ATTRIBUTE-INFLUENCED LR PARSING

Neil D. Jones
Michael Madsen

Computer Science Department
Aarhus University
DK-8000 Aarhus C, Denmark

Abstract

Methods are described which make it possible, when given an arbitrary attribute grammar (or AG),

1. to analyze the AG to determine which of its attributes may be computed during LR parsing;
2. to augment the parser with instructions and data structures to compute many attributes during parsing;
3. to use attribute values to assist the parsing process (e.g. to use symbol table information to decide whether P(X) is an array element or a function call).

INTRODUCTION

Related work

This work builds on a number of other results concerning attribute evaluation during parsing. An early paper by Lewis, Rosenkrantz and Stearns [LRS74] describes evaluation of synthesized attributes during parsing, and introduces the idea of an L-attributed AG, in which both inherited and synthesized attributes may be evaluated in one left-right pass over the parse tree. Bochmann [Boc76] develops this idea further, including multipass evaluation.

Evaluation of inherited attributes during bottom-up parsing is trickier since the parse tree structure is not definitely known (e.g. left recursion gives problems). A method to evaluate both types of attributes during LR parsing of L-attributed AGs is in [Wat77a]. New nonterminals called "copy symbols" are added to the AG. These derive the empty string; their purpose is to drive action routines which

maintain a stack of attribute values. Unfortunately it is not easy to see where or whether copy symbols can be inserted without destroying the LR property. This problem is addressed by Purdom and Brown [PuB79]; They present an efficient algorithm to find "safe" positions in productions for such insertions.

The use of attributes to influence parsing decisions seems to originate with Watt (e.g. [Wat77a]). The technique is further developed and a number of realistic applications are given in [Wat80]. He also describes implementation in top-down, bottom-up and multipass parsers. Milton and Fischer describe the use of this technique in a compiler-writing system which uses LL parsing [MiF79].

Rowland investigates attribute evaluation in bottom-up parsing via left corner parsing [Row77]. Räihä and Ukkonen [RäU80] introduce conditions on attribute grammars parsable by "recursive descent" and "recursive ascent" which allow evaluation during parsing. These generalize the classes of both [MiF79] and [Row77], and allow some use of inherited attributes with left recursion.

Overview

The method to be described systematizes and extends those referenced above. It is based on a more powerful method described in [Mad80a].

Our approach evaluates attributes during LR parsing, based on a preliminary analysis of the structure of the LR parse tables and the AG. If the AG is LR-attributed then every attribute will be evaluated as soon as possible during parsing; otherwise as many attributes as possible are evaluated during parsing,and the remainder are evaluated afterwards. Even for non-LR-attributed AGs (and most realistic programming languages contain a few right dependencies) considerable storage savings may result from this approach, providing the attributes are saved in an expression dag (described below).

Further, the parse-time known attribute values may be used to influence the parsing itself, allowing the use of grammars which are syntactically ambiguous even though semantically unambiguous. Such grammars are often more natural and/or compact than their LR equivalents (and an LR equivalent may not even exist). A common problem solvable by attribute-influenced parsing concerns identifiers: a choice between "procid → identifier", "arrayid → identifier", "simplevar → identifier" is most naturally based on symbol table information. Other natural examples may be found in [Wat80].

Our approach begins with the output of a parser generator (SLR, LALR or LR), so we recall some terminology from [AhU77]. A parse configuration is a pair

$$(S_0\ X_0\ S_1\ \cdots\ X_{m-1}\ S_m\ ,\ a_j \cdots a_n\$)$$

where each X_i is a grammar symbol and each S_i is a state. A state is a set of items of the form $A \rightarrow \alpha.\beta$ where $A \rightarrow \alpha\beta$ is a production (for full LR parsing an item has the form $[A \rightarrow \alpha.\beta\ u]$. We won't make use of the lookahead u and so drop it for nota-

tional simplicity). The parse tables have the functionalities:

ACTION : States x Lookaheads → Actions
GOTO : States x Nonterminals → States

where Actions may be of four types: shift S, reduce $X \to \alpha$, accept and error.

