Generalized LR parsing and attribute evaluation

Paul Oude Luttighuis and Klaas Sikkel

Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE, Enschede, the Netherlands
email: {oudelutt|sikkel}@cs.utwente.nl

Abstract

This paper presents a thorough discussion of generalized *LR* parsing with simultaneous attribute evaluation. Nondeterministic parsers and combined parser/evaluators are presented for the *LL*(0), *LR*(0), and *SKLR*(0) strategies. *SKLR*(0) parsing occurs as an intermediate strategy between the first two. Particularly in the context of simultaneous attribute evaluation, generalized *SKLR*(0) parsing is a sensible alternative for generalized *LR*(0) parsing.

1 Introduction

Natural language theory and programming language theory have a common foundation in formal language theory. In particular, context-free grammars find broad application in both fields. Yet, whereas programming language theory allows for restriction of the class of applicable grammars in order to obtain more efficient parsers, natural language theory typicaly demands parsing algorithms for general context-free grammars.

Semantic issues are generally handled by different formalisms in both theories. While attribute grammars (AGs) are widely used in programming language circles, natural language semantics are dealt with by e.g. feature-structure grammars. A well-known problem in programming language theory is the evaluation of the attributes of an AG *during* parsing. Attribute evaluation during parsing allows for refraining from storing the parse tree and can therefore be space efficient. Moreover, attribute values, which have been computed during parsing, may be used to control the parser, for instance by (partly) resolving parsing conflicts.

This paper reports on such attribute evaluation during parsing. The parsing algorithm used, however, is a typical natural language parsing algorithm: generalized LR parsing, also called Tomita's algorithm. By combining Tomita's algorithm with simultaneous attribute evaluation, programming language theory may benefit from the virtues of Tomita's algorithm, while natural language theory may benefit from the efficient techniques for simultaneous attribute evaluation, found in programming language theory.

Tomita's algorithm provides a clever deterministic implementation of the nondeterminism occurring in parsing non-*LR* grammars in an *LR* fashion. Attribute evaluation during deterministic *LR* parsing is a non-trivial problem and has raised considerable interest in literature on programming language implementation. On the other hand, attribute evaluation during deterministic *LL* parsing is simple, provided we restrict ourselves to the use of L-attributed AGs only, which is what we will do in the entire paper. L-attributed AGs are such that dependencies between attribute values do not flow but from left to right in the parse tree.

Because of this observation, we structured our paper as follows. We start with discussing nondeterministic *LL* parsing. By means of an intermediate step (yielding a technique called *SKLR* parsing), this is transformed into *LR* parsing. Then, the nondeterministic *LL* parser is enhanced with simultaneous attribute evaluation and the same transformation steps are used to obtain attribute evaluation during nondeterministic *LR* parsing. The most severe difficulties occur in the first transformation step.

The following principles characterize our dis-

cussion.

- Generality. No determinism is required beforehand.
- Finiteness. Except for the attribute domains, the data and control structures of our algorithms must be finite.
- Static evaluation. There must be no need at all to check at evaluation time whether an attribute instance has been evaluated yet. An attribute instance must be evaluated at the moment the parser enters the state with which the instance is associated.

Within the bounds of these principles, we push the combination of attribute evaluation and parsing to its very limits. Efficiency is not our first concern. We refrain from using dynamic evaluation techniques, by which the evaluation of attribute values can be postponed. Efficiency considerations may afterwards be used to obtain efficient implementations.

2 Context-free grammars and attribute grammars

We assume that the reader is familiar with context-free grammars (CFGs). The CFGs in this paper always have a production of the form $S \rightarrow \#X\$$ such that S does not occur in any other production. *Left-recursive* CFGs have a nonterminal A and a string α of grammar symbols for which $A \Rightarrow_G^+ A\alpha$. *Hidden-left-recursive* CFGs have a nonterminal A and a strings α, β of grammar symbols for which $A \Rightarrow_G^+ \alpha A \beta$ and $\alpha \Rightarrow_G^+ \varepsilon$.

We will not present a formal definition of attribute grammars (AGs). An AG is based on a CFG. Every grammar symbol carries a series of *attributes* of a certain type. In a parse tree, therefore, *instances* of these attributes occur. The attribute instances in one production in the parse tree are functionally dependent on one another. In the AG, this dependence is specified by a set of *semantic rules*, associated with that production. The function of an *attribute evaluator* is to assign values to the attribute instances according to these semantic functions.

