RULE SPLITTING and ATTRIBUTE-DIRECTED PARSING

David A. Watt

Computing Science Department
University of Glasgow
Glasgow G12 8QQ
Scotland

Abstract

Rule splitting is a phenomenon, most clearly exhibited by attribute grammars and affix grammars, in which the syntactic structure of a phrase is constrained by its attributes. In this paper, rule splitting is illustrated by examples taken from real programming languages, and two varieties of rule splitting are identified and formalized. Implementations of rule splitting (attribute-directed parsing) are demonstrated for top-down and bottom-up parsers, both one-pass and multi-pass. Finally, the problems of exploiting rule splitting in a compiler writing system based on attribute grammars are explored.

1. Introduction

Attribute grammars and affix grammars are currently the most promising tools available in the design of compiler writing systems. This is because these grammars are clean extensions of context-free grammars, capable of specifying the (context-sensitive) syntax and semantics of programming languages, in a way which allows the existing body of context-free parsing techniques to continue to be exploited.

In a conventional compiler constructed from an attribute grammar, (context-free) parsing and attribute propagation are distinct phases, with no feedback from the second to the first.

A number of papers [Bochmann 76, Watt 74b, Watt & Madsen 79] have observed that the phrase structure of some programming language contructs is constrained by certain attributes, a phenomenom known as rule splitting. A natural consequence of this, in the compiler, is to allow these attributes to influence the behaviour of the parser, i.e. to allow limited feedback from the attribute propagation phase, a technique known as attribute-directed parsing.

None of the papers mentioned, however, attempts a systematic study of rule splitting and attribute-directed parsing. Such is the purpose of this paper.

2. Attribute grammars, notation, terminology and conventions

2.1. Summary of attribute grammars

An attribute grammar [Knuth 68] or affix grammar [Koster 71a] (AG) is a context-free grammar (CFG) in which each terminal and nonterminal symbol is augmented by a fixed number of attributes, with fixed domains. Different instances of the same symbol in a syntax tree may have different attributes, and the attributes are able to contain information obtained from other nodes

of the syntax tree.

A distinction is made between inherited and synthesized attributes. Consider a symbol S and a phrase p derived from S. Each inherited attribute of S is supposed to convey information about the context of p. Each synthesized attribute of S is supposed to convey information about the phrase p itself in the given context.

The attributes can be used to specify context-sensitive constraints on a language with a context-free phrase structure. Each AG rule is basically a CF production rule augmented by
(a) a constraint, or a predicate which must be satisfied by certain attributes in each application of this rule, and
(b) an evaluation rule, specifying the evaluation of certain attributes in terms of others.

2.2. Notation and terminology

Let the tuple of inherited attributes of a symbol S be denoted by inh(S), and let the tuple of synthesized attributes of S be denoted by syn(S). It is convenient also to extend inh and syn to sequences of symbols:

$$inh^*(S_1 \ldots S_n) = (inh(S_1), \ldots, inh(S_n))$$
$$syn^*(S_1 \ldots S_n) = (syn(S_1), \ldots, syn(S_n))$$

Then the constraint and evaluation rule associated with the production rule N -> w are functions as follows:

constraint: $inh(N) \times syn^*(w) \rightarrow Boolean$
evaluation rule: $inh(N) \times syn^*(w) \rightarrow inh^*(w) \times syn(N)$

inh(N) and $syn^*(w)$ are called defined attributes. $inh^*(w)$ and syn(N) are called applied attributes. Thus the constraint is a predicate on the defined attributes, and the evaluation rule maps the defined attributes on to the applied attributes. The evaluation rule may be a partial function, provided it is defined at all points where the constraint evaluates to true.

We write AG rules using a notation based on BNF, e.g.:

```
<assignment ↓ ENV>  ::=
        <variable ↓ ENV1 ↑ TYPE1> ":="
            <expression ↓ ENV2 ↑ TYPE2>
                where TYPE1 = TYPE2
                evaluate ENV1 <- ENV, ENV2 <- ENV                    (2.1)
```

Each inherited attribute is prefixed by a downward arrow (↓), and each synthesized attribute is prefixed by an upward arrow (↑). ENV, ENV1, ENV2, TYPE1, TYPE2 are attribute variables and they stand for the various attribute occurrences in this AG rule. The evaluation rule is introduced by "evaluate" and the constraint by "where"; both are expressed in terms of the attribute variables.

When an applied attribute is a simple copy of a defined attribute, we abbreviate by simply using the same attribute variable for both. In (2.1) ENV1 and ENV2 are simple copies of ENV, so we abbreviate the AG rule by replacing each by ENV:

```
<assignment ↓ ENV>  ::=
        <variable ↓ ENV ↑ TYPE1> ":="
            <expression ↓ ENV ↑ TYPE2>
                where TYPE1=TYPE2                                    (2.2)
```

2.3. Predicate projection

Definition 2.1. Let $p(x_1,\ldots,x_n)$ be a predicate dependent on variables x_1, ..., x_n. Then we define $p'(x_i,x_j,\ldots)$ to be the strongest predicate dependent only on the variables x_i, x_j, ... but implied by $p(x_1,\ldots,x_n)$.

For example:

if p(x,y,z) = x=0 and y<z
then
p'(x) = x=0
p'(y) = true
p'(z) = true
p'(y,z) = y<z

2.4. Discriminated unions

The definition of AGs places no restrictions on the domains chosen for the attributes. In a formal definition of the syntax of Pascal by an extended attribute grammar [Watt 79], the domains were based on the abstract data structures of [Hoare 72]: Cartesian products, discriminated unions, sets, maps and sequences.

