A Compiler Generator for Attribute Evaluation During LR Parsing

Jorma Tarhio

University of Helsinki
Department of Computer Science
Teollisuuskatu 23, SF-00510 Helsinki, Finland

Abstract

A compiler generator called Metauncle is introduced. Metauncle produces one-pass compilers in which all attributes are evaluated in conjunction with LR parsing. The description of a language is given to Metauncle as an L-attributed grammar, and the system transforms it before generation of an evaluator to another attribute grammar satisfying the requirements for evaluation. The transformed grammar belongs to the class of so-called uncle-attributed grammars. Besides general information about the system, the definition of uncle-attributed grammars, the idea of the grammar transformation and the default rules of the specification language are presented.

1. Introduction

The evaluation of inherited attributes is difficult in conjunction with LR parsing because the upper part of the derivation tree is incomplete during parsing and thus the tree structure cannot be used to convey values of inherited attributes. There are several methods [Wat77, JoM80, Tar82, Poh83, Mel84, SIN85] to cope with the situation. In the following, we will consider one of these, the uncle method introduced in [Tar82]. The uncle method requires that all the semantic rules for inherited attributes are copy

---

The work was supported by the Academy of Finland.

rules, which simplifies evaluation considerably, because then the attribute evaluator will be able to refer to the values of inherited attributes as copies of values of synthesized attributes associated with the roots of the completed subtrees.

We will introduce a new compiler generator Metauncle [Tar88b], which generates analyzers that evaluate all attributes during LR parsing according to the uncle method. The description of a language is given to Metauncle as an L-attributed grammar, and the system transforms it to another attribute grammar suitable for generation of an evaluator. The transformed grammar belongs to the class of so-called uncle-attributed grammars [Tar82, Tar88a].

The Metauncle system has been implemented using the compiler generator HLP84 [KNP88, KEL88]. A prototype of Metauncle was operational on a Burroughs B7800 in December 1986; the present version is now running on a VAX 8800/8300 under VMS.

The rest of the paper is organized as follows. In Section 2, general information of Metauncle is given. Default rules of the specification language are explained in Section 3. Uncle-attributed grammars are defined in Section 4, and the details of the grammar transformation are explained in Section 5. Finally, experiences of the system are described in Section 6.

2. General description

Metauncle consists of two processors: one performs the grammar transformation and the other generates an attribute evaluator from the transformed grammar. These processors have been implemented using the compiler generator HLP84 [KNP88, KEL88] by describing both of them as an attribute grammar. The implementation language of Metauncle and the generated compilers is Pascal.

The specification language for Metauncle has a traditional form. Attributes, grammar symbols and names of external types are declared before the attributed productions. Productions are given in the BNF-form. Present version of the specification language is slightly modified from the form presented in [Tar88b].

The example grammar in Fig. 1 is written in the specification language and it describes a simple block-structured language. A block consists of two lists: a declaration list and a use list. The function INIT initializes the symbol table, CONC stores a new identifier to the symbol table and CHECK examines whether a used identifier has been properly declared.

```
TYPES
     SYMBOLID;
     SYMBOLTABLE

ATTRIBUTES
     ID: SYNTHESIZED SYMBOLID;
     EI: INHERITED SYMBOLTABLE;
     ES: SYNTHESIZED SYMBOLTABLE

GRAMMAR SYMBOLS
     P; B(EI); DL(EI;ES); D(EI;ES); SL(EI); S(EI); IDENT(ID)

PRODUCTIONS
     P = B                   RULES B.EI := INIT() END;
     B = 'BEGIN' DL ';' SL 'END'
                             RULES DL.EI := B.EI;
                                   SL.EI := DL.ES END;
     DL = D                  RULES D.EI := DL.EI;
                                   DL.ES := D.ES END;
     DL = DL ',' D           RULES DL_2.EI := DL.EI;
                                   D.EI := DL_2.ES;
                                 DL.ES := D.ES END;
     D = 'DECL' IDENT        RULES D.ES := ADD(D.EI,IDENT.ID) END;
     SL = S                  RULES S.EI := SL.EI END;
     SL = SL ',' S           RULES SL_2.EI := SL.EI;
                                   S.EI := SL.EI END;
     S =  B                  RULES B.EI := S.EI END;
     S = 'USE' IDENT         RULES CHECK(S.EI,IDENT.ID) END
```

Fig. 1. Example grammar.

