Attribute Evaluation and Parsing

Rieks op den Akker
Department of Computer Science, University of Twente
P.O. Box 217, NL-7500 AE Enschede, the Netherlands
e-mail: infrieks@cs.utwente.nl

Bořivoj Melichar
Department of Computers, Czech Technical University
Karlovo náměstí 13, 121 35 Prague, Czechoslovakia
e-mail: TEPMB@csearn.bitnet

Jorma Tarhio
Department of Computer Science, University of Helsinki
Teollisuuskatu 23, SF-00510 Helsinki, Finland
e-mail: tarhio@cs.Helsinki.fi

Abstract

Attribute evaluation during top-down, bottom-up and left-corner parsing strategies is considered. For each of these parsing strategies, restrictions are given for L-attributed grammars that allow deterministic non-backtracking, one-pass evaluators to be used.

1. Introduction

Methods for attribute evaluation in conjunction with parsing make it possible to model a one-pass compiler as an attribute grammar. Such a model inherits the advantages of one-pass compilation: namely speed, simplicity, and small space requirements. Attribute values can be used to solve parsing conflicts and assist error recovery. Evaluation during parsing can also be applied to relieve the work of a general (multi-visit) evaluation method so that a part of the attributes are evaluated in conjunction with parsing.

When evaluating attributes during parsing, the evaluation strategy depends on the parsing method used, because the order in which the productions are recognized determines the visit sequence of the nodes in an (imaginary) derivation tree. During *LL* parsing, a full top-down (depth-first, left-to-right) walk through a derivation tree is performed. During *LL* parsing, therefore, it is possible to evaluate all attributes for every attribute grammar that is *LL* parsable and L-attributed.

LR parsing only allows a bottom-up traversal, in which every node is visited once. In general, there is insufficient information about the upper part of the tree during bottom-up parsing, to compute the attribute values of a node further down in the tree, at the time this node is (imaginary) constructed. Therefore, more restrictions on an L-attributed grammar are necessary, in order to make one-pass evaluation during bottom-up parsing possible.

In *LC* parsing some parts of the derivation tree are recognized in a bottom-up way, and others are predicted like in top-down parsing. As a consequence, the evaluation of attributes during *LC* parsing is based on some combination of methods used for evaluation during *LL* parsing and *LR* parsing.

Basic notions and notations used in this paper are presented in section 2. In section 3 we present attribute evaluation during top-down LL parsing, suitable for L-attributed $LL(1)$ grammars. In section 4, we consider attribute evaluation during bottom-up LR parsing and define the class of MLR-attributed grammars. In section 5 we present left-corner parsing, and show how for a restricted class of L-attributed left-corner grammars, all attributes can be evaluated during left-corner parsing. In section 6 we offer some conclusions. Section 7 mentions some translator writing systems that generate processors which employ attribute evaluation during parsing. The bibliography at the end gives an overview of the literature in this area.

2. Basic concepts and notations

This section presents some basic concepts and notations concerning context-free grammars and attribute grammars.

We denote a *context-free grammar* (CFG) by a four-tuple $G=(N,\Sigma,P,Z)$. The sets N of *nonterminals* and Σ of *terminals* form the *vocabulary* $V=N\cup\Sigma$. Elements of V are called *grammar symbols* and they are denoted by roman capitals towards the end of the alphabet. The letters A, B, ... denote elements of N. The letters a, b, ... denote elements of Σ, and u, v and w denote elements of Σ^*. Greek letters $\alpha,\beta,\cdots$ are used to denote the elements of V^*, the set of strings over V. The symbol ε denotes the empty string. $P \subseteq N\times V^*$ is the set of productions. A *production* $p\in P$ is written $X\rightarrow\alpha$, where $X\in N$ is called the *left-hand side* of p, and $\alpha\in V^*$ is called the *right-hand side* of p. The symbol $Z\in N$ is the *start symbol* which has only one production and which does not appear on the right-hand side of any production. We assume a CFG G is *reduced*, i.e. V does not contain useless symbols, and P does not contain useless productions.

The derivation relation $\Rightarrow$ is defined as follows. For any $\alpha,\beta\in V^*$, $\alpha\Rightarrow\beta$ if $\alpha=\gamma_1 A\gamma_2$, $\beta=\gamma_1\gamma_0\gamma_2$ and $A\rightarrow\gamma_0\in P$ where $A\in N$ and $\gamma_0,\gamma_1,\gamma_2\in V^*$. If $\gamma_2\in\Sigma^*$ we write $\alpha\Rightarrow_{rm}\beta$. If $\alpha\Rightarrow_{rm}^*\beta$, we say that β is obtained by a rightmost derivation from α ($\Rightarrow^*$ denotes the reflexive and transitive closure of the relation $\Rightarrow$). Strings in V^* obtained by a rightmost derivation from the start symbol Z are called *right sentential forms*. A sequence $p_1,p_2,\ldots,p_k$ of productions is called a *right parse* of β to α in the grammar G, if β is obtained by a rightmost derivation from α by applying the productions in the reverse order. The set of terminal strings derived from the start symbol Z is denoted by $L(G)$.
A nonterminal A is *left-recursive* if $A\Rightarrow_{rm}^{+}A\,\alpha$ for some $\alpha\in V^*$. If for some $A\in N$ there is a derivation $A\Rightarrow A_1\alpha_1\Rightarrow\cdots\Rightarrow A_n\alpha_n\cdots\alpha_1, (n\geq 1)$ with $A_n=A$, then the productions $A_i\rightarrow A_{i+1}\alpha_{i+1}$ are called *left-recursive* productions of the grammar.

We define the set $First_k(\gamma)$ for $\gamma\in V^*$, as follows. $First_k(\gamma)=\{\,x\in\Sigma^*\mid\gamma\Rightarrow^* x\alpha$ and $|x|=k$, or $\gamma\Rightarrow^* x$ and $|x|<k\,\}$, where the length of a string $\alpha\in V^*$ is denoted by $|\alpha|$.

Our definition of attribute grammars is based on [Knu68] and [Fil83].
An attribute grammar (AG) G over a semantic domain D is a CFG G_0, the underlying CFG of the AG, augmented with attributes and semantic rules. A semantic domain D is a pair (Ω,Φ), where Ω is a set of sets, the sets of attribute values, and Φ is a collection of mappings of the form $f: V_1\times V_2\times\ldots\times V_m\rightarrow V_0$, where $m\geq 0$ and $V_i\in\Omega$, $0\leq i\leq m$.
The set of *attribute symbols* is denoted by A and partitioned into I_A (*inherited* attribute symbols) and S_A (*synthesized* attribute symbols). For each attribute symbol $b\in A$, a set $V(b)\in\Omega$ contains all possible values of the attributes corresponding to b. There is a fixed set of attribute symbols associated with every grammar symbol. For $X\in V$, $A(X)$ denotes the set of attribute symbols of X. An *attribute* is denoted $X.a$, where $X\in V$ and $a\in A(X)$.

$I_A(B)$ ($S_A(B)$) denotes the set of inherited (synthesized) attributes of B. We assume that no inherited attribute symbols are associated with terminals and the start symbol. Attribute sets of each grammar symbol are linearly ordered with the inherited attributes preceding the synthesized attributes.

A production $p: X_0 \to X_1 X_2 \cdots X_n$ has an *attribute occurrence* $k.b$, $0 \le k \le n$, if $X_k.b$ is an attribute. An attribute occurrence $k.b$ of p is called an *input occurrence*, if either $b \in I_A$ and $k = 0$, or $b \in S_A$ and $k > 0$. Otherwise $k.b$ is said to be an *output occurrence*. For each output occurrence $k.b$ of p, there is exactly one *semantic rule* $k.b := f(j_1.a_1, ..., j_m.a_m)$, where every $j_i.a_i$ is an *input occurrence* of p and f is a function of the form $f: V_1 \times V_2 \times ... \times V_m \to V_0$ in Φ such that $V_0 = V(b)$ and $V_i = V(a_i)$ for $1 \le i \le m$. Notice that attribute grammars are in Bochmann normal form.

An attribute grammar is *L-attributed* [LRS74], if for every semantic rule $k.b := f(j_1.a_1, ..., j_m.a_m)$, $b \in I_A$, $j_i < k$ for each $i = 1, ..., m$.

3. Attribute evaluation during LL parsing

In this section we present the construction of a non-backtracking one-pass evaluator for the class LL-AG of L-attributed grammars that have an underlying CFG that is $LL(1)$. First, we select a kind of *LL* parser suitable for attribute evaluation. The classical $LL(1)$ parser (cf. [AhU72]) uses a parsing table and saves in the parsing stack suffixes of left sentential forms of the left derivation being constructed. This parser is not suitable for attribute evaluation, because it is necessary to use a separate stack for storing values of attributes. The reason is that it is not possible to store values of attributes in the parsing stack, because attributes evaluated during parsing are attributes of symbols which are already popped from the parsing stack. Thus, the use of the classical *LL* parser as a basis of attribute evaluation leads to the use of two stacks, which may cause implementation problems.

Another implementation technique for L-attributed $LL(1)$ grammars is based on a recursive descent parser [ASU], consisting of a set of (recursive) procedures, one for each nonterminal symbol of the grammar. It is easy to augment this parser in such a way that it can also perform evaluation of attributes during parsing. The inherited attributes of nonterminal A correspond with input parameters of the procedure for A, the synthesized attributes correspond with output parameters. The values of the attributes are now kept on the run-time parsing stack. Hence, a single stack is sufficient for storing both the syntactical symbols and the attribute values. In order to show the similarities and the differences with the methods for attribute evaluation during *LR* parsing and during *LC* parsing, which are discussed in the next two sections, we show the stack implementation of the recursive descent method. With each symbol stored in the parsing stack, the parser needs information about the rule to which it belongs. The parsing stack will contain *state symbols* that have the form $([A \to \alpha \bullet \beta], \rho)$. The first component, $[A \to \alpha \bullet \beta]$, of a state symbol is a *state item*, if $A \to \alpha\beta$ is a production of the CFG. If a state symbol with this state item is on top of the parsing stack, then the part α before the position marker $\bullet$ has already been expanded, and the part β is the predicted part of the corresponding production. The second component ρ of the state symbol is a pair (ρ_i, ρ_s), where $_i$ is (a pointer to) a sequence of values of inherited attribute instances, and ρ_s is (a pointer to) a sequence of values of synthesized attribute instances. These sequences contain

- The values of inherited attribute instances of the nonterminal occurrence X following the dot in the corresponding state item of the form $[A \to \alpha \bullet X \beta]$.

- The values of synthesized attribute instances of all symbols in part α of the corresponding state item $[A \to \alpha \bullet \beta]$.

Notice that we assume a fixed order of attributes of symbols that occur in the state items.