Of these, discriminated unions are particularly useful, indeed they may be viewed as the most fundamental domain type [Madsen 80]. They are also particularly important in the context of rule splitting, so a definition here follows.

Definition 2.2. If T_1, ... T_n are domains (or Cartesian products of domains) and g_1, ..., g_n are distinct names, then

$$U = (g_1(T_1) \mid ..... \mid g_n(T_n))$$

is a discriminated union with selectors g_1, ..., g_n. If any T_i is void, then we abbreviate $g_i(T_i)$ to g_i.

For every i=1,...,n, and for every a_i in T_i, $g_i(a_i)$ is in U. These g_i are the composition functions for the discriminated union U.

For each i=1,...,n, we also define a predicate, is-g_i, and a partial inverse function, g_i^{-1}, as follows:

$is\text{-}g_i(x)$ = there exists y such that $x = g_i(y)$

$g_i^{-1}(x)$ = if there exists y such that $x = g_i(y)$
then y
else undefined

2.5. Conventions

We use the following conventions throughout:
(a) M and N (possibly subscripted) stand for nonterminals;
(b) u, v, w, x and y (possibly subscripted) stand for sequences of nonterminals and terminals;
(c) p and q (possibly subscripted) stand for predicates.

3. Examples of rule splitting

All the examples quoted here are taken from real programming languages, but simplified to remove unnecessary detail. Nearly all the nonterminals involved have an inherited attribute (their "environment") which is a map from names to modes.

Example 1.

```
<actual parameter ↓ ENV ↓ PARM>  ::=
        <expression ↓ ENV ↑ TYPE>
                where is-value(PARM) and
                      TYPE=value^-1(PARM)                    (3.1a)
      | <variable ↓ ENV ↕ TYPE>
                where is-result(PARM) and
                      TYPE=result^-1(PARM)                   (3.1b)
```

```
<expression ↓ ENV ↑ TYPE>  ::=
            <variable ↓ ENV ↑ TYPE>                    (3.1c)
          | ..........................
```

Here the second attributes of <expression> and <variable> both lie in some domain Type, and the second attribute of <actual parameter> lies in the discriminated union domain

```
Parameter = ( value(Type) | result(Type) )
```

The latter attribute defines the parameter mechanism and the type of the corresponding formal parameter; the attribute is therefore inherited. If the formal parameter is a value-parameter, the actual parameter may be any expression of the same type; if the formal parameter is a result-parameter, the actual parameter must be a variable of the same type.

This is perhaps the example par excellence of rule splitting; the value of the attribute constrains the phrase structure of the actual parameter; and the underlying CFG is actually ambiguous:

```
<actual parameter>  ⇒  <expression>
                    ⇒  <variable>

<actual parameter>  ⇒  <variable>
```

The AG is unambiguous, however, because, for any given value of the second attribute of <actual parameter>, only one of the constraints in (3.1a) and (3.1b) can be satisfied, and therefore only one of these alternative derivations is possible.

Example 2.

```
<primary ↓ ENV ↑ TYPE>  ::=
            <variable ↓ ENV ↑ TYPE>                    (3.2a)
          | <constant ↓ ENV ↑ TYPE>                    (3.2b)
          | ..........................
```

```
<variable ↓ ENV ↑ TYPE>  ::=
            <identifier ↓ ENV ↑ MODE>
                    where is-var(MODE)
                    evaluate TYPE ← var^-1(MODE)                    (3.2c)
        |   ....................................

<constant ↓ ENV ↑ TYPE>  ::=
            <identifier ↓ ENV ↑ MODE>
                    where is-const(MODE)
                    evaluate TYPE ← const^-1(MODE)                  (3.2d)
        |   ....................................
```

Here <identifier> has a synthesized attribute in the domain

```
Mode = ( const(Type) | var(Type) | proc(Plan) | ..... )
```

which specifies whether the identifier is a constant-identifier, a variable-identifier, a procedure-identifier, etc. Here again, the underlying CFG is ambiguous:

```
<primary>  ⇒  <variable>
           ⇒  <identifier>

<primary>  ⇒  <constant>
           ⇒  <identifier>
```

The synthesized attribute of <identifier>, however, can be used to eliminate one or other of these derivations.

Example 3.

```
<statement ↓ ENV>  ::=
          <variable ↓ ENV ↑ TYPE>  ":="
              <source ↓ ENV ↓ TYPE>                                 (3.3a)
      |   <identifier ↓ ENV ↑ MODE>
              "(" <parameter ↓ ENV ↓ PARAMETER> ")"
                  where is-proc(MODE)
                  evaluate PARAMETER ← proc^-1(MODE)                (3.3b)
```

```
<parameter ↓ ENV ↓ PARAMETER>  ::=
          <source ↓ ENV ↓ TYPE>
                  where is-value(PARAMETER)
                  evaluate TYPE ← value^-1(PARAMETER)                    (3.3c)
        | ........................................

<variable ↓ ENV ↑ TYPE>  ::=
          <identifier ↓ ENV ↑ MODE>
                  where is-scalar(MODE)
                  evaluate TYPE ← scalar^-1(MODE)                        (3.3d)
        | <identifier ↓ ENV ↑ MODE>
              "(" <source ↓ ENV ↓ SUBTYPE> ")"
                  where is-array(MODE)
                  evaluate TYPE ← array^-1(MODE),
                           SUBTYPE ← int                                 (3.3e)
```

Here the second attribute of <identifier> is in the domain

```
Mode = ( scalar(Type) | array(Type) |
         proc(Parameter) | ..... )
where
Parameter = ( value(Type) | ..... )
```

In this more complicated example there is no ambiguity but there are LL and LR parsing conflicts:

```
<statement>  ⇒  <variable> := <source>
             ⇒  <identifier> := <source>

<statement>  ⇒  <variable> := <source>
             ⇒  <identifier> ( <source> ) := <source>

<statement>  ⇒  <identifier> ( <parameter> )
             ⇒  <identifier> ( <source> )
```

(A similar example involving function-designators would, however, be ambiguous.) If known at parse-time, the synthesized attribute of <identifier> can be used to resolve the conflicts.