When generating a compiler for a language, the user gives to Metauncle an attribute grammar, external declarations containing constants, variables and semantic functions (in Pascal) and a lexical description. From the attribute grammar Metauncle produces an attribute evaluator. While compiling of a target compiler, the evaluator and the external declarations

are merged with a skeleton compiler, which is a modification of that one used for target compilers in the HLP84 system. A parser for the target compiler is produced with the same parser generator used in the HLP84 system. The default type of a parser is a table-driven LALR(1) parser, but the generator is also capable to make some other kinds of LR parsers, see [KEL88]. The lexical description is given in a separate file in the form used in HLP84 [KNP88].

In a compiler generated by Metauncle, attribute evaluation is directed by an LR parser. All evaluation actions are carried out in conjunction with reductions. To save space, storage areas for attributes are deallocated after the last reference to them.

3. Default rules

In the specification language, all copy rules of a grammar need not to be presented. So the list of productions with semantic rules in Fig. 2 is equivalent with the complete form given in Fig. 1. This is a new feature added to the latest version of Metauncle.

```
P = B                 RULES B.EI := INIT() END;
B = 'BEGIN' DL ';' SL 'END';
DL = D                ;
DL = DL ',' D         ;
D = 'DECL' IDENT      RULES D.ES := ADD(D.EI,IDENT.ID) END;
SL = S                ;
SL = SL ',' S         ;
S =  B                ;
S = 'USE' IDENT       RULES CHECK(S.EI,IDENT.ID) END
```

Fig. 2. Productions without default rules.

The principles for default rules are based on the L-attributedness of the input grammar and on the following name convention. An inherited and a synthesized attribute symbol are assumed to represent the same global structure, if their names are the same except the last letter, which is *i* for an inherited symbol and *s* for a synthesized symbol. Thus inherited *envi* and synthesized *envs* are, for example, assumed to be used for the same purpose.

The right-hand side of a default copy rule for an output occurrence is the previous input occurrence of the same kind in the standard evaluation order for L-attributed grammars.

More formally, let us consider a production $X_0 \rightarrow X_1 X_2 ... X_n$. If X_k, $k > 0$, has an inherited attribute X_k.<a>i with no explicit semantic rule, the right-hand side of the default rule for X_k.<a>i is the first existing attribute occurrence of X_{k-1}.<a>s, ..., X_1.<a>s and X_0.<a>i. If X_0 has a synthesized attribute X_0.<a>s with no explicit semantic rule, the right-hand side of the default rule for X_0.<a>s is the first existing attribute occurrence of X_n.<a>s, ..., X_1.<a>s and X_0.<a>i.

In practical L-attributed grammars, the default rules seem to cover 75 % of all semantic rules.

4. Uncle-attributed grammars

The Metauncle system transforms an L-attributed input grammar to an uncle-attributed grammar from which an attribute evaluator is generated. We will define uncle-attributed grammars following mainly the notations of [Tar88a]. An introduction to the subject can be found in Section 3.1 in [Tar82].

In uncle-attributed grammars, the semantic rules for inherited attributes are restricted to copy rules, and then the values of inherited attributes are available as copies of values of synthesized attributes associated with the roots of the completed subtrees. Let us consider the situation described in Fig. 3. Just after the LR parsing of the subtree B has been finished, the nonterminal B will be in the parsing stack until the reduction by $A \rightarrow BC$ is performed, and the value of the synthesized attribute $B.b$ can be used for the values of inherited attributes $C.c$ and $E.e$ provided that the relevant semantic rules for $C.c$ and $E.e$ are copy rules. The synthesized attribute $B.b$ can be used even if B were not just on the top of the stack like in the situation described.