Following is an overview of our method:

1. First the base grammar of the AG is processed by an LR parser generator.
2. The AG is analyzed (in conjunction with the parser's output) to classify each attribute as known or unknown.
3. The values of all known attributes will be maintained on the stack during parsing. Known synthesized attributes of $X_0, \ldots, X_{m-1}$ will be kept with those symbols; in addition, the values of the known attributes in

$$IN(S_i) = \{ a \mid a \text{ is an inherited attribute of a nonterminal B such that } S_i \text{ contains an item } A \to \alpha.B\beta \}$$

will be kept on the stack with state S_i.

4. Values of unknown attributes of the stack symbols will be kept elsewhere (see the next section).
5. Known attributes are evaluated when performing shift and reduce actions (see Figure 2 for an example of a parser augmented by evaluation actions).
6. Attributes may be used to influence the parsing process by replacing error entries in ACTION by disambiguating predicates (the term is from [MiF79]). These may conveniently take the form

```
CASE
    pred_1 : action_1
     ...
    pred_n : action_n
ESAC
```

In this expression $pred_1, \ldots, pred_n$ are logical expressions depending only on attributes present on the stack, and $action_1, \ldots, action_n$ are in Actions. The first true predicate selects the corresponding action.

Remark Each item $A \to \alpha.B\beta$ in a state S_i represents a prediction of the form of the remaining input. Since this is not yet known, we maintain information of the attributes of every such B, even those which may turn out to be unnecessary on the basis of future input. This redundancy seems to be very small for practical AGs, partly because we store only the values of distinct attributes of S_i.

Example of Notation

We use the concise and readable notation of [Wat77b]. The example in Figure 1 (taken from [Watt77a]) should be self-explanatory; it models variable declaration and usage, with the usual constraints that every usage refers to a declared variable, and that no variable is declared twice.

An attribute occurrence a in an AG rule $X \rightarrow X_1 \ldots X_n$ is called

defining if it is an inherited attribute of X or a synthesized attribute of $X_1, \ldots, X_n$

applied if it is a synthesized attribute of X or an inherited attribute of $X_1, \ldots, X_n$.

Following Bochmann [Boc76] we assume that each applied attribute occurrence is a function of the defining attributes. We further assume the AG is noncircular.

Grammar Symbol	Attributes	Interpretations
pg = program	pg ↑OK	is true iff the program contains no errors of variable usage or declaration
dc = declaration	dc ↑OK	is true iff no variable is declared twice in this declaration
	dc ↑SET	is the set of variables declared in this declaration
st = statement	st ↓SET	is the set of variables declared in this program
	st ↑OK	is true iff this statement contains no undeclared variables
v = variable	v ↑VAR	is the name of this variable (given by lexical analysis)

Attribute Productions

<pg ↑OK1 and OK2>	→ <dc ↑SET ↑OK1> <st ↓SET ↑OK2>
<dc ↑{VAR} ↑true>	→ declare <v ↑VAR>
<dc ↑SET ∪ {VAR} ↑OK and VAR ∉ SET>	→ <dc ↑SET ↑OK> declare <v ↑ VAR>
<st ↓SET ↑true>	→ ϵ (empty string)
<st ↓SET ↑OK and VAR ∈ SET>	→ <st ↓SET ↑OK> use <v ↑VAR>