Example 4.

```
<factor ↓ ENV ↑ TYPE>  ::=
            <primary ↓ ENV ↑ TYPE1>
                "**"  <primary ↓ ENV ↑ TYPE2>
                    where is-int(TYPE1) and
                          is-int(TYPE2)
                    evaluate TYPE ← int                    (3.4a)
          |  <primary ↓ ENV ↑ TYPE1>
                "**"  <primary ↓ ENV ↑ TYPE2>
                    where is-real(TYPE1) and
                          is-int(TYPE2)
                    evaluate TYPE ← real                   (3.4b)
          |  <primary ↓ ENV ↑ TYPE1>
                "**"  <primary ↓ ENV ↑ TYPE2>
                    where is-real(TYPE1) and
                          is-real(TYPE2)
                    evaluate TYPE ← real                   (3.4c)
```

Here the synthesized attributes of <factor> and <primary> are in the domain

```
Type = ( int | real | ..... )
```

This example is complicated in a different sense. There are three different derivations

```
<factor>  ⇒+  <primary> ** <primary>
```

The correct choice is based on examining the synthesized attributes of both instances of <primary>.

Example 5.

```
<factor ↓ ENV ↑ TYPE>  ::=
        <primary ↓ ENV ↑ TYPE1>
            "**"  <primary ↓ ENV ↑ TYPE2>
            <where compatible ↓ TYPE1 ↓ TYPE2 ↑ TYPE>      (3.5a)
```

```
<where compatible ↓ TYPE1 ↓ TYPE2 ↑ TYPE>  ::=
        <empty>
                where is-int(TYPE1) and
                      is-int(TYPE2)
                evaluate TYPE ← int                          (3.5b)
      | <empty>
                where is-real(TYPE1) and
                      is-int(TYPE2)
                evaluate TYPE ← real                         (3.5c)
      | <empty>
                where is-real(TYPE1) and
                      is-real(TYPE2)
                evaluate TYPE ← real                         (3.5d)
```

This example enforces the same type compatibility as Example 4, but it is factored out of (3.5a) by means of a grammatically defined predicate <where compatible>, which derives only the empty string and which exists only to enforce certain relationships among its attributes. This it does by rule splitting (3.5b-d), based on its two inherited attributes.

4. Characterization of rule splitting

The examples in the previous section have certain features in common which help us to characterize more formally what exactly rule splitting is. The most salient common feature is that, in each case, it was possible to eliminate all but one of several alternative derivations by inspection of either the inherited attributes of the common symbol on the left side of a rule group or the synthesized attributes of a common sequence of symbols of the right sides of several rules. This observation leads us to characterize two forms of rule splitting.

Since evaluation rules play no part in rule splitting, they are ignored in the following. The key role is played by the constraints and their

projections (Definition 2.1).

Definition 4.1. Inherited rule splitting is exhibited by a group of rules

$N \rightarrow w_1$ where $p_1(inh(N), syn^*(w_1))$
$N \rightarrow w_2$ where $p_2(inh(N), syn^*(w_2))$
....

(being all the rules with N on the left side) if

$$p_i'(inh(N)) \text{ implies } (\text{not } p_j'(inh(N))) \text{ for all } j \neq i \qquad (4.1)$$

i.e. if $p_1'(inh(N))$, $p_2'(inh(N))$, ... are all mutually exclusive.

Inherited rule splitting is illustrated by Examples 1 and 5. In Example 1 it is exhibited by rules (3.1a,b):

N = <actual parameter>
w_1 = <expression>
w_2 = <variable>
p_1'(ENV,PARM) = is-value(PARM)
p_2'(ENV,PARM) = is-result(PARM)
is-value(PARM) implies (not is-result(PARM))

In Example 5 inherited rule splitting is exhibited by rules (3.5b-d):

N = <where compatible>
w_1 = <empty>
w_2 = <empty>
w_3 = <empty>
p_1'(TYPE1,TYPE2) = is-int(TYPE1) and is-int(TYPE2)
p_2'(TYPE1,TYPE2) = is-real(TYPE1) and is-int(TYPE2)
p_3'(TYPE1,TYPE2) = is-real(TYPE1) and is-real(TYPE2)

Definition 4.2. Synthesized rule splitting is exhibited by the rules

$N_1 \rightarrow v_1 w x_1$ where $p_1(inh(N_1), syn^*(v_1 w x_1))$
$N_2 \rightarrow v_2 w x_2$ where $p_2(inh(N_2), syn^*(v_2 w x_2))$
....