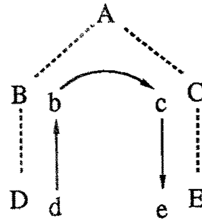

Fig. 3. Use of uncles.

We define a *copy relation*, denoted $\underline{C}$, on the set of the attribute symbols of the grammar as follows: $b \underline{C} c$, if there is a semantic rule $X_i.c := X_j.b$ in a production $X_0 \to X_1 \ldots X_n$, where c is an inherited attribute symbol. When $b \underline{C}^+ c$, we say that c is *copy-dependent* on b. Let b and d be inherited and s be synthesized. If $s \underline{C} d \underline{C}^* b$ and $X_i.d := X_j.s$ is a copy rule in $X_0 \to X_1 \ldots X_n$, we say that the grammar symbol X_j is an *uncle symbol* of b and $X_j.s$ is an *uncle attribute* of b. In Fig. 3, the nonterminal B is an uncle symbol of the inherited attribute symbol e, and $B.b$ is an uncle attribute of e. In Fig. 3 an explanation can be seen for the term "uncle", for the uncle symbol of e is a left brother of an ancestor of the node with which e is associated.

The evaluation strategy can be described with these concepts. All the evaluation actions are applied to synthesized attributes and carried out in conjunction with the reductions. Let us study a parsing situation $(\beta\alpha, u)$, where the reduction by $A \to \alpha$ is the next parsing action. In conjunction with this reduction we will evaluate all the synthesized attributes of A using the values of the synthesized attributes associated with α (we assume those attributes have been evaluated before the current situation) and the values of the inherited attributes of A. The value of an inherited attribute $A.b$ is got from a synthesized attribute associated with the rightmost uncle symbol of b in β (i.e. the topmost uncle symbol of b in the parsing stack). Note that the inherited attributes themselves are never evaluated; their values are available only as copies of the values of synthesized attribute instances.

Definition. An L-attributed grammar G is *uncle-attributed*, if the conditions U1, U2 and U3 are satisfied.

U1. All semantic rules for inherited attributes are copy rules.

U2. If $X_0 \rightarrow X_1 \ldots X_n$ is a production with a copy rule $X_j.b := X_i.c$, where $0 \leq i < j \leq n$, then for every inherited attribute symbol d, such that $b \underline{C}^* d$, none of the grammar symbols $X_{i+1}, \ldots, X_{j-1}$ is an uncle symbol of d.

U3. An inherited attribute symbol is copy-dependent on only one of the synthesized attribute symbols associated with its uncle symbol.

The first condition fixes the form of semantic rules for inherited attributes, and the last condition ensures that we can use the values of uncle attributes unambiguously. The second condition deals with two kinds of copy rules for inherited attributes, where the right-hand side $X_i.c$ is either a synthesized attribute occurrence when $i > 0$ or an inherited attribute occurrence when $i = 0$. This condition guarantees that the topmost uncle symbol of d in the parsing stack will be used. In Fig. 4 there is a counter-example, where the grammar symbol B is an uncle symbol of d. The attribute $C.d$ gets its value from an uncle attribute attached to the occurrence of B which is not the topmost (rightmost) one.

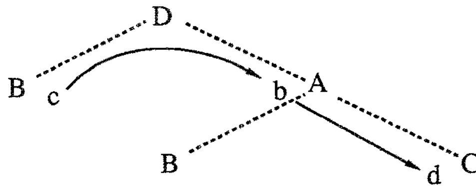

Fig. 4. Conflict against U2.

Theorem. All the attributes of an uncle-attributed LR(k) grammar G can be evaluated during LR parsing.