The LL-evaluator for an LL-AG

```
type tstate =
        record
         st   : {state item};
         inh  : {types of inherited attributes};
         synt : {types of synthesised attributes};
        end;

var S : tstate;

begin
configuration := (([Z' → .Z ],ε;ε), a_1 ··· a_n $);
repeat
    { The configuration is (S_0 S_1 S_2 ··· S_m ;a_j a_{j+1} ··· a_n $) }
    action := ME_G (S_m .st ,a_j );

    case action of
       Shift :
          { S_m .st=[X → α.a_j β] }
          S .st := g (S_m .st);
          S .synt := (synt-attr-from-S_m .synt; synt-attr-of-a_j );
          configuration := (S_0 S_1 S_2 ··· S_{m-1} S ;a_{j+1} ··· a_n $)
       Expand by A → γ :
          { S_m .st = [X → α.A β] }
          Evaluate inherited attributes of A in rule X → αA β;
          S_m .inh := inh-attr-of-A ;
          S .st := [A → .γ];
          configuration := (S_0 S_1 S_2 ··· S_m S ;a_j ··· a_n $)
       Reduce by A → γ :
          { S_m .st = [A → γ. ] and S_{m-1} .st = [X → α.A β] }
          Evaluate synthesized attributes of A in rule A → γ ;
          S_{m-1}.st := g (S_{m-1}.st);
          { S_{m-1}.st = [X → αA .β] }
          S_{m-1}.synt := (synt-attr-from-S_{m-1}.synt; synt-attr-of-A );
          configuration := (S_0 S_1 S_2 ··· S_{m-1} ;a_j ··· a_n $)
       Error:
          error := True
       Accept:
          accept := True
    end

until accept or error

end LL -evaluator.
```

Fig. 1 The LL-evaluator for an LL-AG

The evaluator can perform the following three actions.

1. *Shift*. This action takes place if a terminal symbol follows the dot in the top state item. The state symbol is replaced by a new state symbol, which, informally, means that the dot in the old top state symbol is shifted to the position immediately following the terminal symbol. The values of synthesized attribute instances of the shifted terminal symbol are stored in the new top state symbol.
2. *Expand*. This action takes place if the state item of the top state symbol of the stack has the form $[A \rightarrow \alpha.X\beta]$ and X is a nonterminal symbol. An X-production used for expansion of X is selected using a parse table. The construction of this table will be given later. The values of inherited attribute instances of the occurrence of X in the production $A \rightarrow \alpha X\beta$ are computed, and stored in the top item of the stack. Then, a new state symbol, that corresponds to the production rule selected for expansion, is pushed on the stack.
3. *Reduce* by a particular rule. This action is performed if the state item of the state symbol on top of the stack has the form $[A \rightarrow \gamma.]$, i.e. the dot is at the end position. The synthesized attributes of the nonterminal symbol A are evaluated. The top state symbol is popped off the stack. The new top state symbol will have a state item of the form $[B \rightarrow \alpha.A\beta]$, i.e. the symbol immediately following the dot equals the left-hand side symbol of the production used in the reduction. Then, informally, the dot is shifted in the top state item to the position immediately following the nonterminal symbol A. Synthesized attributes, that have just been evaluated, are stored in this new top state symbol.

During parsing, the action to be performed by the parser is uniquely determined by the state item of the top state symbol, and the look-ahead symbol. The construction of the parsing table ME_G for a given AG G is based on the usual construction of the $LL(1)$ parsing table M_{G_0}, for the underlying CFG G_0 of G. Instead of a pair (A, t), consisting of a nonterminal symbol A and a terminal symbol t, the table ME_G has entries for each pair consisting of a state item I and a terminal symbol t.

For the construction of the parser for a given $LL(1)$ grammar $G=(N,T,P,Z)$, we augment this CFG, and obtain the grammar $G'=(N',T,P',Z')$, where $N'=N\cup\{Z'\}$, and $P'=P\cup\{Z'\rightarrow Z\}$.

The parsing table ME_G is defined as follows.

1. $ME_G(I,t)=$*Shift* if and only if I has the form $[A \rightarrow \alpha.t\beta]$, where t is a terminal symbol.
2. $ME_G(I,t)=$*Expand* $B \rightarrow \gamma$ if and only if I has the form $[A \rightarrow \alpha.B\beta]$ and $M_{G_0}(B,t)=$*Expand* $B \rightarrow \gamma$.
3. $ME_G(I,t)=$*Reduce* $A \rightarrow \gamma$, if I has the form $[A \rightarrow \gamma.]$.
4. $ME_G(I,\$)=$*Accept* if I has the form $[Z' \rightarrow Z.]$.
5. All other entries of the table are error entries.

Notice that the values of attributes that are stored in the sequence ρ do not influence the parsing action. In our notation for state symbols, the sequence of attribute instance values, the pointer ρ of this state symbol points at if this symbol resides in the parsing stack, is included. The part of this sequence that contains the values of instances of inherited attributes of the symbol following the dot, is followed by a semicolon, and the values of the synthesized attribute instances of the expanded prefix of the right-hand side follow the semicolon. If we use a

Syntactic rule	Semantic rules
$Z \to A\ B$	$A.i := 1$ $B.i := A.s + 1$ $Z.s := B.s + 1$
$A \to a\ A\ b$	$A.i := A.i + a.s$ $A.s := A.s + b.s$
$A \to \varepsilon$	$A.s := A.i + 1$
$B \to c\ B\ d$	$B.i := B.i + c.s$ $B.s := B.s + d.s$
$B \to \varepsilon$	$B.s := B.i + 1$

Fig. 2 The rules for Example 1

phrase like "the item in the stack", we actually mean "the state item of the state symbol in the stack", and by "the state in the stack" we mean the state symbol in the stack including its associated pointer. Hence, the parsing stack now simply contains states.

The *LL-evaluator* shown in Figure 1 is an implementation of the parsing and evaluation method described. A *configuration* of this *LL*-evaluator, when parsing a sentence $a_1 \cdots a_n$, is a pair (STC, w), in which *STC* is the *stack contents* of this configuration, i.e. a sequence of states, and w is that part of the input $a_1 \cdots a_n$ that has not been consumed by the parser. The *initial configuration* of the parser with input w is $(([Z' \to .Z],\varepsilon;\varepsilon), w)$, where ε denotes the empty sequence of attribute values. The function g used by the *LL*-evaluator, when it performs a shift or reduce action, gives for each state S with state item of the form $[A \to \alpha.X\beta]$ the state with state item of the form $[A \to \alpha X.\beta]$.

Example 1 Let G be the LL-AG with production rules and semantic rules, as shown in the table in Figure 2. The underlying CFG of G is augmented with production 0: $Z' \to Z$. Nonterminal symbols A and B have inherited attribute i and synthesized attribute s. Nonterminal symbol Z, and the terminal symbols a, b, c, and d all have a synthesized attribute, denoted s. The shift and expand entries of the parsing table ME_G for this example, are shown in Figure 3. □

	a	b	c	d	$
Z	$Z \to .AB$		$Z \to .AB$		$Z \to .AB$
A	$A \to .aAb$	$A \to .$	$A \to .$		$A \to .$
B			$B \to .cBd$	$B \to .$	$B \to .$
a	shift				
b		shift			
c			shift		
d				shift	

Fig. 3 The (partial) parsing table for Example 1

	Stack contents	Rest of input
1	$([Z' \to .Z],\varepsilon;\varepsilon)$,	$a[1]b[2]c[3]d[4]\$$
2	$([Z' \to .Z],\varepsilon;\varepsilon)$ $([Z \to .AB],\varepsilon;\varepsilon)$,	$a[1]b[2]c[3]d[4]\$$
3	$([Z' \to .Z],\varepsilon;\varepsilon)$ $([Z \to .AB],A.i=1;\varepsilon)$ $([A \to .aAb],\varepsilon;\varepsilon)$,	$a[1]b[2]c[3]d[4]\$$
4	$([Z' \to .Z],\varepsilon;\varepsilon)$ $([Z \to .AB],A.i=1;\varepsilon)$ $([A \to a\,.Ab],\varepsilon;a.s=1)$,	$b[2]c[3]d[4]\$$
5	$([Z' \to .Z],\varepsilon;\varepsilon)$ $([Z \to .AB],A.i=1;\varepsilon)$ $([A \to a\,.Ab],A.i=2;a.s=1)$ $([A \to .],\varepsilon;\varepsilon)$,	$b[2]c[3]d[4]\$$
6	$([Z' \to .Z],\varepsilon;\varepsilon)$ $([Z \to .AB],A.i=1;\varepsilon)$ $([A \to aA\,.b],A.i=2;a.s=1,A.s=3)$,	$b[2]c[3]d[4]\$$
7	$([Z' \to .Z],\varepsilon;\varepsilon)$ $([Z \to .AB],A.i=1;\varepsilon)$ $([A \to aAb\,.],\varepsilon;a.s=1,A.s=3,b.s=2)$,	$c[3]d[4]\$$
8	$([Z' \to .Z],\varepsilon;\varepsilon)$ $([Z \to A\,.B],\varepsilon;A.s=5)$,	$c[3]d[4]\$$
9	$([Z' \to .Z],\varepsilon;\varepsilon)$ $([Z \to A\,.B],B.i=6;A.s=5)$ $([B \to .cBd],\varepsilon;\varepsilon)$,	$c[3]d[4]\$$
10	$([Z' \to .Z],\varepsilon;\varepsilon)$ $([Z \to A\,.B],B.i=6;A.s=5)$ $([B \to c\,.Bd],\varepsilon;c.s=3)$,	$d[4]\$$
11	$([Z' \to .Z],\varepsilon;\varepsilon)$ $([Z \to A\,.B],B.i=6;A.s=5)$ $([B \to c\,.Bd],B.i=9;c.s=3)$ $([B \to .],\varepsilon;\varepsilon)$,	$d[4]\$$
12	$([Z' \to .Z],\varepsilon;\varepsilon)$ $([Z \to A\,.B],B.i=6;A.s=5)$ $([B \to cB\,.d],\varepsilon;c.s=3,B.s=10)$,	$d[4]\$$
13	$([Z' \to .Z],\varepsilon;\varepsilon)$ $([Z \to A\,.B],B.i=6;A.s=5)$ $([B \to cBd\,.],\varepsilon;c.s=3,B.s=10,d.s=4)$,	$\$$
14	$([Z' \to .Z],\varepsilon;\varepsilon)$ $([Z \to AB\,.],\varepsilon;A.s=5,B.s=14)$,	$\$$
15	$([Z' \to Z\,.],\varepsilon;Z.s=15)$,	$\$$

Fig. 4 The moves of the LL-evaluator for Example 1

Notice that this table is indexed by grammar symbols instead of state items. We can use this condensed table form, because for all $t \in \Sigma$, the value $ME_G([A \to \alpha . X \beta], t)$ is completely determined by X and t, i.e. this value does not depend on A, α or β. This is true for all $LL(1)$ grammars. The $LL(1)$ attribute evaluator performs the sequence of moves for the input string: $a[1]\,b[2]\,c[3]\,d[4]$ as shown in Figure 4.