We distinguish two kinds of attributes: *synthesized* and *inherited* ones. Synthesized attributes of a symbol depend on other attributes in the production *below* that symbol, whereas inherited attributes depend on attributes in the production *above* it.

In order to simplify the discussion, we use a special form of AGs, caled *untyped L-attributed AGs*, or *ULAGs*. In fact, an ULAG is an AG,

- which is L-attributed. This means that inherited attribute occurrences of right-hand side symbols cannot depend on synthesized attribute occurrences of right-hand symbols to their right.
- in which all attributes have the same type;
- in which every symbol has exactly one inherited and exactly one synthesized attribute.
- in which any attribute occurrence depends on *all* used attribute occurrences to its left in that production.

The first restriction is the most severe of all. The other three are easily dealt with by simple rewrites of the AG.

3 $LL(0)$ and $LR(0)$ parsers

This section present a nondeterministic $LL(0)$ parser and transform it, via an $SKLR(0)$ parser into an $LR(0)$ parser.

3.1 L-parsers

This subsection gives a short description of the parser model, called *L-parser*, used in this paper, without presenting a formal definition.

The core of an L-parser is a transition relation between *instantaneous descriptions*. Such an instantaneous description consists of a stack contents, which is a string of states, and a (remaining) input, which is a string of grammar symbols. Operation starts with a special stack contents, consisting of a special state, the *initial state* and the input string. This instantaneous description may be turned into other ones by means of the transition relation. Operation stops if the top of the stack is another special state, the *final state*.

As opposed to more common parser models, an L-parser is allowed to pre-append symbols to the remaining input during reduction steps. Our reduction steps are such that they pre-append the lhs of the associated production to the input, instead of shifting it immediately. This shift is performed by a compulsory subsequent shift step. Advantages are that

- It simplifies the description of parsers.
- It facilitates the definition of attribute evaluation during parsing.
- It enables parsing of arbitrary sentential forms, instead of just terminal strings.
- It nicely generalizes to parsers for context-sensitive grammars.

3.2 Dotted rules

LR parsing was introduced by Knuth (1965). Although *LL* and *LR* parsing can be elegantly described as being each other's duals (Sippu and Soisalon-Soininen, 1990), they can both be seen as (implicitly) performing a depth-first left-to-right walk over the (virtual) parse tree.

Positions in this walk are indicated by *dotted rules*. In essence, a dotted rule is a production together with some position in its rhs. This position is rather *between* symbols that *at* them. A dotted rule $[A \to X_1 \ldots X_i \bullet X_{i+1} \ldots X_n]$, where $n \geq 0$ and $0 \leq i \leq n$, indicates that the parser's current position during the tree walk is in an application of $A \to X_1 \ldots X_n$ at the position between X_i and X_{i+1}. Dotted rules of the form $[A \to \bullet\alpha]$ are called *starting* dotted rules, others are called *proper* dotted rules.

During a tree walk, three kinds of steps occur: production steps, shift steps, and reduction steps. Production steps are steps down in the parse tree. They always leave from a dotted rule of the form $[A \to \alpha_1 \bullet B\alpha_2]$ and arrive at a dotted rule of the form $[B \to \bullet\beta]$. Shift steps are steps to the right, leaving from a dotted rule of the form $[A \to \alpha_1 \bullet X\alpha_2]$ and arriving at one of the form $[A \to \alpha_1 X \bullet \alpha_2]$. Finally, reduction-shift steps are steps up in the parse tree and they leave from a dotted rule of the form $[B \to \beta\bullet]$ to arrive at one of the form $[A \to \alpha_1 B \bullet \alpha_1]$. See Figure 1.

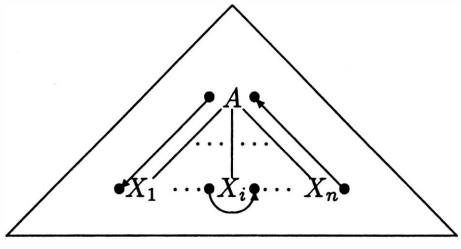

Figure 1: Production, shift, and reduction-shift steps in a parse tree.