(where N_1, N_2, ... are not necessarily the same nonterminal) if there exist M, u_1, u_2, ..., y_1, y_2, ... such that

$$u_i v_i = u_j v_j \text{ for all } i,j=1,2,...$$
$$\text{and } M \Rightarrow^* u_j N_i y_i \Rightarrow u_i v_i w x_i y_i \text{ for all } i=1,2....$$
$$\text{and } p_i'(syn^*(w)) \text{ implies (not } p_j'(syn^*(w))) \text{ for all } j \neq i \qquad (4.2)$$

Synthesized rule splitting is illustrated by Examples 2, 3 and 4. In Example 2 it is exhibited by rules (3.2c,d):

w = <identifier>
N_1 = <variable>, $v_1 = x_1$ = <empty>
N_2 = <constant>, $v_2 = x_2$ = <empty>
p_1'(MODE) = is-var(MODE)
p_2'(MODE) = is-const(MODE)
M = <primary>
$u_1 = y_1$ = <empty>
$u_2 = y_2$ = <empty>

In Example 3 synthesized rule splitting is exhibited by rules (3.3b,d,e):

w = <identifier>
N_1 = <statement>, v_1 = <empty>, x_1 = (<parameter>)
N_2 = <variable>, v_2 = <empty>, x_2 = <empty>
N_3 = <variable>, v_3 = <empty>, x_3 = (<source>)
p_1'(MODE) = is-proc(MODE)
p_2'(MODE) = is-scalar(MODE)
p_3'(MODE) = is-array(MODE)
M = <statement>
u_1 = <empty>, y_1 = <empty>
u_2 = <empty>, y_2 = := <source>
u_3 = <empty>, y_3 = := <source>

In Example 4 synthesized rule splitting is exhibited by rules (3.4a-c):

w = <primary> ** <primary>
N_1 = <factor>, $v_1 = x_1$ = <empty>
N_2 = <factor>, $v_2 = x_2$ = <empty>
N_3 = <factor>, $v_3 = x_3$ = <empty>
p_1'(TYPE1,TYPE2) = is-int(TYPE1) and is-int(TYPE2)
p_2'(TYPE1,TYPE2) = is-real(TYPE1) and is-int(TYPE2)
p_3'(TYPE1,TYPE2) = is-real(TYPE1) and is-real(TYPE2)
M = <factor>
$u_1 = y_1$ = <empty>
$u_2 = y_2$ = <empty>
$u_3 = y_3$ = <empty>

5. Attribute-directed parsing

In a compiler constructed from an AG, analysis of an input string classically proceeds in two distinct phases:
(1) a conventional CF parser is used to construct a syntax tree from the input string;
(2) the nodes of the syntax tree are "decorated" by attributes in accordance with the evaluation rules of the AG, and any constraints on the attributes are tested.

If the AG is L-attributed [Bochmann 76, Lewis et al 74], phase 2 may be performed in a single left-to-right pass over the syntax tree. For a larger class of AGs, phase 2 may be performed in a fixed number of passes over the syntax tree, the set of attributes to be evaluated during each pass being determined at compiler-construction-time [Bochmann 76, Jazayeri & Walter 75, etc.].

{For still larger classes of AGs, a decision on the order of evaluation of the attributes can be delayed until phase 2 itself. The system DELTA [Lorho 75] handles all AGs containing no circularities in the evaluation rules. The

system NEATS [Jespersen et al 79, Madsen 80] even relaxes this restriction. These techniques are neglected here since they normally preclude the kind of attribute-directed parsing we are about to describe. However, some hybrid system is conceivable in which a subset of the attributes are evaluated during phase 1.}

For a large subclass of the L-attributed AGs [Watt 77], phases 1 and 2 can be merged in time. A variety of one-pass parsing methods for L-attributed AGs have been proposed or implemented; these methods include top-down [Bochmann & Ward 75, Koster 71a, Koster 71b], bounded-context [Crowe 72], precedence [Lecarme & Bochmann 74], and LR [Watt 74b].

All these methods are essentially CF parsing methods augmented by some mechanism for evaluating, testing and distributing the attributes. The attributes do not in any way influence the flow of control in the parser.

Rule splitting, however, makes it feasible for the attributes concerned to influence the behaviour of the parser. A choice among several alternative parsing actions may be made by testing these attributes, without having to invoke the usual look-ahead techniques of CF parsing. This can resolve CF parsing conflicts and even ambiguities. Thus, for example, the underlying CFG need not necessarily be LL(1) for the recursive-descent parsing method to be adopted. This enhancement of CF parsing is called attribute-directed parsing.

In this section we first demonstrate the implementation of one-pass attribute-directed parsing in a recursive-descent parser and in an LR parser, assuming that the AG is L-attributed. Then we generalize to the multi-pass case.

5.1. Attribute-directed recursive-descent parsing

A CF recursive-descent parser consists of one parameterless procedure, N, for each nonterminal N of the CFG. The job of procedure N is to parse a phrase which can be derived from the nonterminal N. The body of procedure N is obtained by transcription of the N-rules of the CFG.

In a one-pass recursive-descent parser for an L-attributed AG, each procedure N is augmented by parameters which convey the attributes of the

nonterminal N: an input-parameter for each inherited attribute and an output-parameter for each synthesized attribute. The body of procedure N is augmented by the evaluation rules and constraints associated with the N-rules of the AG. [Bochmann & Ward 75, Koster 71a, Koster 71b]

Such a parser can easily be made to exploit inherited rule splitting. (Refer to Definition 4.1.) The rule group

$N \rightarrow w_1$ where $p_1(inh(N), syn^*(w_1))$
$N \rightarrow w_2$ where $p_2(inh(N), syn^*(w_2))$
....

is transcribed to the procedure

```
procedure N ( in  inh(N);
              out syn(N) );
   begin
   if p1'(inh(N)) then
      parse w1
   else if p2'(inh(N)) then
      parse w2
   ..........................
      .........
   else        {this escape clause may be unnecessary}
      context_sensitive_error
   end
```

In Example 1, rules (3.1a,b) would be transcribed as follows:

```
procedure actual_parameter ( in ENV  : Environment;
                             in PARM : Parameter   );
   var TYPE : Type;
   begin
   if is-value(PARM) then
      begin
      expression (ENV, TYPE);
      ensure (TYPE=value^-1(PARM))
      end
   else if is-result(PARM) then
      begin
      variable (ENV, TYPE);
      ensure (TYPE=result^-1(PARM))
      end
   end
```

In Example 5, rules (3.5b-d) would be transcribed as follows:

```
procedure where_compatible ( in  TYPE1, TYPE2 : Type;
                             out TYPE         : Type );
   begin
   if is-int(TYPE1) and is-int(TYPE2) then
      TYPE := int
   else if is-real(TYPE1) and is-int(TYPE2) then
      TYPE := real
   else if is-real(TYPE1) and is-real(TYPE2) then
      TYPE := real
   else
      context_sensitive_error
   end
```

Grammatically defined predicates such as <where compatible> are extremely useful in language definitions, e.g. [Watt 79]. The latter example demonstrates that they are easily implemented without any special techniques other than attribute-directed parsing.