In this paper we do not study the use of attribute values during parsing. If attribute values are used to solve $LL(1)$ parsing conflicts, the underlying CFG of an L-attributed grammar need not be $LL(1)$ for making a deterministic top-down parser. Notice that a parsing conflict of an $LL(1)$ parser always means that an occurrence of a nonterminal can be expanded by more than one production. Let us call an AG $ALL(1)$ if all $LL(1)$ parsing conflicts can be solved by using the values of the inherited attributes of the nonterminal that has to be expanded. It is shown in [Mil77] that all recursively enumerable languages can be generated by such an $ALL(1)$ grammar. This implies that it is undecidable whether a sentence is generated by an $ALL(1)$ grammar. However, if we restrict the semantic attribute domains to finite ones, then $ALL(1)$ grammars generate exactly the class of deterministic context-free languages, i.e. the class of languages generated by $LR(1)$ grammars! (cf. [Akk88]) This implies that $LR(1)$ parsing can be simulated by deterministic non-backtracking attributed $LL(1)$ parsing, in which attributes are used to solve $LL(1)$ parsing conflicts.

4. Attribute evaluation during LR parsing

The *LR* parsing method is the best known non-backtracking parsing method. *LR* parsers can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written.

Because evaluation of attributes of an L-attributed grammar is very natural in conjunction with *LL* parsing, there has been a widely adopted misunderstanding that it is possible to evaluate more grammars during *LL* parsing than *LR* parsing. However, *LL* parsing can be easily emulated in *LR* parsing [Bro74]. If we insert a different marking nonterminal (generating only an empty string) in front of the right-hand side of every production, these nonterminals are recognized in the same order, during *LR* parsing, as the corresponding productions are applied in *LL* parsing. Because this transformation does not introduce *LL* parsing conflicts, the transformed grammar is still *LL* and thus also *LR*. Hence, it is possible to evaluate in conjunction with *LR* parsing, every attribute grammar that can be evaluated during *LL* parsing. And because *LL* grammars are syntactically a genuine subclass of *LR* grammars, we are able to evaluate more grammars during *LR* parsing.

We shall study a method for attribute evaluation during *LR* parsing presented by Jones and Madsen [JoM80], and revised by Sassa et al. [SIN85]. Other methods have been proposed by Watt [Wat77], Pohlman [Poh83], Melichar [Mel86], and Tarhio [Tar90]. We refer to [AMT90] for more information.

LR parsing is a form of shift-reduce parsing. In shift-reduce parsing a parse tree for an input string is constructed beginning from the leaves and working upwards to the root. We use the following model for a shift/reduce parser. The parser has an *input buffer*, a *parsing stack* and a *parsing table*. In its primitive form the parser pushes symbols of the grammar onto the stack. The main actions are the shift of a terminal symbol that is read from the input to the top of the stack, and reduction, in which a top most substring γ from the stack is replaced by a nonterminal A. This reduce action can only take place if there is a production $A \rightarrow \gamma$ in the grammar. An *LR parser* is a special shift/reduce parser. It is an algorithm that produces for an input string its right parse to the start symbol, or reports an error if the string is not in $L(G)$. An *LR* parser scans the input from left to right without any backtracking. For $LR(k)$ grammars, the decision whether to reduce or to shift a terminal symbol from the input, is uniquely determined by the stack contents and the leading k symbols of the rest of the input string (the look-ahead string). The information for making this decision is given by the LR parsing table. It is well known from the theory of LR parsing that the necessary information from the stack contents can be obtained from a finite automaton, the LR automaton. Therefore, instead of storing grammar symbols, the LR parser stores the states of this automaton in its parsing stack. The construction of the $LR(k)$ parsing table for a given CFG is based on the $LR(k)$ automaton for that CFG.

We will consider in more detail the $LR(0)$ case. The states of the finite $LR(0)$ automaton correspond to sets of $LR(0)$ items. An *LR(0) item* of a context-free grammar $G=(N,\Sigma,P,Z)$ is $[A \to \alpha . \beta]$ where $A \to \alpha\beta$ is a production of G. The *closure* of a set of $LR(0)$ items I is a set of items $CLOSURE(I)$, defined as follows:

- Every item in I is in $CLOSURE(I)$.
- If $[A \to \alpha . B \beta]$ is in $CLOSURE(I)$ and $B \to \gamma$ is a production of G, then add the item $[B \to .\gamma]$ to $CLOSURE(I)$, if it is not already there.

The set of items $GOTO(I,X)$ for a set of items I and a grammar symbol X is $CLOSURE(BASIS(I,X))$, where $BASIS(I,X)=\{[A \to \alpha X . \beta] \mid [A \to \alpha . X \beta] \in I\}$.
Using $CLOSURE$ and $GOTO$ operations, the collection of sets of $LR(0)$ items, $\{I_0, I_1, ..., I_n\}$, is constructed starting from the initial set of $LR(0)$ items, $I_0 = CLOSURE([Z' \to .Z\ \$])$, where we assume the CFG is augmented with a production $Z' \to Z\$$ if the start symbol Z of G occurs in the right-hand side of a production, or if this symbol is the left-hand side of more than one production. We always assume the input is followed by the end marker $\$$.

The sets I_j correspond to the states S_j of the LR(0) automaton. Thus, in particular, S_0 corresponds with set I_0. The transition function δ of the $LR(0)$-automaton corresponds with $GOTO$, i.e. $\delta(S_i,X)=S_j$ if and only if $GOTO(I_i,X)=I_j$. The final state S_f of the automaton is the state $GOTO(I_0,Z)$.

Remark. If $GOTO(I,X)=GOTO(I',Y)$ then $X=Y$. Thus the grammar symbol leading to some state in the $LR(0)$ automaton is unique for that state. This implies that an LR parser that stores the states on the stack, need not store the grammar symbols on the stack too. Let X be the symbol that labels the entries to state S. We use $BASIS(S)$ to denote the set of those items in S that are in a set $BASIS(S',X)$, for some state S'. □

There are three LR parsing methods, $LR(k)$, $LALR(k)$ and $SLR(k)$, which use the same parsing algorithm but employ different parsing tables [AhU72, ASU86]. For simplicity, we now only consider $SLR(1)$, also called simple $LR(1)$. The simple $LR(1)$ parser uses a table, called the *SLR parsing table*. It is based on the $LR(0)$ automaton and, it tells the simple $LR(1)$ parser what to do when the $LR(0)$ automaton contains conflict states.

The basic actions of the simple $LR(1)$ parser, shown in Figure 5, have the following meaning.

- **Shift**. The current input symbol is read, and the state determined by the goto table is placed on top of the stack.
- **Reduce** by a production $A \to \alpha$. First, $|\alpha|$ states are popped off the stack. The goto table gives the next state symbol q according to the state symbol on the top of the stack and the nonterminal A. The state q is pushed on the stack. The production $A \to \alpha$ is delivered as output.
- **Accept**. Parsing has been completed successfully.

The simple LR(1) parser/evaluator

Input:

    A sentence $a_1 \cdots a_n \$$.

Output:

    accept = True if and only if the sentence is correct.
    If accept = True, the output contains the right-parse of the sentence.

```
begin
stack := (S_0, AI(S_0), -);
evaluate attributes in IN(S_0) and store them in field AI(S_0);
a :=read(input);
error := False;
accept := False;
repeat
    state:= top(stack);
    case action(state,a) of
        Reduce A → α:
            make AI(A) in temporary storage;
            evaluate s-attributes of A and store them in AI(A);
            pop |α| symbols from the stack;
            state:=top(stack);
            push goto(state, A) (=S) on the stack;
            evaluate attributes in IN(S) and store them in field AI(S);
            copy AI(A) from temporary storage in field AI(A);
            output A → α
        Shift to S on X :
            push S on the stack;
            get the values of s-attributes of X from the
            lexical analysis and store them in field AI(X);
            evaluate attributes in IN(S) and store them in field AI(S);
            a := read(input);
        Accept :
            accept := True;
        Error :
            error := True;
    end;
until error or accept ;
end.
```

Fig. 5 SLR parser with attribute evaluation

- **Error**. The input string does not belong to $L(G)$.

If $[A \to \alpha . B \beta]$ is an item in state S, then the inherited attributes of B are associated with state S of the LR-automaton. Formally, the set $IN(S)$ of *inherited attributes of state S* is defined as

$$IN(S) = \{ B.a \mid a \in I_A(B) \text{ for some } B \text{ such that } [A \to \alpha . B \beta] \text{ in } S \}.$$

The inherited attributes in $IN(S)$ are evaluated when a state S is pushed on the parsing stack.

If $[A \rightarrow \alpha \cdot \beta] \in S$ then every attribute of every symbol in α is considered different, also in the case that the same symbol occurs more than once in α. It follows from the construction of the LR-states that in an L-attributed grammar all attributes in $IN(S)$ can be computed by means of an expression in which only the input attributes of S occur as basic elements. During parsing, these attributes are stored in the (attributed) parsing stack.

We store values of attribute instances with state symbols. Instead of state symbol S_i we actually store in stack a triple $(S_i, AI(S_i), AI(X_i))$, where $AI(S_i)$ contains the values of attributes in $IN(S_i)$ (AI stands for attribute instances), and $AI(X_i)$ contains the values of synthesized attributes of X_i, where X_i is the unique grammar symbol of transitions to S_i. The evaluation action connected with parsing actions are shown in Fig. 5.

Example 2 Consider the AG Gt shown in Figure 6.

Syntactic rules	Semantic rules
$Z \rightarrow L$	$L.iplot := true$ $L.ipos := (0,0)$
$L \rightarrow LS$	$L_2.ipos := L_1.ipos$ $S.ipos := L_2.spos$ $L_1.spos := S.spos$ $L_2.iplot := L_1.iplot$ $S.iplot := L_2.splot$ $L_1.splot := S.splot$
$L \rightarrow S$	$S.ipos := L.ipos$ $L.spos := S.spos$ $S.iplot := L.iplot$ $L.splot := S.splot$
$S \rightarrow (L)$	$L.ipos := S.ipos$ $S.spos := S.ipos$ $L.iplot := true$ $S.splot := S.iplot$
$S \rightarrow C$	$S.spos := f_1(C.id, S.ipos)$ $S.splot := f_2(C.id, S.iplot)$

Fig. 6 The example AG Gt

Grammar Gt describes a language for simple turtle graphics. Terminal C is a command with six alternatives: *north, south, east, west, plot* and *unplot.* Attributes *ipos* (inherited) and *spos* (synthesized) convey the coordinates of the plotter head, and attributes *iplot* (inherited) and *splot* (synthesized) indicate whether the plotter head is *up* or *down.* In order to make the grammar not too big, most of the semantics of the commands are hidden in the semantic functions f_1 and f_2. A command sequence between parentheses is interpreted as follows: put plotting on, perform the command sequence, and return to the state preceding the sequence. First, we consider the construction of a parser for Gt, later we will also study evaluation of its attributes. Figure 7 shows the $LR(0)$ item sets for grammar Gt. Figure 8 shows the simple $LR(1)$ parsing table based on the $LR(0)$ automaton for this example AG. An entry $r2$ means reduce using the second production in Figure 6. An entry $s3$ means shift and push state S_3 on the stack. The entry Ac means accept, and an entry 6 means push state S_6 on the stack.