3.3 $LL(0)$ parsers

Dotted rules are the states of our $LL(0)$ parsers. The initial state is $[S \to \# \bullet X\$]$, the final one $[S \to \#X\$\bullet]$. At any moment, the $LL(0)$ parser may perform

- a production step. In this case, the current state must be of the form $[A \to \alpha_1 \bullet B\alpha_2]$. Any dotted rule of the form $[B \to \bullet\beta]$, can be the next current state, and hence be pushed onto the stack. The remaining input is not changed;
- a shift step, in which case the current state must be of the form $[A \to \alpha_1 \bullet X\alpha_2]$ and the remaining input must start with an X. The new current state is $[A \to \alpha_1 X \bullet \alpha_2]$ and is pushed onto the stack. The X leading the remaining input is removed;

- a reduction-shift step. In this case, the current state must be of the form $[A \to X_1 \ldots X_n \bullet]$. Now, by the nature of the algorithm, the topmost $n+2$ elements of the stack[1] are

$$[B \to \beta_1 \bullet A\beta_2][A \to \bullet X_1 \ldots X_n] \ldots [A \to X_1 \ldots X_n \bullet]$$

The topmost $n+1$ of these are popped. A is pre-appended to the input. Finally, a shift step is performed.

Nondeterminism only occurs in two cases.

- If the current state is of the form $[A \to \alpha_1 \bullet B\alpha_2]$ and there is more than one rule with B at its lhs, different production steps are possible. This is called a *produce-produce conflict*.
- If the current state is of the form $[A \to \alpha_1 \bullet B\alpha_2]$, both a shift step and at least one production step is possible, provided the remaining input is headed by B. This is called a *produce-shift conflict*.

3.4 $SKLR(0)$ parsers

For most CFGs the $LL(0)$ parser is very nondeterministic. As soon as a nonterminal occurs as the lhs of more than one production, nondeterminism

Uniting states is a technique, taken from LR parsing, for increasing determinism. The transformation from LL to LR parsing can be seen to consist of two subsequent steps, which we will call *production step elimination* and *determinization*.

A glance at the $LL(0)$ parser shows that nondeterminism occurs at production steps. The first transformation step, production step elimination, consists in eliminating explicit production steps by uniting those dotted rules into one state that are interrelated by such a step.

Hence, in *singleton-kernel* $LR(0)$ *parsing*, or $SKLR(0)$ parsing, states are sets of dotted rules. Every state contains one proper dotted rule, its *kernel*. The other dotted rules in a state are those that can be obtained from the kernel by means of one or more production steps.

For an $SKLR(0)$ parser, being in some state implies that the parser is, in the virtual parse tree, *simultaneously* at all positions that are indicated by a sequence of dotted rules in this state, that starts with the kernel and of which every following dotted rule can be reached from the preceding one by means of a production step.

Let us discuss some differences with $LL(0)$ parsers.

- First of all, there are no production steps here. This is the very essence of our first transformation step. A sequence of consecutive production steps is made at once upon entering a new current state.
- Therefore, states changed from being dotted rules to sets of dotted rules.
- A reduction step involves popping n elements from the stack, rather than $n+1$, because a production step does not cause a push any more.

Nondeterminism occurs in an $SKLR(0)$ parser in the following cases.

- If some state contains two different dotted rules of the form $[A \to \alpha_1 \bullet X\alpha_2]$ and $[B \to \beta_1 \bullet X\beta_2]$, a shift step leaves the next current state not uniquely determined. This is a *shift-shift conflict*,
- If some state contains a dotted rule of the form $[A \to \alpha \bullet X\beta]$ and a dotted rule of the form $[A \to \alpha\bullet]$, so that both a shift and a reduction step are possible (a *shift-reduce conflict*),
- If some state contains more than one dotted rule of the form $[A \to \alpha\bullet]$, so that more than one different reduction step is possible (a *reduce-reduce conflict*).

As opposed to the $LL(0)$ parser, the $SKLR(0)$ parser applies to at least some left-recursive grammars. Though nondeterministic, $SKLR(0)$ parsers are terminating for a reasonable class of left-recursive CFGs. An advantage of $SKLR(0)$ parsers over $LR(0)$ parsers is that the parsing tables are more space efficient, in the worst case. Any $SKLR(0)$ parser will have a number of states that equals the sum of the lengths of the rhss. However, $SKLR(0)$ parsers are more nondeterministic than $LR(0)$ parsers.