Figure 1. Example of Attribute Grammar Notation

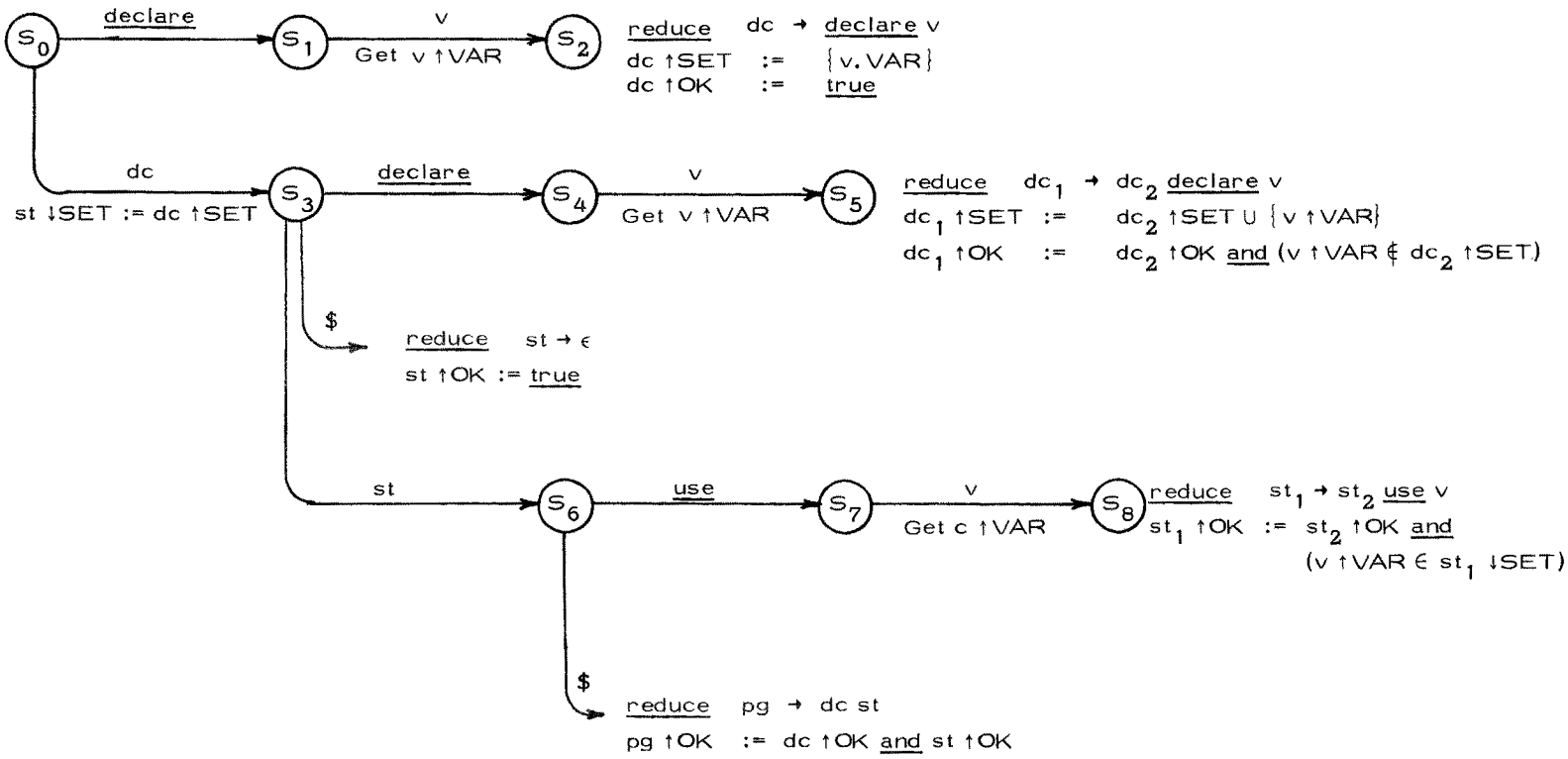

Figure 2. Attribute-Evaluating Parse Table for Figure 1, as a Finite State Machine

STORAGE MANAGEMENT

We now describe two ways to store the values of the unknown attributes.

Attributed parse trees

This is the most straightforward, and involves tagging each parse tree node with a record containing the values of all its attributes. The known attributes of stack symbols $X_0,\ldots,X_{m-1}$ may be placed in the tree; however, the values of known attributes in states $S_0,\ldots,S_m$ must be kept on the stack since it is not known whether or not they will be part of the tree. After parsing is completed the tree is traversed (e.g. by the methods of [KeW76] or [CoH79]) to compute unknown attribute values.