A general implementation of synthesized rule splitting is not possible in a top-down parser. But consider the special case where all the N-rules are of the form:

$N \rightarrow wx_1$ where $p_1(inh(N), syn^*(wx_1))$
$N \rightarrow wx_2$ where $p_2(inh(N), syn^*(wx_2))$
....

such that

$p_i'(syn^*(w))$ implies (not $p_j'(syn^*(w))$) for all $j \neq i$ (5.1)

This rule group can be transcribed as follows:

```
procedure N ( in  inh(N);
              out syn(N) );
   begin
   parse w, and thereby deduce syn*(w);
   if p1'(syn*(w)) then
      parse x1
   else if p2'(syn*(w)) then
      parse x2
   ......................
      ........
   else        {this escape clause may be unnecessary}
      context_sensitive_error
   end
```

In Example 4, rules (3.4a-c) would be transcribed as follows:

```
procedure factor ( in  ENV  : Environment;
                   out TYPE : Type        );
   var TYPE1, TYPE2 : Type;
   begin
   primary (ENV, TYPE1);
   accept ("**");
   primary (ENV, TYPE2);
   if is-int(TYPE1) and is-int(TYPE2) then
      TYPE := int
   else if is-real(TYPE1) and is-int(TYPE2) then
      TYPE := real
   else if is-real(TYPE1) and is-real(TYPE2) then
      TYPE := real
   else
      context_sensitive_error
   end
```

5.2. Attribute-directed LR parsing

In a CFG G, if $Z \Rightarrow^* vNx \Rightarrow vwx$ (where Z is the distinguished nonterminal of G, N->w is a production rule of G, and x is a string of terminals), and if #(N->w) is a special symbol uniquely associated with the production rule N->w, then vw#(N->w) is a characteristic string of vwx. The LR parsing machine of G is the deterministic finite-state machine which accepts only the characteristic strings of G. The LR parsing machine has terminal-transitions, nonterminal-transitions, and reduce-transitions (those labelled by the special #-symbols). The LR parsing machine is used in conjunction with a stack on which are stored (state, symbol) pairs. [DeRemer 71]

A one-pass LR parser for an L-attributed AG uses a second stack, the attribute stack. Immediately prior to parsing a symbol, its inherited attributes are placed at the top of the attribute stack; and parsing the symbol has the effect of stacking its synthesized attributes immediately above its inherited attributes. The LR parsing machine is augmented by special transitions which specify either the copying of attributes to the top of the attribute stack, or the application of an evaluation rule, or the testing of a constraint. [Watt 74a, Watt 74b]

These special actions in general may introduce parsing conflicts even when the underlying CFG is LR(k). Such a conflict arises if any special transition leads out of a state from which there also leads a terminal-transition, a reduce-transition, or another special transition (unless the conflict can be resolved by the usual look-ahead).

Rule splitting, however, gives rise to a state from which there lead several special transitions which specify the testing of mutually exclusive constraints (and no other special transitions, no terminal-transitions and no reduce-transitions). This type of state has been called a multi-predicate state [Watt 74a]. The action taken by the parser in a multi-predicate state is simply to test each of the constraints (in any order or in parallel), and traverse the transition corresponding to the one constraint which is satisfied.

Inherited rule splitting gives rise to the situation illustrated in Figure 1(a), provided the constraints on the inherited attributes of the common left-side nonterminal, N, are tested before parsing the right side of an N-rule. Figures 1(b) and 1(c) illustrate Examples 1 and 5 respectively.

Synthesized rule splitting gives rise to the situation illustrated in Figure 2(a), provided the constraints on the synthesized attributes of the common right-side sequence of symbols, w, are tested immediately after parsing w. Figures 2(b), 2(c) and 2(d) illustrate Examples 2, 3 and 4 respectively.

When we use an LR parser, we can generalize the definition of synthesized rule splitting, by changing the first part of (4.2) to read:

$u_i v_i$ and $u_j v_j$ access the same state in the LR parsing machine, for all $i,j=1,2,\ldots$

In each case, the effect of rule splitting is a multi-predicate state which directs the parser to one of several different states depending on the values of the attributes concerned (inh(N), or $syn^*(w)$, respectively). The multi-predicate state in the LR parser performs the same role as the cascade of tests in the procedure of the recursive-descent parser.

5.3. The multi-pass case

If the AG is such that all the attributes can be evaluated and tested in n>1 passes over the syntax tree, then the value of n, and the set of attributes which can be evaluated and tested during each pass, can be determined from the AG [e.g. Bochmann 76]. Only the first-pass attributes can be evaluated and tested during parsing. Thus attribute-directed parsing can use only these first-pass attributes.

Let $inh_i(S)$ and $syn_i(S)$ be those inherited and synthesized attributes of a symbol (or sequence of symbols) S which are determined during the i'th pass. {Thus $inh_1(S)$, ..., $inh_n(S)$ form a partition of inh(S); and $syn_1(S)$, ..., $syn_n(S)$ form a partition of syn(S).}

Then inherited rule splitting can be exploited only if (refer to Definition 4.1):

$$p_i'(inh_1(N)) \text{ implies } (\text{not } p_j'(inh_1(N))) \text{ for all } j \neq i \qquad (5.2)$$

This condition is stronger than (4.1).