[1] When writing down stack contents as a string, the top will be at this string's right.

3.5 $LR(0)$ parsers

As mentioned, three kinds of nondeterminism occur in $SKLR(0)$ parsers. The second transformation step, determinization, completely abolishes the shift-shift conflicts. Wherever such a conflict may occur, all possible next current states are taken together to form a new state. So, states in $LR(0)$ parsers will be unions of $SKLR(0)$ parser states. Unfortunately, shift-reduce and reduce-reduce conflicts remain, and generally even grow in number.

Hence, in $LR(0)$ parsers,

- states are unions of $SKLR(0)$ states, and
- there are no shift-shift conflicts.

Nondeterminism occurs in $LR(0)$ parsers only as shift-reduce or reduce-reduce conflicts.

4 Extending parsers with attribute evaluation

In this section, we add attribute evaluation to the $LL(0)$ parser and present transformations of this attributed parser, via an attributed $SKLR(0)$ parser to an attributed $LR(0)$ parser. It appears that production step elimination causes problems related to attribute evaluation. Remember that the discussion is restricted to ULAGs.

4.1 L-parser/evaluators

This subsection presents L-parser/evaluators. They form the basis for the parser/evaluators of this section.

When attribute evaluation is performed during parsing and the parse tree should not be stored, entirely nor partially, the attribute values should be calculated at the moment that the parser, during its walk through the virtual parse tree, is at the corresponding node. The attribute values are kept within the states on the stack.

First of all, we must capture the fact that the lexical analyzer yields, instead of a string of alphabet symbols, a string of pairs, each pair consisting of a symbol and a synthesized attribute value associated with the symbol. This also enables us to pre-append the lhs of a production, after a reduction step, to the remaining input, together with its synthesized attribute value.

Also, we have to be careful with the definition of the states of a parser/evaluator. In some way, the attribute values must be attached to the states. However, at parser/evaluator-generation time, attribute values are not known yet. They are calculated at run time. Therefore, we make a distinction between *static* and *dynamic* states. Dynamic states are those entities that occur on the stack during an actual run of the parser/evaluator. Static states underlie dynamic states. Static states are constructed at parser/evaluator-generation time. Obviously, the only part of a dynamic state that cannot be calculated at parser/evaluator-generation time concerns the actual attribute values.

4.2 $LL(0)$ parser/evaluators

This subsection introduces the $LL(0)$ parser/evaluator.

Within states of the form $[A \to \alpha_1 \bullet X\alpha_2]$, the value of the inherited attribute occurrence of X is kept. Within states of the form $[A \to \alpha_1 X \bullet \alpha_2]$, the value of the synthesized attribute occurrence of X is kept. Notice that in most states, being those that have at least one symbol (say Y) following and one symbol (say X) preceding the dot, carry a value of the inherited attribute occurrence of Y as well as the value of the synthesized one of X. Yet, the synthesized attribute occurrence of X is evaluated earlier than the inherited one of Y.

Now, let us consider the three kinds of steps of the $LL(0)$ parser.

- Consider the production steps. A new state of the form $[B \to \bullet\beta]$ is pushed. If $\beta \neq \varepsilon$, the inherited attribute value of the first symbol of β, say X, must be calculated. By the L-attributedness of the AG, it can only depend on the inherited attribute occurrence of B, of which the value can be found in the state $[A \to \alpha_1 \bullet B\alpha_2]$, below the current one on the stack.
- Consider the shift steps. A new state of the form $[A \to X_1 \ldots X_n X \bullet \alpha]$ is pushed. We have two cases.
  - If $\alpha \neq \varepsilon$, this state must contain the synthesized attribute value of X and the inherited attribute value of

the first symbol of α, say Y. The synthesized attribute value of X is copied from the remaining input. It has been put there by the lexical analyzer or by the preceding reduction action. The inherited attribute value of Y must be calculated. Because of the L-attributedness of the AG, it only depends on the synthesized attribute occurrences of $X_1, \ldots, X_n, X$ and on the inherited attribute occurrence of A. The synthesized attribute value of X is taken from the stack. The others can be found at a distance from the stack's top that equals the position number of their corresponding symbol in the string $X_n \ldots X_1 A$.