Expression Dags

An alternate approach, described in [LRS74] and implemented in [Mad80b], can yield significant space savings over the method above, and avoid the complexity or incompleteness of known tree traversal algorithms. The parse tree T is not stored at all; instead an expression dag is built during parsing. This has at most one node for each attribute of each node of T. Let $\underline{a}$ be an attribute of some node of T. Then there will be an expression $\underline{a} = ex(\underline{a}_1,\ldots,\underline{a}_n)$ giving the value of $\underline{a}$ in terms of the attributes of other nodes (n equals 0 for constant or lexically supplied attribute values). Node $\underline{a}$ in the dag will be labelled with "ex", and there will be an ordered sequence of edges from $\underline{a}$ to $\underline{a}_1,\ldots,\underline{a}_n$. This graph will be acyclic for any T, since we have assumed the AG to be noncircular.

Clearly the dag may be easily constructed during parsing. It may be compacted during construction by using the previously allocated node whenever an identity attribute expression occurs. Another savings lies in the fact that dag nodes only need be allocated for unknown attributes. Figure 3 contains an example dag for the AG of Figure 1 (an unrealistic example since all attributes in Figure 1 may be evaluated during parsing!).

Attributes may be easily and efficiently evaluated while parsing by a recursive algorithm (essentially a depth-first search).

The parsing algorithm as described will be based on the dag model, but is easily modified to work with attributed derivation trees.

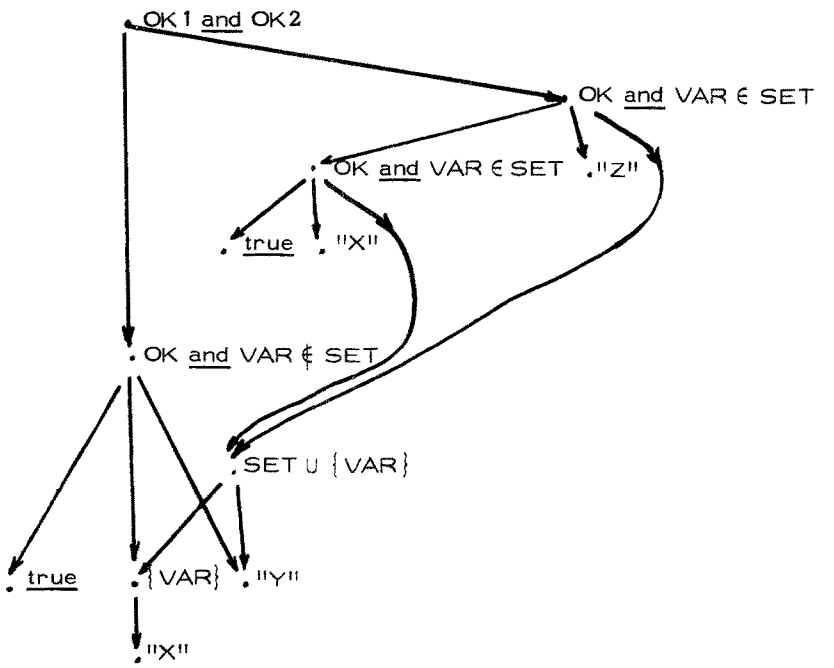

Figure 3. Dag for "declare X declare Y use X use Z"

AN ATTRIBUTE-EVALUATING PARSER

An attributed parse configuration is a pair

$$(S_0 \overline{S}_0 X_0 \overline{X}_0 \cdots S_{m-1} \overline{S}_{m-1} X_{m-1} \overline{X}_{m-1} S_m, a_i \cdots a_n \$)$$

where each S_j is a parse state, each X_j is a grammar symbol, and for $0 \leq j < m$

$\overline{S}_j$ is a record containing

a) the values of the known attributes in $IN(S_j)$
b) pointers to dag nodes for the unknown attributes in $IN(S_j)$

$\overline{X}_j$ is a record containing

a) the values of the known synthesized attributes of X_j
b) pointers to dag nodes for the unknown synthesized attributes of X_j