S_0:	$Z' \rightarrow .Z\,\$$ $Z \rightarrow .L$ $L \rightarrow .LS$ $L \rightarrow .S$ $S \rightarrow .(L)$ $S \rightarrow .C$	S_4:	$S \rightarrow (L\,.)$ $L \rightarrow L\,.S$ $S \rightarrow .(L)$ $S \rightarrow .C$
S_1:	$Z \rightarrow L\,.$ $L \rightarrow L\,.S$ $S \rightarrow .(L)$ $S \rightarrow .C$	S_5:	$S \rightarrow (L)\,.$
S_2:	$L \rightarrow LS\,.$	S_6:	$L \rightarrow S\,.$
S_3:	$S \rightarrow (.L)$ $L \rightarrow .LS$ $L \rightarrow .S$ $S \rightarrow .(L)$ $S \rightarrow .C$	S_7: S_8:	$S \rightarrow C\,.$ $Z' \rightarrow Z\,.\$$

Fig. 7. The LR(0) item sets for Gt

Empty entries in this table are error entries. □

State	action				goto		
	C	(	)	$\$$	Z	L	S
S_0	$s7$	$s3$			8	1	6
S_1	$s7$	$s3$		$r1$			2
S_2	$r2$	$r2$	$r2$	$r2$			
S_3	$s7$	$s3$				4	6
S_4	$s7$	$s3$	$s5$				2
S_5	$r4$	$r4$	$r4$	$r4$			
S_6	$r3$	$r3$	$r3$	$r3$			
S_7	$r5$	$r5$	$r5$	$r5$			
S_8				Ac			

Fig. 8 The SLR parsing tables for the example grammar

The problem how to refer to the right attribute occurrences in the attribute stack is not solved satisfactory by Jones and Madsen in [JoM80] and [Mad80]. The problem is solved, however, by Sassa and others (cf. [SIN85] or [SIN87]). We follow their exposition with some minor modifications. We distinguish occurrences of an attribute of a nonterminal symbol at different positions in the attribute stack. An occurrence of attribute $A.a$ in the stack is a pair $(A.a, \mathit{offset}(A.a))$ where the second element indicates the position in the attribute stack relative to the top of this stack.

Consider the *parsing configuration*

$$(S_0 X_1 \cdots X_{m-k}\, S_{m-k} \cdots X_m\, S_m;\, a_j \cdots a_n\, \$\,).$$

The *offset* of an attribute in the stack is defined as follows. If a is a synthesized attribute of X_{m-k} or if a is an inherited attribute of state S_{m-k} then $offset(a) = k$.
Let $[A \rightarrow X_{m-p} \cdots X_{m-i} \cdots X_m \cdot B\beta]$ be an item in state S_m on top of the stack. It follows from the nature of *LR* parsing that if a is an inherited attribute of A then $offset(a)$ is $p+1$. If a is a synthesized attribute of X_{m-i} then $offset(a)$ is i.

The set $INP(S)$ of *input attribute occurrences* of state S is defined as follows.

$INP(S) =$
$\{(A.a, k) \mid a \in S_A(A)$ for some A s.t. $[B \rightarrow \alpha_1 A \alpha_2 \cdot \beta]$ in S and $k = offset(A.a)\} \cup$
$\{(A.a, k) \mid a \in I_A(A)$ for some A s.t. $[A \rightarrow \alpha \cdot \beta]$ in $BASIS(S)$ and $k = offset(A.a)\}$.

We use numbers as superscripts of nonterminal symbols to distinguish occurrences of a nonterminal symbol following the dot in different items in a particular state S. (e.g. $A^1, A^2, \ldots$) In the same way we distinguish occurrences of an inherited attribute $A.a$ of these occurrences of A by $A^1.a, A^2.a, \ldots$.
We define $F_S(A^t.a)$, the set of *semantic expressions* for the occurrence $A^t.a$ of $A.a \in IN(S)$. It is defined in terms of attribute occurrences in $INP(S)$.

For each state S and for each $A.a \in IN(S)$, let $E_S(A.a)$ denote the set of semantic expressions of $A.a$.

$$E_S(A.a) = \bigcup_{1 \leq t \leq p} F_S(A^t.a),$$

where p is the number of items in state S in which A occurs at the position following the dot. $F_S(A^t.a)$ is defined for all occurrences of inherited attributes in S, simultaneously, as follows:

$F_S(A^t.a)$ is the smallest set such that:

- if $[B \rightarrow \alpha \cdot A^t \beta] \in BASIS(S)$ and the semantic rule for $A.a$ associated with production $B \rightarrow \alpha A \beta$ is $A.a \leftarrow expr(a_1, \ldots, a_n)$, and $\underline{a}_i = (a_i, k)$ with k is the *offset* of the occurrence of a_i in this item, then $expr(\underline{a}_1, \ldots, \underline{a}_n) \in F_S(A^t.a)$.
- if $[B \rightarrow \cdot A^t \beta] \in S$ is an item directly derived from item $[C \rightarrow \alpha \cdot B^v \gamma] \in S$, and the semantic rule for $A.a$ associated with production $B \rightarrow A\beta$ is $A.a \leftarrow expr(a_1, \ldots, a_n)$, then $expr(e_1, \ldots, e_n) \in F_S(A^t.a)$, for all $e_i \in F_S(B^v.a_i)$ $(1 \leq i \leq n)$.

Definition 3 An attribute grammar G is *MLR-attributed*, if

- G is L-attributed.
- The underlying CFG of G is simple $LR(1)$.
- For all states S of the $LR(0)$ automaton of G, for all attributes a in $IN(S)$, the set $E_S(a)$ of semantic expressions of a contains one element. □

If an attribute grammar is MLR-attributed, the attribute a in $IN(S)$ of a state S can be evaluated when this state is pushed on the parsing stack using the semantic expression in $E_S(a)$. Synthesized attributes are computed during reduce and shift actions of the parser.

Example 4 This continues Example 2. The sets E_S of semantic expressions of the inherited attributes of the states of the $LR(0)$ automaton are shown in Figure 9. Grammar *Gt* is clearly MLR-attributed, because every set of expressions E_S contains only one expression. In Fig. 9b the history of parsing configurations is given for input *north north* (*west*) *east*. Only values of the attribute instances associated with the topmost state symbol are shown (99 denotes coordinates (9,9) and t denotes *true*). □

	$A.a$ in $IN(S)$	$E_S(A.a)$
S_0	$L.ipos$	(0,0)
	$L.iplot$	*true*
	$S.ipos$	(0,0)
	$S.iplot$	*true*
S_1	$S.ipos$	$(L.spos, 0)$
	$S.iplot$	$(L.splot, 0)$
S_2	–	
S_3	$L.ipos$	$(S.ipos, 1)$
	$L.iplot$	*true*
	$S.ipos$	$(S.ipos, 1)$
	$S.iplot$	*true*
S_4	$S.ipos$	$(L.spos, 0)$
	$S.iplot$	$(L.splot, 0)$
S_5	–	
S_6	–	
S_7	–	
S_8	–	

Fig. 9 The semantic expressions of the inherited attributes of Gt

Stack contents	**Input**
$(S_0,(00,t,00,t),-)$	*north north (west) east$*
$S_0(S_7,-,north)$	*north (west) east$*
$S_0(S_6,-,(01,t))$	*north (west) east$*
$S_0(S_1,(01,t),(01,t))$	*north (west) east$*
$S_0S_1(S_7,-,north)$	*(west) east$*
$S_0S_1(S_2,-,(02,t))$	*(west) east$*
$S_0(S_1,(02,t),(02,t))$	*(west) east$*
$S_0S_1(S_3,(02,t,02,t),-)$	*west) east$*
$S_0S_1S_3(S_7,-,west)$	*) east$*
$S_0S_1S_3(S_6,-,(-12,t))$	*) east$*
$S_0S_1S_3(S_4,(-12,t),(-12,t))$	*) east$*
$S_0S_1S_3S_4(S_5,-,-)$	*east$*
$S_0S_1(S_2,-(02,t))$	*east$*
$S_0(S_1,(02,t),(02,t))$	*east$*
$S_0S_1(S_7,-,east)$	*$*
$S_0S_1(S_2,-,(12,t))$	*$*
$S_0(S_1,(12,t),(12,t))$	*$*
$S_0(S_8,-,-)$	*$*

Fig. 9b Evaluation of input 'north north (west) east'

In our example, the semantic expressions are very simple. A general expression is of the form $f\ (x_1,...,x_n)$, where each x_i is either an input attribute occurrence or an expression. Every time a state symbol is pushed on the stack, the inherited attributes of that state are evaluated. The offsets of input attribute occurrences determine where the corresponding values can be found.

Evaluation of synthesized attributes associated with nonterminals is straightforward. Consider an attributed parsing configuration $(S_0 X_1 S_1 \cdots X_m\, S_m\, ; a_j \cdots a_k\, \$)$, where S_i represents a triple $(S_i, AI(S_i), AI(X_i))$. Suppose that a reduction by $A_0 \rightarrow A_1 \cdots A_n$ is the next parsing action. In this situation, synthesized attributes of A_0 are evaluated using the values of synthesized attributes of $A_1,...,A_n$, found in $AI(X_{m-n+1}),..., AI(X_m)$, and the values of inherited attributes of A_0, found in $AI(S_{m-n})$.

Not all L-attributed LR grammars can be evaluated during parsing. For example, if the grammar has function rule $A_2.x := f\ (A_1.x)$, associated with a left-recursive production $A \rightarrow Aa$, and f is not the identity function, no evaluation method is able to evaluate attributes during LR parsing. We finally present in this section an example of another grammar that is not MLR-attributed.

Example 5 The attribute grammar G is given by the following productions and semantic rules.