- If $\alpha = \varepsilon$, it suffices to copy the synthesized attribute value from the remaining input.

- Consider a reduction step with $A \rightarrow X_1 \ldots X_n$ as the production involved. No new states are pushed. However, A is pre-appended to the remaining input. It must be accompanied by its synthesized attribute value. This value is calculated by using the values of the synthesized attribute occurrences of $X_1, \ldots, X_n$ and the inherited attribute of A. These can be found at a distance from the stack's top that equals the position number of their corresponding symbol in the string $X_n \ldots X_1 A$. Also, a reduction step is only made if the semantic condition associated with the production evaluates to TRUE.

A dynamic state must contain an inherited and a synthesized attribute value. The function needed to calculate the inherited attribute value from another one is determined by the dotted rule itself. This is not the case for the synthesized attribute since its corresponding function is determined by the production applied beneath the dotted rule's production. Now, suppose we have the dotted rule $[A \rightarrow X_1 \ldots X_m \bullet Y_1 \ldots Y_n]$ as the state's core. If $m > 0$, the synthesized attribute value of X_m is simply copied into the synthesized attribute 'slot' of the state. If $n > 0$, we must calculate the inherited attribute value of Y_1, which depends on $m + 1$ other values, being the inherited attribute value of A and the m synthesized ones of $X_1, \ldots, X_m$. If $n = 0$, we do not have to calculate an inherited attribute value.

A static state in our $LL(0)$ parser/evaluator is a dotted rule, extended with the function that calculates the inherited attribute of this dotted rule. This is called an *extended dotted rule*. An extended dotted rule contains a *string* of functions[2] instead of just one. This is to prepare for future transformations. Extended dotted rules are denoted by $[A \rightarrow \alpha \bullet \beta | \varphi]$.

A static attributed $LL(0)$ state is an extended dotted rule. A dynamic attributed $LL(0)$ state is a triple consisting of the associated static state, and the associated synthesized and inherited attribute value, respectively.

An $LL(0)$ parser/evaluator is as (non)deterministic as its underlying $LL(0)$ parser.

4.3 $SKLR(0)$ parser/evaluators

This subsection will present $SKLR(0)$ parser/evaluators and the problems involved in their definition. Basically, two main techniques are used. First, we use string rewrite systems in order to capture semantic knowledge of the semantic functions. We need this knowledge in order to keep our states finite, but they also help decreasing nondeterminism. Second, the problem of how to address the proper inherited attribute value in a state is solved by including a kind of place marker in the dynamic states.

4.3.1 Production step elimination

Let us first consider which attribute values are associated with an $SKLR(0)$ state. First of all, we have the inherited and synthesized attribute value of the kernel of the state. Also, we have the inherited attributes of the other dotted rules. They have no synthesized attributes because they are starting dotted rules. So, every $SKLR(0)$ state contains exactly one synthesized attribute value, but generally more than one inherited attribute value. Hence, in order to extract an inherited attribute value from a state, one needs more arguments than just the state itself. A first guess, of course, is that the additional argument should be

[2] Concatenation is this string denotes function composition.

the dotted rule with which the intended value is associated. Unfortunately, this has two problems.

- Inherited attribute values (of a nonterminal A) are needed only when a shift or reduction-shift step is made according to a production that has A as its lhs. Therefore, at the moment of extracting the value, the only information available is the symbol immediately following the dot in the intended dotted rule (being A), and not the entire dotted rule. In general, the same symbol may immediately follow the dot in more than one dotted rule in one state.

- Generally, even the entire dotted rule is not sufficient to uniquely identify an inherited attribute value. More than one inherited attribute value may be associated with the same dotted rule in the same state.

In an elegant definition, a static attributed $SKLR(0)$ state would be a set of extended dotted rules, being the closure, with respect to some relation, of some kernel element. A first concern is whether the function, kept within the extended dotted rule, should denote the function needed to calculate the associated inherited attribute value from the one immediately preceding it (with respect to that relation), or the function needed to calculate the associated inherited attribute value from the inherited attribute value of the kernel. In this last case, we have a string of functions, which denotes composed functions, inside the extended dotted rule. We have decided for the second alternative.