The behaviour of the parsing algorithm is determined by the choice of the known attribute set, K.

```
PROCEDURE Parse:
BEGIN
Configuration := (S0 , a1...an$);
Dag := empty;
DO FOREVER
    Let Configuration = (S0S̄0 ... X̄m-1Sm , ai...an$);
    action := ACTION[Sm,ai]; {ACTION = parse table}
    IF action = conflict THEN action := CASE {use disambiguating predicate}
                                            pred1: action1
                                              ...
                                            predn: actionn
                                        ESAC
    IF action = accept or error THEN ESCAPE;
    Compute values of known attributes in In(Sm);
    Create new dag nodes for unknown attributes In(Sm);
    Push S̄m = record containing these values and node pointers;
    IF action = shift S
    THEN [X := ai; pop ai from input]
    ELSE [Let action be reduce X → α;
          k := |α|; S := GOTO(Sm-k,X);
          Pop Configuration down to (S0...Sm-kS̄m-k , ai...an$);]
    Let Configuration be (S0...SpS̄p , aj...an$);
    Compute known synthesized attributes of X;
        (from lexical analysis if X is terminal)
    Create new dag nodes for unknown synthesized attributes of X;
    X̄ := record containing these values and node pointers;
    Configuration := (S0...SpS̄pXX̄S , aj...an$)
OD
END
```

Figure 4. Attribute Influenced Parser

Remarks on the parsing algorithm.

1. Correctness and efficiency of the algorithm are affected by the choice of K, the set of known attributes. For correctness K should be small enough so that whenever an attribute value is computed, all values it depends on are available on the stack (this may be trivially accomplished by setting $K = \emptyset$ so that all attributes are evaluated via the dag). For efficiency K should be as large as possible.

2. If an unknown attribute is copied in an AG rule, a new dag node need not be created.

3. Attribute values in $In(S_m)$ will not be used if $action[S_m, a_i] = \underline{reduce}\ X \to \alpha$ with $\alpha \neq \epsilon$. A test could be inserted to bypass this computation.

4. The algorithm could be extended to handle some right dependencies as follows. A state transition goes from a state S to state

$$S' = CLOSURE(\{A \to \alpha X.\beta \mid A \to \alpha.X\beta \in S\})$$

Once the known synthesized attributes of X are computed it may be possible to evaluate some new inherited attributes of symbols in αX. In addition the dag evaluator might be called at this point to evaluate the synthesized attributes of any symbol Y of αX whose inherited attributes of Y have now all been evaluated. It appears that these possibilities can be handled by extending our methods, but at the cost of a considerable increase in complexity.

CHARACTERIZATION OF KNOWN ATTRIBUTES

To do this we analyze the information available to the parsing algorithm when computing a known attribute a. Three cases arise:

1. a is a synthesized attribute of a terminal. The value of a is given by lexical analysis.

2. a is a synthesized attribute of a nonterminal A. Then a is computed whenever a reduction by an AG rule $A \to \alpha$ is performed. It is a characteristic of LR parsers that the symbols of α will be on the stack top (between the states), so all information a depends on is potentially available.

3. a is an inherited attribute of a symbol A and the items of S_m with an A following the dot are $B_i \to \alpha_i.A\beta_i$ $(i = 1, \ldots, n)$. Another LR parser characteristic is that for all i, j either α_i is a suffix of α_j or vice versa. All the attributes of $\alpha_1, \ldots, \alpha_n$ and the inherited attributes of $B_i, \ldots, B_n$ are thus potentially available; however no attribute of $\beta_1, \ldots, \beta_n$ and no synthesized attribute of A is available. Further, if there are be two indices i, j which cause a to receive different values then a cannot be computed dur'ng parsing.

Suppose now that K is such that the algorithm evaluates every known attribute correctly. Then all information needed to compute any known attribute is available during parsing. This immediately implies two properties of the set K of known attributes:

I If $\underline{a} \in K$ and $\underline{a} = ex(\underline{a}_1, \ldots, \underline{a}_n)$ by AG rule $A \to \alpha$, then $\underline{a}_1, \ldots, \underline{a}_n \in K$.

II No $\underline{a} \in K$ is right dependent. We define $\underline{a}$ to be <u>right dependent</u> if it is an inherited attribute of some nonterminal A, and there is a rule $B \to \alpha A \beta$ which defines $\underline{a} = ex(\underline{a}_1, \ldots, \underline{a}_n)$ where at least one $\underline{a}_i$ is a synthesized attribute of some symbol appearing in $A\beta$.