Proof. See [Tar88a].

5. Uncle transformation

An L-attributed grammar must be transformed to the uncle-attributed form before the generation of an evaluator. In this transformation called the uncle transformation, the conflicts against conditions U1, U2 and U3 are removed one at a time. This requires insertions of new marker nonterminals generating the empty string into the grammar.

Though the uncle transformation can make every L-attributed LR grammar uncle-attributed, it is not guaranteed that the grammar is any more LR after the transformation because of the marker nonterminals. For example, we cannot enter such a nonterminal in front of the right-hand side of a left-recursive production. However, the parsing conflicts do not seem to be very usual in the case of practical grammars. We will return to this subject later on.

The transformation consists of five phases. The first phase makes an L-attributed input grammar satisfy U1, i.e. all semantic rules for inherited attributes will be copy rules.. The technique used is well-known (see e.g. [ASU86]): a nontrivial rule for an inherited attribute is moved to a rule for a synthesized attribute associated with a marker nonterminal inserted in front of the nonterminal occurrence involved with the original inherited attribute. Let us consider an example. Let a semantic rule $C.c$:= F($A.a, B.b$) be associated with a production $A \rightarrow BC$. The transformed structure will be:

$$
\begin{aligned}
&A \rightarrow B\,X\,C\\
&\qquad X.a := A.a\\
&\qquad X.b' := B.b\\
&\qquad C.c := X.c'\\
&X \rightarrow \varepsilon\\
&\qquad X.c' := \mathrm{F}(X.a, B.b')
\end{aligned}
$$

The example structure before and after transformation is described in Fig. 5. In the following we call this kind of a local transformation a *rule transfer*.

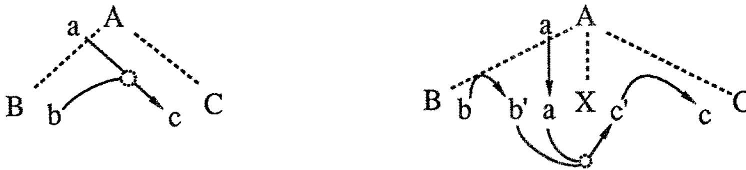

Fig. 5. A rule transfer.

The optional second phase eliminates a part of the conflicts against condition U2. This phase is explained in detail later.

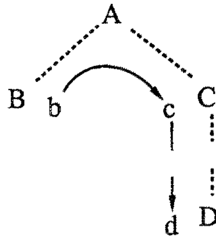

Fig. 6. A conflict against U3.

The third phase eliminates the conflicts against condition U3. Let us consider the situation in Fig. 6. If $c\ \underline{C}^*\ d$ for some inherited d, c is copy-dependent on b and B is an uncle symbol of d. There is a conflict against U3, if d is also copy-dependent on another synthesized attribute symbol associated with B. The solution is to perform a rule transfer for the copy rule $C.c := B.b$. The transformed production is shown in Fig. 7.

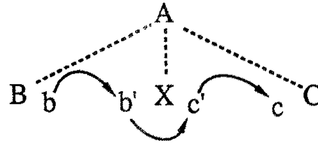

Fig. 7. A solution to the conflict against U3

The elimination of conflicts against U2 is trickier because it must be done in two phases. Let us consider an example where the semantic rule $C.c := A.a$ is associated with the production $A \rightarrow BC$, and B is an uncle symbol of c. In the fourth phase a rule transfer is performed for $C.c := A.a$. The modified structure is shown in Fig. 8.

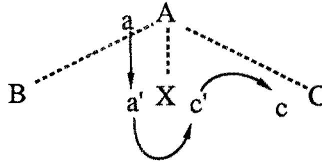

Fig.8. Rule transfer in the fourth phase

If B is not an uncle symbol of a, the conflict has been eliminated. However, the conflict may still be present like in Fig. 9 (a), where B is an uncle of a and a', and so the conflict against U2 is still present. This conflict will be recognized and eliminated in the fifth phase of the transformation where the rule $X.a' := A.a$ is transferred in front of B this time (Fig. 9 (b)).