Syntactic rule	**Semantic rules**
$Z \rightarrow B\ A$	$A.x := B.s$
	$B.y := 1$
$A_0 \rightarrow C\ A_1 B$	$B.y := A_0.x$
	$A_1.x := C.s$
$A_0 \rightarrow A_1 B\ d$	$B.y := A_0.x$
	$A_1.x := A_0.x$
$C \rightarrow c$	$C.s := 2$
$A \rightarrow a$	
$B \rightarrow b$	$B.s := 1$

G is not MLR-attributed. The problem concerns the offset of the inherited attribute of A. The LR automaton for G has a state which contains items $[A \rightarrow CA\ .B\ ]$ and $[A \rightarrow A\ .Bd\ ]$. If this state is pushed on the parsing stack it is not known how far from the top we find the inherited attribute of A from which we have to copy the inherited attribute value of B. This depends on whether the B is in the right-hand side of the second or the third production. This problem can be solved by splitting the production $A \rightarrow CAB$ in two productions, $A \rightarrow CH$ and $H \rightarrow AB$ (H is a new nonterminal). The conflicting productions are then in different item sets. □

5. Attribute evaluation during left-corner parsing.

In this section we consider a one-pass attribute evaluator based on the left-corner parsing method, and we define a class of LC-attributed grammars. This class is related to the one-pass left-corner evaluator, just as the class of LL-AG is related to LL-parsing (see section 3). The left-corner of a production of the form $A \rightarrow X\alpha$ is the symbol X, the left-most symbol of the right-hand side of the production. In left-corner parsing left-corners of applied productions are recognized in a bottom-up way, where the remaining part of the right-hand side is predicted, like in a top-down parsing method. A left-corner parser for a CFG G uses an input buffer and a parsing stack. Its actions are determined by a parsing table, the left-corner parse table constructed for the CFG G. If the sentence w to be parsed is a correct sentence of G, the left-corner parser delivers the left-corner parse of w.

To define the left-corner parse of a string with respect to a given CFG, we need the following homomorphism. Let A be a set of symbols and $\Sigma \subseteq A$. The *Σ-erasing homomorphism on A*, $h_\Sigma : A^* \to A^*$ is defined by $h_\Sigma(a)=a$ if $a \notin \Sigma$ and $h_\Sigma(a)=\varepsilon$ if $a \in \Sigma$. Moreover, $h_\Sigma(\alpha +\!\!+ \beta) = h_\Sigma(\alpha) +\!\!+ h_\Sigma(\beta)$, where $+\!\!+$ denotes string concatenation, i.e. $\alpha +\!\!+ \beta$ is $\alpha\beta$. For a language L, we define $h_\Sigma(L) = \{ h_\Sigma(x) \mid x \in L \}$. Let $G = (N, \Sigma, P, Z)$ be a CFG, $|P| = m$, Δ a set $\{ p_1, ..., p_m \}$ of production labels, such that $\Sigma \cap \Delta = \emptyset$, and $\lambda_G : P \to \Delta$ a labeling function that associates with each production in P a unique symbol in Δ. We will omit the subscript G and simply write λ instead of λ_G. To a CFG G, a label set Δ, and a labeling function λ, we associate the CFG $G_{lc} = (N, \Sigma \cup \Delta, P_{lc}, Z)$, in which P_{lc} is defined as follows.

$$P_{lc} = \{ A \to p_i \mid A \to \varepsilon \in P \ , \ \lambda(A \to \varepsilon) = p_i \} \cup$$
$$\{ A \to X p_i \alpha \mid A \to X \alpha \in P \ , \ \lambda(A \to X \alpha) = p_i \}.$$

It will be clear that $h_\Delta(L(G_{lc})) = L(G)$. We use the grammar G_{lc} in order to define the left-corner parse of a string $x \in L(G)$ with respect to G.

Definition 6 Let G be a CFG, $x \in L(G)$ and Δ, and G_{lc} as defined above. The sequence of labeling symbols $\pi \in \Delta^*$ is a *left-corner parse of x with respect to G* if there is a string $y \in L(G_{lc})$, such that $h_\Sigma(y) = \pi$ and $h_\Delta(y) = x$. □

Example 7 Let G be the underlying CFG of the AG of Example 1 (see Figure 2). Figure 10 shows the productions of this grammar, and the productions of the CFG G_{lc} associated with G and the set of production labels $\Delta = \{ p_1, p_2, p_3, p_4, p_5 \}$.

1.	$Z \to A\ B$	$Z \to A\ p_1 B$
2.	$A \to a\ A\ b$	$A \to a\ p_2 A\ b$
3.	$A \to \varepsilon$	$A \to p_3$
4.	$B \to c\ B\ d$	$B \to c\ p_4 B\ d$
5.	$B \to \varepsilon$	$B \to p_5$

Fig. 10. The left-corner parse grammar of Example 7

Let $x = a\, p_2 p_3 b\, p_1 c\, p_4 p_5 d$. Clearly $x \in L(G_{lc})$, $h_\Sigma(x) = p_2 p_3 p_1 p_4 p_5$, and $h_\Delta(x) = abcd$. Hence, the left-corner parse of the sentence $abcd$ is $p_2 p_3 p_1 p_4 p_5$. □

The left-corner parser is shown in Figure 11. The parser pushes symbols on the stack that are either from V, the grammar alphabet, or items of the form $[A, X]$, where $A \in N$ and $X \in V$. The first component of such an item is the *goal symbol*, and the second component is the left-corner symbol of this item. We assume a CFG is augmented with start production $Z' \to Z\$$, and each sentence ends with $\$$. The initial stack contents is the symbol Z'. The kinds of actions the left-corner parser performs are *Left-corner found*, *Shift*, *Expand by* some production of the grammar (α^r in the Expand case in Figure 11 denotes the reversal of the string α), *Reduce*, *Accept* and *Error*.

In order to define the left-corner parsing table MLC_G for a CFG G, we need some definitions.

Definition 8 Let $G = (N, \Sigma, P, S)$ be a grammar. For each symbol $X \in V = N \cup \Sigma$, we define the *set of chains $CH(X)$ of X* (with respect to G) as follows:

- If $X \in \Sigma$ then $CH(X)=\{<X>\}$.
- If $X \in N$ then $<X> \in CH(X)$ and if $X \rightarrow \varepsilon$ in P then $<X, \varepsilon> \in CH(X)$.
- If $<\rho> \in CH(Y)$ for some $Y \in V$, and $X \rightarrow Y\gamma$ in P, then $<X, \rho> \in CH(X)$ □

Hence, a chain in $CH(X)$ is a sequence of symbols, starting with X. Moreover, Y follows Z in a chain, if there is a production $Z \rightarrow Y\alpha$ for some $\alpha \in V^*$. Elements of $CH(X)$ are called *chains of* X and denoted by σ. The last element of a chain σ is denoted by $l(\sigma)$. Notice that $CH(X)$ is an infinite set if and only if there is a derivation $X \Rightarrow_{rm}^{+} Xz$, $z \in \Sigma^*$, in G.

Let chain $\sigma=<X_0, X_1, ..., X_n> \in CH(X_0)$, with $n \geq 1$. It follows from the definition of a chain that there is a derivation

$$X_0 \Rightarrow^{p_1} X_1\gamma_1 \Rightarrow^{p_2} \cdots \Rightarrow^{p_n} X_n\gamma_n, \quad \gamma_i \in V^*, \ (1 \leq i \leq n) \qquad (*)$$

in G. The sequence of productions $p_1 p_2 \cdots p_n$ used in derivation (*) is called *a production sequence associated with the chain* σ. The production sequence associated with chains of the form $<X>$ is the empty sequence. The string γ_n is called an *r-string* of chain σ, or the *r-string* of the sequence $\pi = p_1 p_2 \cdots p_n$ of productions. Notice that a chain may have more than one r-string. The length of a sequence of productions π is denoted by $|\pi|$. The last element of a sequence π or σ is denoted by $l(\pi)$ and $l(\sigma)$, respectively.

Let $G=(N,\Sigma,P,Z)$ be a CFG. For $A \in N$ and $X \in V$, let $CH(A,X)$ be the set $\{\sigma \in CH(A) \mid l(\sigma)=X\}$, i.e. the set of chains of A that end with symbol X. Moreover, let $PS(A,X)$ be the set $\{\pi \mid \pi$ is a production sequence of a chain $\sigma \in CH(A,X)$ and $|\pi| \geq 1\}$.

Definition 9 For all $A \in N$, $X \in V$ and $u \in \Sigma$, the *partial set of production sequences* compatible with look-ahead symbol u, $PPS(A,X,u)$, is defined as follows: $\pi \in PPS(A,X,u)$, if and only if:

- $\pi \in PS(A,X)$
- There is a production $B \rightarrow \alpha A \delta$ with $\alpha \neq \varepsilon$ and $u \in First_1(\gamma\delta.Follow(B))$, where γ is an r-string of π. □

We define the left-corner parsing table for a CFG G as follows.

1. $MLC_G(A,u)$ = *Left-corner found*, if $CH(A,u) \neq \emptyset$.
2. $MLG_G(u,u)$ = *Shift.*
3. $MLC_G(A,u)$ = *Expand by* $B \rightarrow \varepsilon$, if $PPS(A,B,u) \neq \emptyset$ and $B \rightarrow \varepsilon \in P$.
4. $MLC_G([A,X],u)$ = *Expand by* p, if $p = l(\pi)$, for some $\pi \in PPS(A,X,u)$.
5. $MLC_G([A,A],u)$ = *Reduce*, if $u \in Follow(A)$.
6. $MLC_G([Z',Z],\$)$ = *Accept.*
7. All entries of the table MLC_G not defined in 1-6 are *Error* entries.

For an arbitrary CFG G the left-corner parsing table may contain multiply-defined entries. If this is indeed so, then the left-corner parser has a parsing conflict. The following conflicts are possible.

1. An expand ε-rule/expand ε-rule conflict. In this case, there are two distinct ε-rules for which the conditions in part 3 of the definition of the table are satisfied.
2. A shift/expand conflict. In this case there is a pair (A,u) such that $MLC_G(A,u)$ is defined in part 1 as a *Left-corner found*-entry, and in part 3 as an *Expand* entry.
3. An expand/expand conflict. In this case there is a pair $([A,X],u)$ such that two distinct productions can be used in the *Expand* action.

4. An expand/reduce conflict. In this case there is a pair $([A,A],u)$ such that part 4 in the definition of the table prescribes an *Expand* action and part 5 defines this entry as a *Reduce* action. This conflict can only occur if A is a left-recursive nonterminal.