To completely characterize K we also need to account for the last sentence of case 3 above. This is done by finding a symbolic description of the set of values assumed by $\underline{a} \in In(S)$ as we range over all parse trees.

Recall (e.g. [AhU77]) that each parse state is of the form $S = CLOSURE(BASIS(S))$ where BASIS(S) is either the initial basis $\{Start \to . S\}$, or is of the form

$$BASIS(S) = \{A \to \alpha X . \beta \mid A \to \alpha . X\beta \in S'\}$$

for some state S'. All attributes in In(S) must ultimately come from those of BASIS(S). Thus we let the <u>defining attributes</u> of S be those available on entry into state S:

a) inherited attributes of A such that $A \to \alpha . \beta$ is in BASIS(S)

b) synthesized attributes of symbols of a longest α such that $A \to \alpha . \beta$ is in BASIS(S).

Note: Every attribute of every symbol in α is considered distinct, even if symbols are repeated.

Every item in $S = CLOSURE(BASIS(S))$ is the last term of a sequence $A_0 \to \alpha_1 . A_1 \beta_1, A_1 \to . A_2 \beta_2, \ldots, A_{n-1} \to . A_n \beta_n$ where $A_0 \to \alpha_1 . A_1 \beta_1 \in BASIS(S)$. Each inherited attribute of A_i is a function of the defining attributes of S. An expression denoting this function may be obtained inductively from the expressions for A_{i-1} attributes. The set of all such expressions for S may be described as a function

$$\mathcal{E}_S: \text{In(S) } \cup \text{ defining attributes of S} \to \text{Sets of expressions in defining attributes of S}$$

$\mathcal{E}_S$ is defined recursively by:

a) $\mathcal{E}_S(\underline{a}) = \underline{a}$ if $\underline{a}$ is a defining attribute of S

b) $\mathcal{E}_S(\underline{a}) = \{ex(e_1, \ldots, e_n) \mid$ S contains an item $B \to \alpha . A\beta$, $\underline{a}$ is an inherited attribute of A defined by $\underline{a} = ex(\underline{a}_1, \ldots, \underline{a}_n)$, and $e_i \in \mathcal{E}_S(\underline{a}_i)$ for $i = 1, \ldots, n\}$

if $\underline{a} \in In(S)$

We give some examples.

1. Figure 1 with $S = \{pg \to dc.st,\ st \to .,\ st \to .st\ \underline{use}\ v\}$

$$\mathcal{E}_S(st \downarrow SET) = \{dc \uparrow SET\}$$

2. AG rules: $\langle A \downarrow \underline{a}\rangle \to 0\ \langle B \downarrow \underline{a}+1\rangle\ 0$

$\langle A \downarrow \underline{a}\rangle \to 0\ \langle B \downarrow \underline{a}+2\rangle\ 1$

$\langle B \downarrow \underline{b}\rangle \to 2$

State: $S = \{A \to 0.B0,\ A \to 0.B1,\ B \to .2\}$

Expressions for S:

$$\mathcal{E}_S(\underline{b}) = \{\underline{a}+1,\ \underline{a}+2\}$$

3. AG rules: $\langle A \downarrow \underline{a}\rangle \to 0\ \langle B \downarrow \underline{a}+1\rangle$

$\langle B \downarrow \underline{b}\rangle \to \langle B \downarrow \underline{b}+2\rangle\ 1$

$\langle B \downarrow \underline{b}\rangle \to 1$

State: $S = \{A \to 0.B,\ B \to .B1,\ B \to .1\}$

Expressions for S:

$$\mathcal{E}_S(\underline{b}) = \{\underline{a}+1,\ (\underline{a}+1)+2,\ ((\underline{a}+1)+2)+2,\ \ldots\}$$

In examples 2 and 3 the possibility of multiple values implies that $\underline{b}$ cannot be computed during parsing. We can at last state the third property of the set K of known attributes:

III If $\underline{a} \in K \cap In(S)$ for some state S, then $\mathcal{E}_S(\underline{a})$ contains only one expression.

Conversely it may be seen that if I, II and III are satisfied then every known attribute will be evaluated correctly during parsing. We define the AG to be LR-attributed if $K = \{\underline{a} \mid \underline{a}$ is any attribute $\}$ satisfies II and III, so every attribute may be evaluated during parsing. Note that this implies the AG is L-attributed.