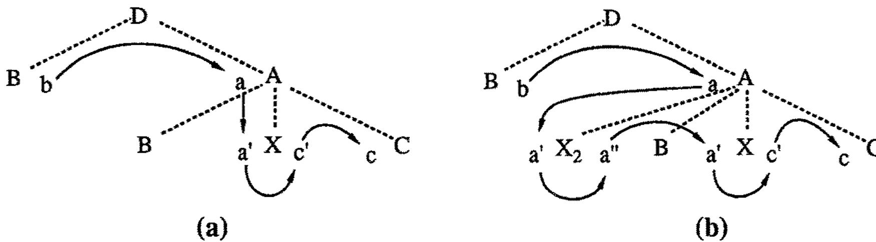

Fig. 9. Solution for an inherited conflict.

Now we return to the second phase. All the conflicts against condition U2 can be removed in the fourth and fifth phase of the uncle transformation, but this approach may produce parsing conflicts. For example, if we replace the nonterminal *A* by *B* in the example described in Figures 8 and 9, the resulting structure would cause a parsing conflict, because *B* is then left-recursive. In some cases a parsing conflict can be prevented by performing another kind of a transformation depicted in Fig. 10, where the uncle symbol *B* has been hidden.

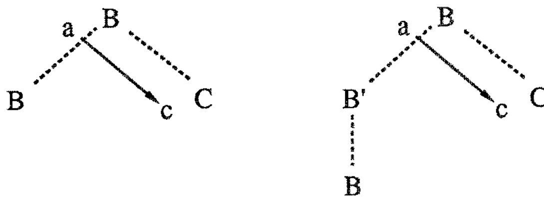

Fig. 10. Hiding of an uncle symbol.

This kind of hiding of uncle symbols does not, however, solve all the conflicts against U2. It is possible that the copy chain starts from the same production where the conflict against U2 is detected and then the hiding of an uncle symbol does not help.

It is easy to show that the uncle transformation removes all conflicts against U1 - U3 [Tar88a].

6. Experiences

To test the Metauncle system several attribute grammars have been processed. One of them describes a large subset of the static semantics of Pascal [Tuu87]. To give a view of the effect of the uncle transformation on the size of an attribute grammar, statistics of the L-attributed grammar for Pascal given as input and of the uncle-attributed form produced by the uncle transformation are given in Fig. 11.

	L-attributed	*Uncle-attributed*
Grammar symbols	148	183
Attribute symbols	19	47
Productions	297	332
Semantic rules	700	807

Fig. 11. Figures about two Pascal grammars

Evaluation conflicts appear always as parsing conflicts caused by marker nonterminals inserted by the uncle transformation. Because the changes made to the grammar are always local and they correspond to the conditions for uncle-attributed grammars, it is quite easy to infer the reasons for the conflicts by comparing the transformed grammar with the original one, because the transformed grammar is an ordinary attribute grammar written in the very same specification language as the original one. However, the conflicts are not very common in practical L-attributed grammars. In preparing the Pascal grammar mentioned above about ten evaluation conflicts were encountered. The solving of the conflicts took for a graduate student about 2 % of the total work time of the project.

The present version of Metauncle produces an analyzer of Pascal in 80 seconds of processor time on a VAX 8800 starting from the L-attributed description. The total time would be smaller, if the uncle transformation and the generation of an evaluator were merged. However, there are advantages in having two separate processors. Namely, the system is conceptually simpler to control and it is easier to understand reasons for evaluation conflicts as explained above. A separate transformation also offers flexibility and possibilities to make experiments. For example, a grammar may be only once augmented with the default rules (using an option of the first processor), and the augmented form is then developed further to optimize the coding time and to prevent possible errors in repeated augmentations.