Of these conflicts the first two are also $LR(1)$ conflicts. They do not occur if G is $LR(1)$. The second two conflicts are typical left-corner conflicts.

```
The left-corner parser using a left-corner parsing table
begin
config := (Z';a1 ··· an $);
accept := False;
error := False;
repeat
    { The configuration is (I0I1I2 ··· Im;aj aj+1 ··· an $) }
    action := MLC_G (Im,aj);
    case action of
        LcFound :
                { Im = A }
                config := (I0I1I2 ··· Im-1 [A,aj];aj+1 ··· an $);
        Shift :
                { Im = aj }
                config := (I0I1I2 ··· Im-1;aj+1 ··· an $);
        Expand by B → ε :
                { Im = A }
                config := (I0I1I2 ··· Im-1 [A,B]; aj ··· an $);
        Expand by B → X α :
                { Im = [A,X] }
                config := (I0I1I2 ··· Im-1 [A,B] α^r ; aj ··· an $);
        Reduce :
                { Im = [A,A] }
                config := (I0I1I2 ··· Im-1;aj ··· an $)
        Error:
                error := True
        Accept:
                { Im = [Z',Z] and m = 0 }
                accept := True
    end
until accept or error
end Left corner parser.
```

Fig. 11 The left-corner parser

A *parsing configuration* (STC,w) of the left-corner parser identifies the state of the parser at a particular moment while parsing an input sentence x. It consists of the stack contents STC, which is a sequence of stack symbols (the last symbol is the top most stack symbol), and the unread part w of the input string x. The left-corner parsing table for the grammar G of Example 7 is shown in Figure 12. The meanings of the entries in this table are as follows. *Lcf* means *Left-corner found*, s means a *Shift* action, $e3$ indicates an *Expand* action using production p_3 of the grammar, *Red* means a *Reduce* action, and *Ac* indicates that the action is *Accept*. The empty entries in the table are *Error* entries.

	a	b	c	d	$\$$
a	s				
b		s			
c			s		
d				s	
Z'	Lcf		$e3$		$e3$
A	Lcf	$e3$			$e3$
B			Lcf	$e5$	$e5$
$[Z',a]$	$e2$	$e2$			
$[Z',A]$			$e1$		$e1$
$[A,a]$	$e2$	$e2$			
$[A,A]$		Red			
$[B,c]$			$e4$	$e4$	
$[B,B]$				Red	Red
$[Z,Z']$					Ac

Fig. 12 The left-corner parsing table for Example 7

Figure 13 shows the configurations of the left-corner parser and the output, for this example grammar, for the input string *abcd$*.

	stack	**input**	**output**
1	Z'	$abcd\$$	
2	$[Z',a]$	$bcd\$$	
3	$[Z',A]\,b\,A$	$bcd\$$	p_2
4	$[Z',A]\,b\,[A,A]$	$bcd\$$	p_2p_3
5	$[Z',A]\,b$	$bcd\$$	p_2p_3
6	$[Z',A]$	$cd\$$	p_2p_3
7	$[Z',Z]\,B$	$cd\$$	$p_2p_3p_1$
8	$[Z',Z]\,[B,c]$	$d\$$	$p_2p_3p_1$
9	$[Z',Z]\,[B,B]\,d\,B$	$d\$$	$p_2p_3p_1p_4$
10	$[Z',Z]\,[B,B]\,d\,[B,B]$	$d\$$	$p_2p_3p_1p_4p_5$
11	$[Z',Z]\,[B,B]\,d$	$d\$$	$p_2p_3p_1p_4p_5$
12	$[Z',Z]\,[B,B]$	$\$$	$p_2p_3p_1p_4p_5$
13	$[Z',Z]$	$\$$	$p_2p_3p_1p_4p_5$

Fig. 13 The moves of the left-corner parser

We now present a definition of $LC(k)$ grammars. In order to clarify the difference between and the similarity to the $LR(k)$ grammars, we first present the definition of $LR(k)$ grammars.

Definition 10 A CFG G is said to be $LR(k)$ *grammar*, $k \geq 0$, if the three conditions

1. $Z \Rightarrow_{rm}^{*} \alpha A w \Rightarrow_{rm} \alpha\beta w$,

2. $Z \Rightarrow_{rm}^{*} \gamma Bx \Rightarrow_{rm} \alpha\beta y$, and
3. $First_k(w) = First_k(y)$
   imply that $\alpha Ay = \gamma Bx$.

A production $A \rightarrow \beta$ of G is said to *satisfy the LR(k) condition* if the conditions 1, 2 and 3 always imply $\alpha Ay = \gamma Bx$. □

Definition 11 A CFG G is said to be $LC(k)$, $k \geq 0$, if each ε-production satisfies the $LR(k)$-condition (see Definition 10), and if for each production $A \rightarrow X\beta$, the conditions

1. $Z \Rightarrow_{rm}^{*} \alpha Az_1 \Rightarrow_{rm} \alpha X\beta z_1 \Rightarrow_{rm}^{*} \alpha Xy_1z_1$
2. $Z \Rightarrow_{rm}^{*} \alpha' Bz_2 \Rightarrow_{rm} \alpha'\alpha'' X\gamma z_2 \Rightarrow_{rm}^{*} \alpha'\alpha'' Xy_2z_2$
3. $\alpha'\alpha'' = \alpha$ and $First_k(y_1z_1) = First_k(y_2z_2)$,

always imply that $\alpha A = \alpha' B$ and $\beta = \gamma$. □

This form of the definition of $LC(k)$ grammars is from Soisalon-Soininen and Ukkonen [SoU76]. Other characterizations of the left-corner grammars can be found in [Akk88]. It is shown in [SoS77] that $LL(k)$ grammars are $LC(k)$ and that $LC(k)$ grammars are $LR(k)$. These inclusions are proper. From a practical point it is interesting to notice that $LC(k)$ grammars may be left-recursive, although the class of $LC(k)$ languages ($k > 0$) coincides with the class of $LL(k)$ languages. For readers interested in the precise extension of the class of $LC(0)$ languages we refer to [Akk89].

Here, we will only consider $LC(1)$ parsing, i.e. only one symbol look-ahead is used. The left-corner parsing table constructed for a CFG G, does not contain multiply-defined entries if and only if G is $LC(1)$. A full proof of this statement is tedious and long, and can be found in [Akk88].

We now define the class of LC-attributed grammars and present the LC-parser/evaluator, based on the left-corner parser. Let S be a set $PPS(A, X, u)$ associated with a CFG G. We define the set of *inherited attributes of* S as follows

$IN(S) = \{ a \mid a \in I_A(B),\ B$ is left-hand side of $l(\pi)$, for some $\pi \in S \}$.

If G is an $LC(1)$ grammar, then for any two production sequences π_1 and π_2 in S, $l(\pi_1) = l(\pi_2)$. Thus, the inherited attributes in $IN(S)$ are inherited attributes of a nonterminal symbol of the grammar, namely the nonterminal symbol that is the left-hand side of $l(\pi_1)$.

Let π be a production sequence in S. If π has length one, then the left-hand side of $l(\pi)$ is A, and the inherited attributes of S are the inherited attributes of A. Let $\pi = p_1p_2 \cdots p_n$ with $n > 1$, $p_i : X_{i-1} \rightarrow X_i\gamma_i$ ($1 \leq i \leq n$), $X_0 = A$, and $X_n = X$. Let $a \in IN(S)$. This means that a is an inherited attribute of symbol X_{n-1}.

Suppose that G is the underlying CFG of an L-attributed grammar. Then the inherited attributes of symbols X_i only depend on inherited attributes of the symbol X_{i-1}. Thus, inherited attribute a depends (via a sequence of semantic functions associated with the productions that occur in π, excluding the last production) on the inherited attributes of A.

For each attribute $a \in IN(S)$, we define the set of semantic expressions $E_S(a)$ as follows.

- If π in S and $|\pi| = 1$ then a is the semantic expression associated with π, and a is an element of $E_S(a)$.
- If π in S equals $\pi' q$ where q is the production $B \rightarrow X\gamma$, and the semantic rule for $X.a$ associated with q is $X.a \leftarrow expr(a_1, \ldots, a_n)$, then $expr(e_1, \ldots, e_n)$ is an element of $E_S(a)$. This is the semantic rule associated with π, in which e_i is the semantic expression of

$B.a_i$ associated with π'. (Notice that all a_i are inherited attributes of B).

Definition 12 An attribute grammar G is *LC-attributed*, if

- G is L-attributed.
- The underlying CFG of G is $LC(1)$.
- For all partial sets of production sequences S, for all attributes a in $IN(S)$, the set $E_S(a)$ of semantic expressions of a contains one element. □

It can be shown that it is decidable whether an AG is LC-attributed (cf. [Akk88]). Notice that, if an LC-attributed grammar has a left-recursive production, then the semantic rules for the inherited attributes of the left-corner symbol of this production must be copy-rules. In case of a copy-rule $X.a \leftarrow B.b$, the semantic expression for $X.a$ is obtained from the semantic expression of $B.b$ by a simple substitution. The condition that these semantic rules are copy-rules is not sufficient for an AG to be LC-attributed. The copy-rules should also preserve an ordering of inherited attributes, because the semantic expressions *expr* $(A.x, A.y)$ and *expr* $(A.y, A.x)$ are, of course, different.

Example 13 Consider the attribute grammar given in Figure 14.

Syntactic rule	Semantic rules
$E' \to E$	$E.i := \varepsilon$; $E'.s := E.s$
$E \to E + T$	$E_2.i := E_1.i$; $T.i := E_2.s +\!\!+ p_1$ $E_1.s := T.s$
$E \to T$	$T.i := E.i$; $E.s := T.s +\!\!+ p_2$
$T \to T * F$	$T_2.i := T_1.i$; $F.i := T_2.s +\!\!+ p_3$ $T_1.s := F.s$
$T \to F$	$F.i := T.i$; $T.s := F.s +\!\!+ p_4$
$F \to (E)$	$E.i := F.i +\!\!+ p_5$; $F.s := E.s$
$F \to a$	$F.s := F.i +\!\!+ p_6$

Fig. 14 The productions and semantic rules for Example 13

If t is a derivation tree of this AG with yield w, and π is the value of the attribute s of the unique node of t that has label E', then π is the left-corner parse of w. In general, this AG defines the translation from the input sentence into its left-corner parse. It will be clear that this AG is LC-attributed. □

The LC-parser/evaluator (see Figure 15) is based on the left-corner parser of Figure 11. An attributed parsing configuration is similar to the parsing configuration but instead of the stack symbols of the form B and $[B, C]$ used by this parser, we have *attributed items* of the form

- $([A \to \alpha \bullet B \beta], inh(A), syn(\alpha), inh(B))$ (instead of stack symbols of the form B), and
- $([A \to \alpha \bullet B \beta; C], inh(A), syn(\alpha), inh(B), syn(C))$ (instead of stack symbols of the form $[B, C]$).

These attributed items are comparable to those used by the LL parser/evaluator of section 3. In the attributed configurations of the left-corner evaluator, $inh(A)$ is a sequence of values of the inherited attribute instances of A in the stack. The sequence of values of synthesized attributes of C is denoted by $syn(C)$. The notation $syn(\alpha)$, where $\alpha = X_1 \cdots X_n$, denotes the sequence that consists of the sequences $syn(X_i)$.

The syntactic actions of the evaluator are determined by its *Action* table and its associated table M, that prescribes the stack changes. These tables are derived from the left-corner parsing table MLC_G in the following way. The attribute part of an attributed item does not influence the parsing actions to be performed by the evaluator. Therefore, they are not mentioned in the definition of the tables.