Unfortunately, such a definition yields static states with infinitely many extended dotted rules in case of left-recursion. A partial solution to this problem may be obtained as follows.

In most of attribute grammar theory, the semantics of the semantic functions is not taken into account. Only attribute *dependencies* are important. Evaluator implementation is based on the a priori availability of implementations of the semantic functions. If we persist in this principle here, we cannot but forbid ULAGs with left-recursive underlying CFGs. However, being able to handle left-recursive grammars is a major advantage of LR parsing over LL parsing. It is not desirable to let the addition of simultaneous attribute evaluation nullify this. So, we are forced to pay attention to the semantics of the semantic functions.

There are many formalisms by which we might do this. Without indulging to elaborate discussions of this area of computer science, we present one solution, which enables to handle at least the most practical cases. This solution uses string rewrite systems.

4.3.2 String rewrite systems

A string rewrite system is a special kind of *rewrite system*. A (general) rewrite system contains an arbitrary set of *objects* and a binary relation $\Rightarrow$ on this set. A rewrite step transforms an object into another one according to a rewrite rule. An object that can be obtained from another object by a (possibly empty) sequence of rewrite steps, respectively one rewrite step, is called a *descendant*, respectively a *direct descendant*, of that object. An object is called *irreducible* if no rewrite rule can be applied to it any more. If an object can be rewritten into an irreducible object, this irreducible object is called a *normal form* of the original one. If no infinite sequences of rewrite steps can occur, the system is called *terminating*. The system is called *confluent* if any two different descendants of the same object have a common descendant. The system is called *complete* if it is both terminating and confluent. Complete rewrite systems implement a total function yielding the (unique) normal form for any given object.

A *string rewrite system* (or *semi-Thue system*, or *STS*, shortly) is a special kind of rewrite system. The objects are strings over some set of symbols. A rewrite rule here is a pair of strings. An application of such a rewrite rule to a string consists in substituting a substring, that matches the lhs of the rule, by the rhs of the rule. A *finite* STS has finitely many string rewrite rules.

Now, STSs are used in the following way. When constructing a static state, which consists of extended dotted rules, the function strings in these extended dotted rules are first subjected to the STS. Using STSs is only a partial solution to the problems mentioned: particular ULAGs will still have to be rejected, because they yield infinite static states even after the use of the STS.

4.3.3 High-lights

As mentioned earlier, we also need to include, in our dynamic states, an indication that allows us to extract the proper inherited attribute value from it, the next time such a value is requested. For sake of brevity, the complicated discussion of these high-lights is omitted. We restrict ourselves to mentioning that, in principle, extended dotted rules suffice as high-lights. However, more sophistication, that is, using so-called *extended symbols* or even *extended sets* as high-lights decreases the amount of nondeterminism. Also, high-lights appear to have syntactical consequences: a particular high-light in a dynamic state restricts the set of possible steps. For instance, a certain high-light may forbid a shift of some symbol a, even when the underlying static state allows such a shift step.

Unlike $LL(0)$ parser/evaluators, $SKLR(0)$ parser/evaluators may show more nondeterminism than their underlying parser. One can distinguish *syntactic conflicts* form *semantic conflicts*. Semantic conflicts may occur when high-lights are changed and the next high-light is not uniquely determined.

4.4 $LR(0)$ parser/evaluators

States in $LR(0)$ parsers are unions of $SKLR(0)$ parser states. Analogously, static attributed $LR(0)$ states are unions of static attributed $SKLR(0)$ states. This causes a technical problem, because the static states may now contain multiple kernel elements. The solution involves the introduction of so-called *selection functions*. These form the most important difference between $LR(0)$ and $SKLR(0)$ parser/evaluators. However, we will refrain from discussing this problem here.

4.5 Examples of practical string rewrite systems

In this subsection, we present some examples of STSs that are applicable to frequently appearing semantic functions in attribute grammars.

First of all, copy rules often occur. A copy rule occurs when an attribute evaluation function is the identity function. For the identity function, say I, we have a simple rule.

$$\mathrm{I} \to \varepsilon$$

In other words, the identity function can simply be removed from any string of functions.