To evaluate the overall efficiency of a compiler generated by Metauncle, the Pascal processor produced by Metauncle was compared on a VAX 8800 with a Pascal processor produced by HLP84. The processor generated by Metauncle was slightly faster than the other processor. As a

comparison, the same tests were also run by the standard Pascal compiler of VAX (the times for this compiler include the code generation, too). The hand-written processor of VAX was typically about 2-3 times faster than the processor generated by Metauncle.

Another grammar written for Metauncle is a description of the specification language for input grammars [Ran88]. The processor generated from that grammar is used as a tool in developing new descriptions. One reason for making such a self description is the unsatisfactory error recovery of a processor generated by HLP84. The description could be easily extended so that the uncle transformation and the generation of a evaluator could be performed by processors generated by Metauncle itself.

One characteristic feature of the uncle method is the search for uncle symbols in the parsing stack. (Conceptually, it is slightly misleading to speak about a stack, because every item of the data structure should be accessible in our approach.) This feature is theoretically time-consuming, because the evaluation algorithm may need $O(n^2)$ time in the worst case for an input of length n [Tar82]. However, the uncle symbols are always close to the top of the parsing stack in the case of practical grammars. The search time was only 1 % of the total compilation time in experiments done with compilers generated by Metauncle.

The experiments support the fact that the uncle method combined with the uncle transformation offers an easy way to generate practical compilers for a large class of L-attributed grammars. This class is competitive with the classes accepted by related compiler generators [Tar88a]. The ease of implementing Metauncle using the HLP84 system was also a nice example of the power of compiler writing tools.

References

[ASU86] A. V. Aho, R. Sethi and J. D. Ullman: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.

[JoM80] N. D. Jones and C. M. Madsen: Attribute-influenced LR Parsing. In: *Aarhus Workshop on Semantics-Directed Compiler Generation* (ed. N. D. Jones), Lecture Notes in Computer Science 94, Springer-Verlag, Berlin-Heidelberg-New York, 1980, 393–407.

[KEL88] K. Koskimies, T. Elomaa, T. Lehtonen and J. Paakki: TOOLS/HLP84 report and user manual. Report A-1988-2. Department of Computer Science, University of Helsinki, 1988.

[KNP88] K. Koskimies, O. Nurmi, J. Paakki and S. Sippu: The design of a language processor generator. *Software Practice & Experience* **18, 2** (1988), 107-135.

[Mel84] B. Melichar: Evaluation of attributes during LR syntax analysis. In: *Vorträge des Problemseminars Attributierte Grammatiken und ihre Anwendungen*, Pruchten. WPU Rostock, 1984.

[Poh83] W. Pohlmann: LR parsing for affix grammars. *Acta Informatica* **20, 4** (1983), 283–300.

[Ran88] O. Rannisto: Revisions for the Metauncle system (in Finnish). Draft. Department of Computer Science, University of Helsinki, 1988.

[SIN85] M. Sassa, H. Ishizuka and I. Nakata: A contribution to LR-attributed grammars. *Journal of Information Processing* **8, 3** (1985), 196–206.

[Tar82] J. Tarhio: Attribute evaluation during LR parsing. Report A-1982-4. Department of Computer Science, University of Helsinki, 1982.

[Tar88a] J. Tarhio: Attribute grammars for one-pass compilation. Report A-1988-11. Department of Computer Science, University of Helsinki, 1988.

[Tar88b] J. Tarhio: The compiler generator Metauncle. Report C-1988-23, Department of Computer Science, University of Helsinki, 1988.

[Tuu87] H. Tuuri: An attribute grammar checking the semantics of Pascal for the Metauncle metacompiler (in Finnish). Report C-1987-59. Department of Computer Science, University of Helsinki, 1987.

[Wat77] D. A. Watt: The parsing problem for affix grammars. *Acta Informatica* **8, 1** (1977), 1–20.