COMPUTATION OF KNOWN ATTRIBUTES

We now describe a reasonably efficient method to find a maximal set of known attributes. Let U be the set of unknown attributes. Properties I, II and III may be restated as asserting that $U \supseteq f(U)$, where

$f(U) = U_I \cup U_{II} \cup \{\underline{a} \mid$ AG rule $A \to \alpha$ with $\underline{a} = ex(\underline{a}_1, \ldots, \underline{a}_n)$ and $\exists i\ \underline{a}_i \in U\}$

$U_I = \{\underline{a} \mid \underline{a}$ is right dependent$\}$

$U_{II} = \{\underline{a} \mid \exists S\ \underline{a} \in In(S)$ and $\#\mathcal{E}_S(\underline{a}) > 1\}$

We want a maximal K and so a minimal U which satisfies $U \supseteq f(U)$. The unique solution is the minimal fixed point of f, namely $\bigcup_{n=0}^{\infty} f^n(\emptyset)$. This is easily computed by a simple marking algorithm, given U_I and U_{II}. U_I can be found by scanning the AG rules. To compute U_{II} we replace $\mathcal{E}_S$ by the following finite version which is just as good for our purposes:

$$\mathcal{E}_S'(\underline{a}) = \begin{cases} \mathcal{E}_S(\underline{a}) & \text{if } \#\mathcal{E}_S(\underline{a}) \leq 1 \\ ? & \text{otherwise} \end{cases}$$

To compute $\mathcal{E}_S'$ we use the bottom-up algorithm which naturally corresponds to the recursive definition of $\mathcal{E}_S$, modified as follows: whenever a set with more than one element would have been obtained, replace it by ? .

Remarks

1. The parser step "compute known attributes in $In(S_m)$" amounts to evaluating $\{\mathcal{E}_S(\underline{a}) \mid \underline{a} \in In(S_m)\}$. Duplicated attribute expressions in this set need only be evaluated once. A typical example where this saves attribute copying is the following in which only one copy of attribute a is needed:

AG Rules: $\langle A \downarrow \underline{a}\rangle \to \langle E \downarrow \underline{e}\rangle$

$\langle E \downarrow \underline{e}\rangle \to \langle E \downarrow \underline{e}\rangle + \langle T \downarrow \underline{e}\rangle$

$\langle E \downarrow \underline{e}\rangle \to \langle T \downarrow \underline{e}\rangle$

$\langle T \downarrow \underline{t}\rangle \to \langle T \downarrow \underline{t}\rangle * \langle ID\rangle$

$\langle T \downarrow \underline{t}\rangle \to \langle ID\rangle$

State: $S = \{A \to .E,\ E \to .E + T,\ E \to .T,\ T \to .T * ID,\ T \to .ID\}$

Expressions for S:

$$\{\mathcal{E}_S(\underline{a}) \mid \underline{a} \in In(S)\} = \{\underline{a}\}$$

2. It could be argued that the $\mathcal{E}_S'(a)$ computation is expensive since the repeated substitution could blow up the expression's size. However an appropriate and efficient way to do this is via expression dags, used as in [AhU77] to remove common subexpressions.

3. Note that the AG is LR attributed iff $U_I = U_{II} = \emptyset$.

Extension of the Method

A more powerful analysis can be done (and is done in [Mad80a]) by classifying each occurrence of an attribute in an item of a parser state as known or unknown (so the same attribute may be known in one state and unknown in another). This classification may also be used to split the LR states. This approach can compute more attributes during parsing than the one presented here, but involves a more complex AG analysis.

CONCLUSIONS

We have described methods which make it possible, when given an attribute grammar and an LR parser generator, to produce an augmented parser which evaluates a great many attributes during parsing (all of them, if the AG is LR-attributed). This achieves by automatic means the effect of the introduction of copy rules. The classification of attributes into "known" and "unknown" should greatly facilitate rule splitting. Given a conflict parse table entry and a list of known attributes, construction of the appropriate disambiguating predicates should be straightforward (provided the known attributes make this possible).