A second class of functions suitable for rewriting is formed by the constant functions. Let $c : A \to B$ be any constant function and $f : B \to B$ any function. Then, the rule

$$cf \to c$$

can be added. It expresses that any function applied before the constant function is superfluous.

Finally, we notice that another rule can be used when f is, for instance, the logical negation, or a function, taking and yielding pairs, that exchanges the pair's first and second constituent. This rule is

$$ff \to I$$

5 Implementing parsers

In this section, we discuss implementation issues of the nondeterministic parsers as presented in Section 3. A main problem here is that the conceptual nondeterminism must be implemented on a deterministic machine. A first remark is that all parsers discussed here can be made table-driven in the usual way.

5.1 $LL(0)$ parsers

In $LL(0)$ parsers, a production step is performed as follows. Conceptually, the parser initiates a new parser for every possible new state to be pushed.

A shift step is performed as follows. Suppose the current state is $[A \to \alpha_1 \bullet X\alpha_2]$ and Y heads the remaining input. If $X = Y$, $[A \to \alpha_1 X \bullet \alpha_2]$ is pushed and Y is removed from the remaining input. If $X \neq Y$, no shift step can be performed.

A reduction step is performed as follows. If the current state is not of the form $[A \to X_1 \ldots X_n \bullet]$, no reduction step can be performed. If it is, $n+1$ elements are popped from the stack and A is preappended to the input. After that, a compulsory shift step follows.

If no production step, as well as no shift nor reduction step is possible, the parser dies. If more than one kind of step is possible, new parsers are initiated for both alternatives. In $LL(0)$ parsers

this can occur with production and shift steps (a produce-shift conflict). If a reduction-shift step is possible, no other steps are.

5.2 *SKLR*(0) parsers

In *SKLR*(0) parsers, a shift step is performed as follows. Suppose X heads the remaining input. Then, the set of all dotted rules in the current state of the form $[A \rightarrow \alpha_1 \bullet X\alpha_2]$ is determined. For every such dotted rule, a new parser is initiated, which pushes the state with $[A \rightarrow \alpha_1 X \bullet \alpha_2]$ as its kernel.

A reduction step is performed as follows. The set of all dotted rules of the form $[A \rightarrow X_1 \ldots X_n \bullet]$ in the current state is determined. For every such dotted rule, a new parser is initiated, which pops n elements from the stack.

If no shift nor reduction step is possible, the parser dies. If both shift and reduction steps are possible, new parsers are started for both.

5.3 *LR*(0) parsers

In *LR*(0) parsers, a shift step is performed as follows. Suppose X heads the remaining input. Then, the set of all dotted rules in the current state of the form $[A \rightarrow \alpha_1 \bullet X\alpha_2]$ is determined. If this set is empty, the parser dies. If not, a new state is pushed with this set as its kernel set.

Reduction steps are performed as in *SKLR*(0) parsers. If no shift nor reduction step is possible, the parser dies. If both shift and reduction steps are possible, new parsers are started for both.

5.4 Handling nondeterminism: a simple approach

Our parsers contain a possibly huge amount of nondeterminism. Because our strategy is to try out all possibilities in case of nondeterminism, we must have a way to implement this. A simple approach is the following: a parser first determines which production, shift and reduction steps are possible. Suppose there are n different possible next steps. Then the parser copies its instantaneous description n times, yielding one copy for every possible step. Then, for every copy, it performs the associated step and initiates a new parser to continue parsing with the resulting new instantaneous description. After that, it dies. If $n = 0$, the parser dies without initiating new ones.

A parser stops (successfully) when its instantaneous description has the final state q_F as its current state.

For the time being, a sequential implementation is intended. Words like 'parallelism', 'synchronization', and different 'parsers' are used in a conceptual sense.

5.5 Sharing instantaneous descriptions

There is a lot of unnecessary copying of instantaneous descriptions in the simple approach. Major portions of the copies will coincide. Sharing these coinciding parts may substantially increase efficiency.