1. If item X has the form $[A \to \alpha . B \beta]$, with $\alpha \neq \varepsilon$, or $A = Z'$, and $MLC_G(A, u) = $ *Left-corner found*, then $Action(X, u) = LcFound$, and $M(X, a) = [A \to \alpha . B \beta ; u]$.
2. If item X has the form $[A \to \alpha_1 . u \alpha_2]$, with $\alpha_1 \neq \varepsilon$ and $A \neq Z'$, and $MLG_G(u, u) = Shift$, then $Action(X, u) = $ *Shift-term*, and $M(X, u) = [A \to \alpha_1 u . \alpha_2]$.
3. If item X has the form $[A \to \alpha_1 . B \alpha_2]$, with $\alpha_1 \neq \varepsilon$ or $A = Z'$, and $MLC_G(B, u) = $ *Expand by* $D \to \varepsilon$, then $Action(X, u) = $ *Expand by* $D \to \varepsilon$, and $M(X, u) = [D \to .][A \to \alpha_1 . B \alpha_2 ; D]$.
4. If item X has the form $[A \to \alpha_1 . B \alpha_2 ; Y]$, and $MLC_G([B, Y], u) = $ *Expand by* $D \to Y \beta$, then $Action(X, u) = $ *Expand by* $D \to Y \beta$, and $M(X, u) = [D \to Y . \beta][A \to \alpha_1 . B \alpha_2 ; D]$.
5. If item X has the form $[A \to \alpha_1 . B \alpha_2 ; B]$, and $MLC_G([B, B], u) = Reduce$, then $Action(X, u) = $ *Shift-nont*, and $M(X, u) = [A \to \alpha_1 B . \alpha_2]$.
6. If item X has the form $[A \to \alpha .]$, then $Action(X, u) = Reduce$, and $M(X, u) = \varepsilon$.
7. If X is the item $[Z' \to Z . \$]$, then $Action(X, \$) = Accept$.
8. All entries of the tables not defined in 1-7 are *Error* entries.

If $M(X, u) = Y_1 Y_2$, then item X on top of the stack is replaced by items Y_1 and Y_2 (with Y_1 on the top and Y_2 below it). It will be clear from the definition of these tables that they do not contain multiply-defined entries if and only if the table MLC_G does not contain them.

The semantic actions of the parser/evaluator are computations of attributes of the new top most stack symbol which use only attribute values of the actual top most stack symbol.

We will now show that the LC parser/evaluator can parse $LC(1)$ grammars and evaluate all LC-attributed grammars. Suppose that in a parsing configuration, the stack symbol $[A \to \alpha . B \beta ; C]$ is on top of the parsing stack. Then the associated item fields for attribute values will contain the inherited attributes of A, the synthesized attributes of symbols in α, the inherited attributes of B (the active goal), and the synthesized attributes of C (the recognized left-corner). A symbol of this form appears on the top of the stack after a *Reduce* action with a C-production. Now, either a *Shift-nont* or an *Expand by* $D \to C \beta$ action can occur. In the case of a *Shift-nont* action, the active goal symbol B must equal the recognized left-corner symbol C. The synthesized attributes of C are then copied in the field of the semantic stack for the synthesized attributes of symbols before the dot in the top most item. The inherited attributes of the first symbol following the dot are computed from the inherited attributes of A and the synthesized attributes of α and C. In the case of an *Expand* action, the inherited attributes of D, which is the left-hand side of the recognized production, are computed from the inherited attributes of B, which is the active goal symbol. If the grammar is LC-attributed this is always possible, using for inherited attribute $D.a$ the semantic expression in $E_S(a)$ where S is the set $PPS(B, D, u)$ and u is the look-ahead symbol. Furthermore,

LC parser/evaluator for an LC-attributed grammar

```
begin
accept := False;
error := False;
stack :=([Z' → .Z $],ε,ε,ε);
a := read(input)   { input contains a2 ··· an $, and a = a1 }
repeat
      { attributed config. is ([Z' → .Z $],ε,ε,ε, ··· X ; aj ··· an $);
      case Action(X,a) of
         LcFound :
               { X=[A → α.B β]; M(X,a)=[A → α.B β;a] }
               pop; push (M(X,a));
               a := read(input);
         Expand by D → Y β :
               { X=[A → α1.B α2;Y] }
               { M(X,a)=[D → Y .β][A → α1.B α2;D] }
               pop; push (M(X,a));
               compute inh(D); copy syn(Y);
               if 1:β∈N then compute inh(1:β) end;
         Expand by D → ε :
               { X=[A → α1.B α2] }
               { M(X,a)=[D → .][A → α1.B α2;D] }
               pop; push (M(X,a)); compute inh(D);
         Shift-term :
               { X=[A → α1.a α2]; M(X,a)=[A → α1a .α2] }
               pop; push M(X,a);
               a := read(input);
               if 1:α2∈N then compute inh(1:α2) end;
         Shift-nont :
               { X=[A → α1.B α2;B] and M(X,a)=[A → α1B .α2] }
               pop; push (M(X,a)); copy syn(B);
               if 1:α2∈N then compute inh(1:α2) end;
         Reduce :
               { X=[A → α.] }
               pop; compute syn(A);
         Accept : accept := True
         Error : error := True
      end
until accept or error
end LC parser/evaluator.
```

Fig. 15 The LC parser/evaluator

the synthesized attributes of C are copied into the field for the synthesized attributes of C of the new top most stack symbol and the inherited attributes of the first symbol after C in the production that has just been recognized, are computed. After a *Reduce* action with the production $A \to \alpha$, the synthesized attributes of A are computed from the synthesized attributes of α and the inherited attributes of A. All these attributes are on the top of the semantic stack. After the pop action on the parsing stack, the new top most stack symbol has the form $[B \to \alpha . \beta ; A]$.

Just as in the LL-evaluator of section 3, a pointer can be associated with each item in the stack that refers to the list of values of attribute instances.

We know that the $LL(1)$ grammars are a proper subset of the $LC(1)$ grammars. Does this proper inclusion also hold for the corresponding classes of one-pass attribute grammars; is LL-AG a subclass of the LC-attributed grammars? If G is an $LL(1)$ grammar then for all $A, X \in N$ and $u \in \Sigma$, the set $PPS(A, X, u)$ contains at most one element. From this we may conclude that an LL-AG is indeed LC-attributed.

There are MLR-attributed grammars that are not LC-attributed, because there are simple $LR(1)$ grammars that are not $LC(1)$. The class LL-AG is not a subclass of the class of MLR-attributed grammars defined in section 4. Hence, the class of LC-attributed grammars is also incomparable with the class of MLR-attributed grammars. The problem with the offset of attributes in the stack doesn't occur for LC-attributed grammars, if we use left-corner parsing. This is obvious, because the production is recognized as soon as its left-corner symbol is recognized. The grammar shown in Example 5 (see section 4) is not MLR-attributed, but it is LC-attributed. One should notice that this does not contradict the fact that all LC-attributed grammars can be evaluated by some one-pass LR parser/evaluator. But the definition of a class of LR-attributed grammars that contains the class of LC-attributed grammars, and hence the class LL-AG, is less restrictive than the definition of MLR-attributed grammars. The reader is referred to [AMT90] for these larger classes of LR-attributed grammars.

It is possible to transform an $LC(1)$ grammar G into an $LL(1)$ grammar, say $\tau(G)$, in such a way that $L(\tau(G))$ is the same language as $L(G)$. In order to define such a transformation we need the following relation. (A_ε denotes the set $A \cup \{\varepsilon\}$.)

Definition 14 The relation $\geq_{lc}^{G}$ with respect to a CFG G is defined as follows:

- $\geq_{lc}^{G} \subseteq N \times V_\varepsilon$,
- $(X, Y) \in \geq_{lc}^{G}$ if and only if $X \to \alpha$ is a production of G and $Y = 1{:}\alpha$. □

We write $X \geq_{lc} Y$ instead of $(X, Y) \in \geq_{lc}^{G}$, and $\geq_{lc}^{+}$ denotes the transitive closure of $\geq_{lc}$.

Let $G = (N, \Sigma, P, Z)$ be a CFG and let $\bar{N}$ be the set $\{A \in N \mid A = Z$ or there is a production in P of the form $B \to \alpha A \beta$, where $\alpha \neq \varepsilon\}$. (Thus $A \in \bar{N}$ if A is the start symbol of G or A occurs in the right-hand side of a production of G of which it is not the left-corner). Let $\bar{N}$ be ordered: $\bar{N} = \{A_1, A_2, \ldots, A_n\}$. The grammar $\tau(G)$ is the CFG (N', Σ, P', Z). N' is a superset of $\bar{N}$ that contains all symbols of the form $[A, Y]$ ($A \in \bar{N}$ and $Y \in V_\varepsilon$) that appear in the productions of $\tau(G)$. The set of productions P' is defined as follows.

Initially, $P' = \varnothing$. P' will contain only those productions that are added to P' in one of the following three steps.

1. For all i, $1 \leq i \leq n$, for all $a \in \Sigma_\varepsilon$ add to P' the production $A_i \to a\,[A_i, a]$, if $A_i \in \bar{N}$ and $A_i \geq_{lc}^{+} a$.
2. For all $[A_i, Y]$, where $Y \in V_\varepsilon$, which occur in the right-hand side of a production in P', for all productions in P of the form $B \to Y\beta$, such that $A_i \geq_{lc}^{*} B$, add the production $[A_i, Y] \to \beta\,[A_i, B]$ to P', if it is not already in P'.
3. Add $[A_i, A_i] \to \varepsilon$ to P', if $A_i \in \bar{N}$.

The CFG $\tau(G)$ is reduced, and $L(G) = L(\tau(G))$.

The following example illustrates this transformation.

Example 15 Consider the CFG G given by the productions: $Z \to Z + T$, $Z \to T$, $T \to T \times id$, and $T \to id$.

Symbols id, $+$ and $\times$ are terminal symbols. The productions of the transformed grammar $\tau(G)$ are shown in Figure 16. □

$Z \rightarrow id\ [Z,id]$	$[Z,id] \rightarrow [Z,T]$
$[Z,T] \rightarrow [Z,Z]$	$[Z,T] \rightarrow \times id\ [Z,T]$
$[Z,Z] \rightarrow +T\ [Z,Z]$	$[Z,Z] \rightarrow \varepsilon$
$T \rightarrow id\ [T,id]$	$[T,id] \rightarrow [T,T]$
$[T,T] \rightarrow \varepsilon$	$[T,T] \rightarrow \times id\ [T,T]$

Fig. 16 The productions of the transformed grammar

Notice that the nonterminal symbols of $\tau(G)$ are the stack symbols used by the left-corner parser. Transformation τ has the following property. From the left-parse of each sentence $w \in L(G)$, with respect to the CFG $\tau(G)$, it is possible to obtain the left-corner parse of w with respect to the original CFG G.