Synthesized rule splitting can be exploited only if (refer to Definition 4.2):

$$\begin{aligned} & u_i v_i = u_j v_j \text{ for all } i,j=1,2,... \\ \text{and } & M \Rightarrow^* u_i N_i y_i \Rightarrow u_i v_i w x_i y_i \text{ for all } i=1,2,... \\ \text{and } & p_i'(syn_1^*(w)) \text{ implies } (\text{not } p_j'(syn_1^*(w))) \text{ for all } j \neq i \end{aligned} \qquad (5.3)$$

The third part of (5.3) is stronger than the third part of (4.2).

The implementations of attribute-directed parsing in the multi-pass case are similar to those for the single-pass case (sections 5.1 and 5.2), except for the substitutions of inh_1 and syn_1 for inh and syn respectively.

6. Implications for a compiler writing system

The following problems will be encountered in attempting to detect and exploit rule splitting in a compiler writing system (CWS):
(1) detecting potential cases of synthesized rule splitting (this should be trivial for inherited rule splitting);
(2) determining a predicate projection p' (Definition 2.1) from a constraint p associated with an AG rule;
(3) determining whether given predicates p_1', p_2', ... are mutually exclusive.

These problems are, in general, unsolvable, but we offer partial solutions which should be satisfactory in practice; at least these solutions are adequate for handling all the examples of rule splitting in section 3. We also suggest an interactive CWS which could seek human assistance in occasional situations where the partial solutions are inadequate.

6.1. Detecting potential cases of rule splitting

If a nonterminal N has at least one inherited attribute, and if each of the N-rules has a constraint which depends on that attribute, then we have a potential case of inherited rule splitting.

We can similarly detect a potential case of synthesized rule splitting of the restricted sort (5.1) which can be handled by a top-down parser.

Because the general definition of synthesized rule splitting is more complicated (4.2), and since it can be handled only by a bottom-up parser anyway, it is best to detect potential cases by actually constructing an LR parser and seeing whether multi-predicate states appear. It is necessary to adopt some consistent strategy as to when constraints are tested. The simplest strategy is to test each constraint as soon as all the attributes on which it depends are known.

6.2. Determining predicate projections

Suppose we are given a predicate p defined as the conjunction of a set of simpler predicates:

$$p(x_1,\ldots,x_n) = q_1(x_1,\ldots,x_n) \text{ and } \ldots\ldots \text{ and } q_m(x_1,\ldots,x_n) \qquad (6.1)$$

where each q_i actually depends on only some of the variables $x_1, \ldots, x_n$. Suppose that we wish to determine $p'(x_i,x_j,\ldots)$.

Let $q(x_i,x_j,\ldots)$ be the conjunction of those predicates $q_1, \ldots, q_m$ which actually depend on the variables $x_i, x_j, \ldots$; then $p(x_1,\ldots,x_n)$ implies $q(x_i,x_j,\ldots)$. q is less strong than p', but otherwise q satisfies the requirements of Definition 2.1. Thus q may serve as an approximate solution to p'.

Constraints associated with AG rules are quite likely to be presented in the form (6.1). This is indeed the case in all the examples of section 3, and this partial solution is in fact accurate in all these examples. In Example 1, rule (3.1a):

p(ENV,PARM,TYPE) = is-value(PARM) and TYPE=value^{-1}(PARM)
q(PARM) = is-value(PARM)
p'(PARM) = is-value(PARM)

6.3. Determining whether given predicates are exclusive

In general, knowledge of the properties of the attribute domains over which the given predicates are defined is needed to determine whether the predicates are exclusive.

For example, we know from the properties of the discriminated union

$$U = (g_1(T_1) \mid \ldots\ldots \mid g_n(T_n))$$

(Definition 2.2) that the predicates is-g_1, ..., is-g_n are mutually exclusive. This is sufficient to establish the mutual exclusiveness of the relevant

predicate projections in all the examples of section 3.

6.4. An interactive CWS

If the partial solutions outlined above are found to be inadequate, the CWS could be made interactive, seeking human assistance in unclear situations. The compiler writer would supply an AG defining the language to be implemented; the CWS would look for potential cases of rule splitting, deal with the clear cases itself as outlined above, and refer unclear cases back to the compiler writer for a decision as to whether rule splitting is actually present and whether it should be exploited.

This leads to a point we have ignored up to now. Even where rule splitting is present, it is unnecessary to exploit it in the absence of any CF parsing conflict or ambiguity. Indeed, it may then even be undesirable to exploit it since it may adversely influence syntactic error reporting and recovery. This is another reason why an interactive CWS may be a good way of dealing with rule splitting.

7. Conclusions

This paper has attempted a systematic study of the phenomenon of rule splitting and the associated implementation technique of attribute-directed parsing. A number of realistic examples of rule splitting were given. Two kinds of rule splitting, "inherited" and "synthesized", were formalized. Their implementations in top-down and bottom-up attribute-directed parsers were demonstrated, for both the single-pass case and the multi-pass case. Finally, the implications of all this for compiler writing systems were discussed, pointing out some problems which are, in general, unsolvable and require either a pragmatic or an interactive approach.

Attribute grammars have been used as a medium for our discussion because

they facilitate our definitions of rule splitting and our descriptions of the corresponding implementation techniques. We conclude by referring readers to the "extended attribute grammars" of [Watt & Madsen 79], which allow instances of rule splitting to be exhibited rather clearly. Several examples of rule splitting in the context of a complete grammar may be found in [Watt 79].

