical Report 82-514, Dept. of Comp. Science, Cornell University (1982).
- [16] Saarinen, M. On constructing efficient evaluators for attribute grammars, Lecture Notes in Computer Science 62 (1978), 382-386.
- [17] Sasaki, H. Global storage allocation in attribute evaluation, Ph.D. Thesis, Dept. of Comp. Science, Tokyo Inst. of Tech. (1985)
- [18] Tokuda, T. A transformation method for reducing the number of states of bypassed Knuth LR(k) parsers, Trans. IECEJ E67,5 (1984), 259-266.
- [19] Tokuda, T. Two methods for eliminating redundant copy operations from the evaluation of attribute grammars [Submitted to J. Information Processing (1986)].
- [20] Tokuda, T. An attribute assignment approach to the interpretation and evaluation of non-procedural computing systems [Submitted to J. Information Processing (1986)].