Suppose that G is an LC-attributed AG. If the semantic rules for the inherited attributes of left-corner symbols of G are copy-rules, then it is easy to augment the transformation τ and obtain an attributed transformation that results in an L-attributed $LL(1)$ grammar. By an implementation of this augmented transformation, we can extend a compiler writing system for LL-AG and obtain a compiler writing system for LC-attributed grammars. The reader is invited to produce an LL-AG from the LC-attributed grammar of Example 13. For more details concerning the transformation τ, we refer to [Akk88].

6. Concluding remarks

One-pass compilation based on attribute grammars has several advantages over more general methods. One of the advantages is that it is not necessary to store the complete parse tree for evaluation of the attributes. Another advantage is that attribute values can be used to solve parsing conflicts, so that the underlying CFG of an AG does not have to be deterministically parsable by the parsing method used. We have presented three classes of attribute grammars for which attribute evaluation can be performed during parsing. We have shown that a parser/evaluator for the class of LC-attributed grammars can be defined using concepts and techniques that are inherited from the implementation of L-attributed $LL(1)$ grammars, and the implementation techniques used for the evaluation of inherited attributes during LR parsing.

7. Existing systems

We give a list of some existing translator writing systems that generate processors which employ attribute evaluation during parsing. We consider only systems with one-pass evaluation of both inherited and synthesized attributes in conjunction with parsing. This list is certainly not exhaustive. More systems are described in [DJL88].

Systems for attribute evaluation during top-down parsing include CWS2 [BoW78], MUG1 [GRW76], MIRA (LILA) [LDH83], APARSE [MKR79], SUPER [Ser82], and TCGS [Sch91]. APARSE was the first system where the values of attributes were used to influence parsing. TCGS, the Twente Compiler Generator System, is a compiler writing system for L-attributed $LL(1)$ grammars in Extended BNF-notation. It produces a scanner and a recursive descent parser/evaluator.

MUG1 [GRW76], Rie [SIS90], Poco [Eul85], Metauncle [Tar89] and Haripriyan's system [HSS88] are translator writing systems for attribute evaluation during *LR* parsing. Rie is a system that can produce parser/evaluators for the class of *LR*-attributed grammars defined in [SIN85]. Metauncle, developed in the HLP project at the University of Helsinki, generates evaluators for uncle-attributed grammars [Tar90]. Haripriyan's system implements Pohlmann's evaluation method [Poh83]. The system SABLE from Twente University generates an *LALR* (1) parser/evaluator that can use attribute values to solve parse conflicts. The input grammar can be syntactically ambiguous [Vel88].

There appear to be very few compiler writing systems that generate a parser/evaluator based on left-corner parsing. Programmar is a system that generates a backtracking parser/evaluator for affix grammars, based on left-corner parsing [Mei86].

Acknowledgements
We are grateful to Albert Nijmeijer for critically reading a draft of this paper.

Bibliography

The following list contains papers and books that cover attribute evaluation and parsing. They are either referred to in this paper, or they are included because they are not mentioned in the bibliography [DJL88].

[ASU86] Aho, A.V., Sethi, R. and Ullman, J.D. *Compilers, Principles, Techniques and Tools.* Addison-Wesley Publ., Reading, Mass., 1985.

[AhU72] Aho, A.V. and Ullman, J.D. *The Theory of Parsing, Translation and Compiling.* Vol.1 and Vol.2, Prentice Hall, Englewood Cliffs, N.J., 1972.

[Akk88] op den Akker, R. Parsing attribute grammars. Doctoral dissertation, Dept. Comput. Sci., University of Twente, The Netherlands, 1988.

[Akk89] op den Akker, R. On LC(0) grammars and languages. *Theoretical Comput. Sci.* **66** (1989) 65-85.

[AMT90] op den Akker, R., Melichar, B. and Tarhio J. The hierarchy of LR-attributed grammars. In: Proc. of International Conference WAGA on Attribute Grammars and their Applications (ed. P. Deransart and M. Jourdan), Lect. Notes Comput. Sci. 461, Springer-Verlag, Berlin, 1990, 13-28.

[Bea82] Beatty, J.C. On the relationship between the LL(1) and LR(1) grammars, *Journal of the ACM* **29** (1982) 1007-1022.

[BoW78] Bochman, G. and Ward, P. Compiler writing system for attribute grammars. The Computer Journal 21, 2 (1978), 144-148.

[Bro74] Brosgol, B.M. Deterministic translation grammars. Ph.D. Thesis, TR-74, Harvard University, Cambridge, Mass., 1974.

[DJL88] Deransart, P., Jourdan, M. and Lorho, B. *Attribute Grammars - Definitions, Systems and Bibliography.* Lectu. Notes Comput. Sci. 323, Springer-Verlag, Berlin, 1988.

[Eul85] Eulenstein, M. POCO - Compiler generator user manual. Techn. Bericht A2/85, Universitat des Saarlandes, 1985.

[Fil83] Filé, G. The theory of attribute grammars. Doctoral dissertation, Twente University of Technology, Enschede, The Netherlands, 1983.

[GRW76] Ganzinger, H., Ripken, K. and Wilhelm, R. MUG1 - an incremental compiler-compiler. In: ACM 1976 Annual Conference, 415-418.

[Hal87] Hall, M. The optimization of automatically generated compilers. Ph.D. thesis, Dept. Comput. Sci., University of Colorado, 1987.

[HSS88] Haripriyan, H., Srikant, Y. and Shankar, P. A compiler writing system based on affix grammars. Comp. Lang. 13 (1988), 1-11.

[JoM80] Jones, N.D. and Madsen, C.M. Attribute-influenced LR parsing. In: Proc. of the Aarhus Workshop on Semantics-Directed Compiler Generation, N.D. Jones (ed.) Springer-Verlag,Berlin, 1980, 393-407.

[Knu68] Knuth, D.E. Semantics of context-free languages, *Mathematical Systems Theory* **2** (1968) 127-145. Correction in: *Mathematical Systems Theory* **5** (1971) p.95.

[LDH83] Lewi, J.,De Vlaeminck, K., Huens, J. and Steegmans, E. The language implementation laboratory LILA: an overview. In: Microcomputers; developments in industry, business and education (eds. D. Wilson and C. van Spronsen), Elsevier, 1983, 11-21.

[LRS74] Lewis, P.M., Rosenkrantz, D.J. and Stearns, R.E. Attributed translations, *J. Comput. System Sci.* **9** (1974) 279-307.

[Mad80] Madsen, C.M. Parsing attribute grammars. M. Sc. thesis. Dept. Comput. Sci., University of Aarhus, Aarhus, 1980.

[Mei86] Meijer, H. Programmar: a translator generator. Doctoral dissertation, University of Nijmegen, The Netherlands, 1986.

[Mel86] Melichar, B. Attributed translation directed by LR parser and its implementation. In: Proc. of the Bautzen Workshop on Compiler Compilers and Incremental Compilation, Informatik Reporte 1989: 12, Akademie der Wissenschaften der DDR, GDR, 1986, 273-290.

[Mil77] Milton, D. Syntactic specification and analysis with attributed grammars. Technical report #304, Comput. Sci. Dept., University of Wisconsin-Maddison, Wisconsin, 1977.

[MKR79] Milton, D., Kirchoff, L. and Rowland, B. An ALL(1) compiler generator. In: ACM SIGPLAN '79, Symposium on Compiler Construction, SIGPLAN Notices 17, 10 (1979), 152-157.

[NaS86] Nakata, I and Sassa, M. L-attributed LL(1)-grammars are LR(1)-attributed, *Information Processing Letters* **23** (1986) 325-328.

[Poh83] Pohlmann, W. LR parsing of affix grammars, *Acta Informatica* **20** (1983) 283-300.

[Row77] Rowland, B.R. Combining parsing and evaluation for attribute grammars. Technical Report #308, Comput. Sci. Dept., University of Wisconsin-Madison, Madison, Wisconsin, 1977.

[Sas88] Sassa, M. Incremental attribute evaluation and parsing based on ECLR-attributed grammars. Technical Report ISE-TR-88-66, Institute of Information Sciences and Electronics, University of Tsukuba, 1988.

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

[SIN87] Sassa, M., Ishizuka, H. and Nakata, I. ECLR-attributed grammars: a practical class of LR-attributed grammars, *Information Processing Letters* **24** (1987) 31-41.

[SIS90] Sassa, M., Ishizuka, H., Sawatani, M. and Nakata, I. Rie - introduction and user's manual. Technical Report ISE- TR-90-82, Institute of Information Sciences and Electronics, University of Tsukuba, 1990.

[Sch91] Schepers, W.J.M. Twente Compiler Generator System, user's guide, Memoranda Informatica 91-aa, Dept. Comput. Sci., University of Twente, The Netherlands, 1991.

[Ser82] Serebryakov, V. Principal features of the input language and implementation of the translator design system SUPER. Prog. and Comput. Softw. 8, 1 (1982), 52-56.

[SoS77] Soisalon-Soininen, E. Characterization of LL(k) languages by restricted LR(k) grammars, Report A-1977-3. Dept. Comput. Sci., University of Helsinki, Finland, 1977.

[SoU76] Soisalon-Soininen, E. and Ukkonen, E. A characterization of LL(k) languages. In: *Automata, Languages and Programming*, S. Michaelson and R. Milner (eds.), Edinburgh University

Press, Edinburgh, 1976, 20-30.

[StT89] B. Stiefel and P. Thiel: Application of attributed grammar for syntax and attribute-directed bottom-up translation. In: Proc. of the Second Workshop on Compiler Compiler and High Speed Compilation (ed. D. Hammer), Informatik Reporte 1989: 3, Akademie der Wissenschaften der DDR, GDR, 1989, 222-238.

[Tar88] Tarhio, J. Attribute grammars for one-pass compilation. Report A-1988-11, Ph.D Thesis, Dept. Comput. Sci., University of Helsinki, Finland, 1988.

[Tar88b] J. Tarhio: The compiler writing system Metauncle. Report C-1988-23, Dept. Comput. Sci., University of Helsinki, Finland, 1988.

[Tar89] Tarhio, J. A compiler generator for attribute evaluation during LR parsing. In: Proc. of the Second Workshop on Compiler Compiler and High Speed Compilation (ed. D. Hammer), Lect. Notes Comput. Sci. 371, Springer-Verlag, Berlin, 1989, 146-159.

[Tar90] Tarhio, J. Uncle-attributed grammars. BIT **30** (1990) 437-449.

[Vel88] Veldhuijzen van Zanten, G.E. SABLE: a parser generator for ambiguous grammars. Memorandum Informatica 88-62, Dept. Comput. Sci., University of Twente, The Netherlands, 1988.

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