References

Bochmann 76.

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

Bochmann & Ward 75.

Bochmann, G.V., Ward, P.: Compiler writing systems for attribute grammars. Département d'Informatique, Université de Montréal, Publication #199, July 1975

Crowe 72.

Crowe, D.: Constructing parsers for affix grammars. Comm. ACM 15, 728-734 (1972)

Eriksen et al 79.

Eriksen, S.H., Kristensen, B.B., Madsen, O.L.: The BOBS-system. Aarhus University, Report DAIMI PB-71 (revised version), 1979

Ganzinger et al 77.

Ganzinger, H., Ripken, K., Wilhelm, R.: Automatic generation of optimizing multipass compilers. In: Proc. IFIP 77 Congress, pp. 535-540. Amsterdam: North-Holland 1977

Hoare 72.

Hoare, C.A.R.: Notes on data structuring. In: Structured Programming (O.-J. Dahl, E.W. Dijkstra, C.A.R. Hoare), pp. 83-174. London-New York: Academic

Press 1972

Jazayeri & Walter 75.

Jazayeri, M., Walter, K.G.: Alternating semantic evaluation. Proc. ACM Annual Conference, Minneapolis, 1975

Jespersen et al 79.

Jespersen, P., Madsen, M., Riis, H.: NEATS, New Extended Attribute Translation System. Aarhus University, 1979

Knuth 68.

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

Koster 71a.

Koster, C.H.A.: Affix grammars. In: ALGOL 68 Implementation (J.E. Peck, ed.), pp. 95-109. Amsterdam: North-Holland 1971

Koster 71b.

Koster, C.H.A.: A compiler compiler. Mathematisch Centrum, Amsterdam, Report MR127 (November 1971). Also: Using the CDL compiler compiler. In: Compiler Construction, an Advanced Course (F.L. Bauer, J. Eickel, eds.), pp. 366-426. Lecture Notes in Computer Science, Vol. 21. Berlin-Heidelberg-New York: Springer 1974

Lecarme & Bochmann 74.

Lecarme, O., Bochmann, G.V.: A (truly) usable and portable compiler writing system. In: Proc. IFIP 74 Congress, pp. 218-221. Amsterdam: North-Holland 1974

Lewis et al 74.

Lewis, P.M., Rosenkrantz, D.J., Stearns, R.E.: Attributed translations. J. Computer and System Sciences 9, 279-307 (1974)

Lorho 75.

Lorho, B.: Semantic attributes processing in the system DELTA. In: Methods of algorithmic language implementation (C.H.A. Koster, ed.), pp. 21-40. Lecture Notes in Computer Science, Vol. 47. Berlin-

Heidelberg-New York: Springer 1977

Madsen 80.

Madsen, O.L.: On defining semantics by means of extended attribute grammars. Aarhus University, Report DAIMI PB-109, January 1980

Watt 74a.

Watt, D.A.: Analysis-oriented two-level grammars. University of Glasgow, Ph.D. thesis, January 1974

Watt 74b.

Watt, D.A.: LR parsing of affix grammars. Computing Science Department, University of Glasgow, Report 7, August 1974

Watt 77.

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

Watt 79.

Watt, D.A.: An extended attribute grammar for Pascal. SIGPLAN Notices 14, 2, 60-74 (1979)

Watt & Madsen 79.

Watt, D.A., Madsen, O.L.: Extended attribute grammars. Aarhus University, Report DAIMI PB-105, November 1979

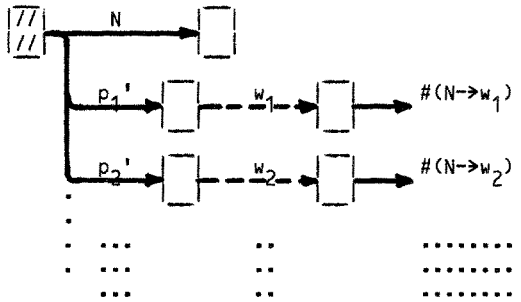

Figure 1(a). Effect of inherited rule splitting in an LR parser.
(Refer to Definition 4.1.)
The multi-predicate state is shaded.

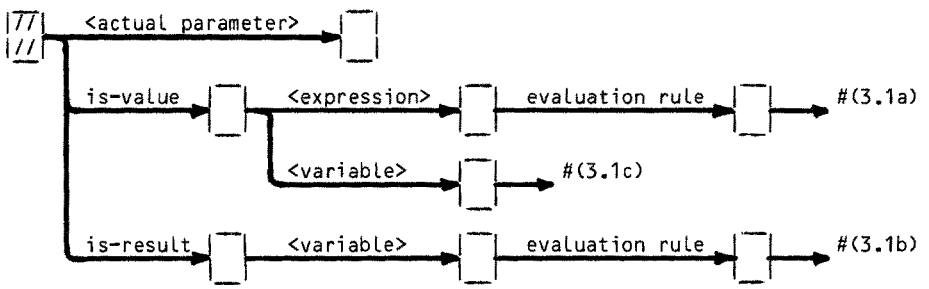

Figure 1(b). Inherited rule splitting: Example 1.

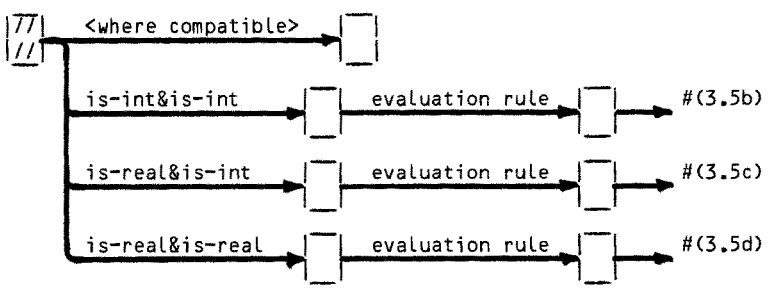

Figure 1(c). Inherited rule splitting: Example 5.