The method is not restricted to L-attributed grammars, although right dependencies give rise to unknown attributes. Further the class of LR-attributed AGs seems quite large compared to previous classes. It extends Rowland's method [Row77] by not treating all inherited attributed of left corner symbols as unknown. It also extends [Wat77a] and [PuB79] by allowing non-trivial attribute expressions in places where copy rules would cause conflicts, as in example 2 after the definition of $\mathcal{E}_S$. Every RA-attributed AG as defined in [RäU80] is both LR(k) and LR-attributed since condition (RA3) there implies that $\#\mathcal{E}_S(a) \leq 1$ for all $S, \underline{a}$. The LR-attributed class is larger since the last example of [RäU80] is LR-attributed, even with $F \rightarrow (E)$.

The AG analysis and parser construction algorithms are relatively simple and appear likely to be quite efficient in practice.

[Mil77] Milton, D.: Syntactic specification and analysis with attributed grammars. Computer Science Technical Report #304, University of Wisconsin-Madison (1977).

[MiF79] Milton, D. and Fischer, C.: LL(k) parsing for attributed grammars. In: Automata, Languages and Programming, (ed. Maurer, H.A.) 422-430. Lecture Notes in Computer Science, vol. 7. Berlin-Heidelberg-New York, Springer (1979).

[PuB79] Purdom, P. and Brown, C.A.: Semantic Routines and LR(k) Parsers. Tech. Report 83, Computer Science Department, Indiana Univ. (1979).

[Row77] Rowland, B.: Combining parsing and evaluation for attributed grammars. Computer Science Technical Report # 308, University of Wisconsin-Madison (1977).

[RäU80] Räihä, K. and Ukkonen, E.: One-pass evaluation of attribute grammars using recursive parsing techniques. To appear in IFIP Proceedings, 1980.

[Sch76] Schulz, W.A.: Semantic analysis and target language synthesis in a translator. Ph.D. Thesis, University of Colorado, Boulder, Colorado (1976).

[Wat77a] Watt, D.A.: The parsing problem for affix grammars. In: Acta Informatica 8, 1-20 (1977).

[Wat77b] Watt, D.A.: An extended attribute grammar for Pascal. Report no. 11, Computing Department, University of Glasgow (1977). Also in: SIGPLAN Notices 14, no. 2, 60-74 (1979).

[Wat80] Watt, D.A.: Rule splitting and attribute-directed parsing, These proceedings.

REFERENCES

[AhU77] Aho, A. and Ullman, J.: Principles of Compiler Design. Addison-Wesley (1977).

[Boc76] Bochmann, G.V.: Semantic evaluation from left to right. In: Comm. ACM 19, 55-62 (1976).

[CoH79] Cohen, R. and Harry, E.: Automatic Generation of Near-Optimal Linear-Time Translation for Non-circular Attribute Grammars. In: Conference Record of 6th ACM Symposium on Principles of Programming Languages. 121-134 (1979).

[Dem77] Demers, A.: Generalized left-corner parsing. In: Conference Record of the 4th ACM Symposium on Principles of Programming Languages, 170-182 (1977).

[KeW76] Kennedy, K. and Warren, S.K. Automatic Generation of Efficient Evaluations for Attribute Grammars. In: Conference Record of 3rd ACM Symposium on Principles of Programming Languages, 32-49 (1976).

[Knu68] Knuth, D.E.: Semantics of context-free languages. In: Mathematical Systems Theory 2, 127-145 (1968).

[Knu71] Knuth, D.E.: Semantics of context-free languages: correction. In: Mathematical Systems Theory 5, 95-96 (1971).

[LRS74] Lewis, P.M., Rosenkrantz, D.J. and Stearns, R.E.: Attributed Translations. In: Journal of Computer and System Sciences 9, 279-307 (1974).

[Mad80a] Madsen, M.: Parsing Attribute Grammars. Thesis, University of Aarhus, Denmark (1980).

[Mad80b] Madsen, O.L.: On defining semantics by means of extended attribute grammars. Report DAIMI PB-109, 65 pp, University of Aarhus, Denmark, (1980).