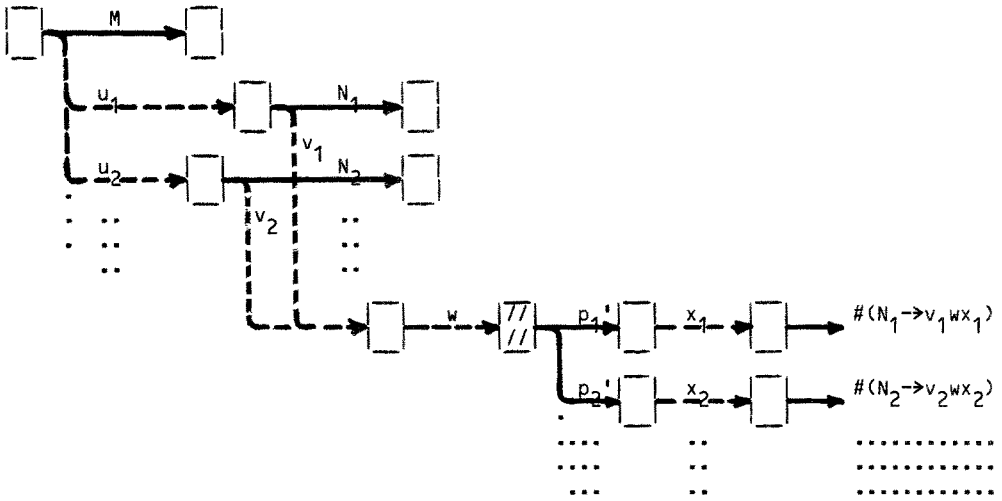

Figure 2(a). Effect of synthesized rule splitting in an LR parser.
(Refer to Definition 4.2.)
The multi-predicate state is shaded.

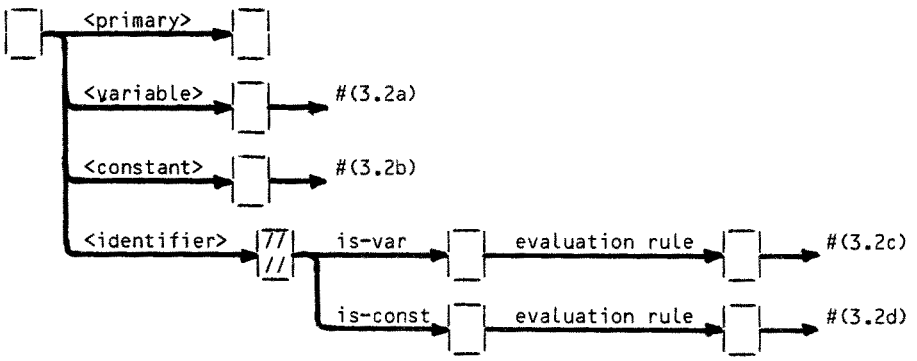

Figure 2(b). Synthesized rule splitting: Example 3.

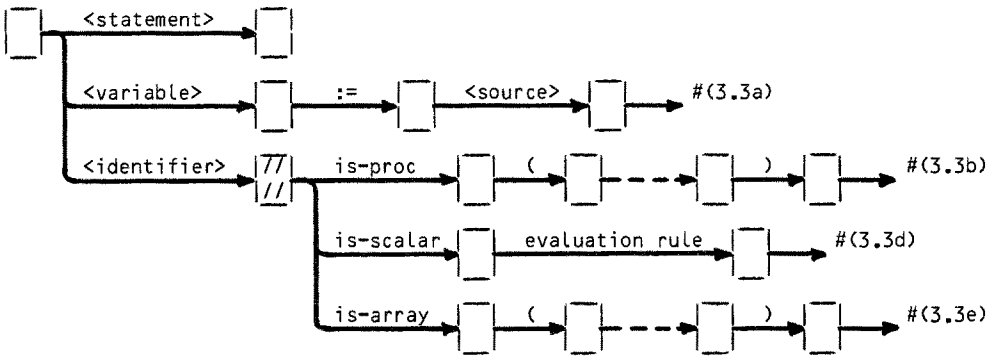

Figure 2(c). Synthesized rule splitting: Example 3.
(Some details omitted for space reasons.)

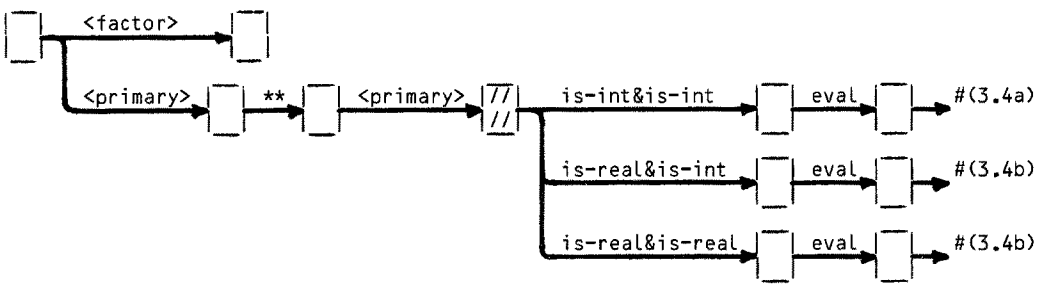

Figure 2(d). Synthesized rule splitting: Example 4.