A parser can be seen to move on a tape that is formed by the stack and the remaining input. Upon a nondeterministic choice, originally, the tape was copied several times before the step. However, major parts of the tape are the same for all copies. These parts, a bottom part (prefix) of the stack and a suffix of the remaining input can be shared by all alternatives. This yields a data structure which we will call a *graph-structured instantaneous description.*

Basically, this is a directed acyclic graph, with one source (the bottom of the stack) and one sink (the end of the input). It is maintained in such a way, that every path from the source to the sink corresponds to a single instantaneous description. On every such path, there is one current position, which marks the border between the stack and the input of the corresponding instantaneous description. Obviously, maximal sharing is reached when no parent has different but equally labeled children (at the stack side) and no child has different but equally labeled parents (at the input side). This technique will save space, and may cost time (because of the additional problems of manipulating shared data), but it may save much more (because instantaneous descriptions will not be exhaustively copied).

Though this graph-structured instantaneous description may seem symmetric, there is an asymmetry to be found in that the stack part of an instantaneous description is both growing and shrinking during a run, whereas the input part only shrinks (after initialization). Therefore, re-

maining inputs of two different parsers will always be such, that one is a suffix of another. So, at the remaining-input side, it is very easily assured that no parser destroys other parsers' data. This is different at the stack side. There, a reduction step, causing stack elements to be popped, endangers other parsers' data. In this case, the instantaneous description should be partially copied (that is, the endangered part only) before the step is actually performed. The partial copy is connected with the original at the border of the endangered part.

There is no reason, other than for efficiency, to decrease the amount of sharing used. One might even refrain from sharing data at one of both sides of the shared instantaneous description. In this case, our shared data structure has the structure of a tree. Two possibilities occur. The first is when the input side lacks all sharing. This yields a *tree-structured instantaneous description*, which has one source (the bottom of the stack) and many sinks, indicating the end of the remaining input for every separate parser. The other possibility occurs when the stack side lacks all sharing. This causes a data structure with many sources and one sink. We will call this a *funnel-structured instantaneous description*. We refrain from giving technical details.

5.6 Synchronizing the parsers

Additional efficiency may be gained by requiring parallelism to be synchronous, or, even stronger: by demanding synchronization on shift steps.

To see this, notice that the shift step is the only step in which the input is affected (because of the combined reduce-shift step). Then, if all parsers remove the same symbol from the input at the same time, they all will always have the same remaining input, which therefore can be factored out of the graph-structured instantaneous description. This technique will save space, but cost time, since parsers may have to wait for synchronization.

Because the remaining input is no longer part of the graph-structured instantaneous description, it is rather a *tree-structured stack*, which has one source and multiple sinks, one for the top of the stack of every single parser. Now, because the steps of the parser depend on the current state only, we know that all branches of this tree-structured stack, that have the same state at their leaves, will have identical tree-structured substacks originating from them in the future. Therefore, it might be advantageous to unify these leaves into one. This will save space, as well as time and/or processors, simply because identical subruns will be performed only once. However, it may also cost time, because searching for identically labeled leaves takes time. This data structure is called a *graph-structured stack*. It has one source and, at any moment, generally multiple sinks, one for every state occurring at the top of a stack at that moment.

The idea of a graph-structured stack is not new. It originated from Tomita (1985), in which, however, it is only used in the context of generalized *LR* parsing. However, graph-structured stacks were introduced differently. There is no notion of asynchronous (pseudo-)parallelism nor of graph-structured instantaneous descriptions. Moreover, the concept of a tree-structured stack is different, in that the sharing is the other way around: the tree has the current state at its root, and the stack bottom (or rather bottoms) at its leaves. We automatically arrived at the reverse notion of tree-structured stacks by not presupposing shift synchronization. In order to stay consistent with earlier terms, we use the name *funnel-structured stack* for Tomita's notion of tree-structured stacks.

5.7 Tree- and graph-structured stacks in detail

A detailed discussion of the use of tree- and graph-structured stacks in generalized $LL(0)$, $SKLR(0)$, and $LR(0)$ parsing is given in Oude Luttighuis and Sikkel (1992), but omitted here for brevity's sake.

5.8 Parallelism in parsers

Here, we will discuss ways to actually implement the defined parsers and parser/evaluators in parallel.

The parallelizations of these algorithms can be classified according to which data structure(s) is (are) distributed over different processors. By this distribution, we do not necessarily mean physical distribution, but rather conceptual distribution. Of course, parallelism may also be ob-

tained by distributing control structure, but in our algorithms, control structure is captured in a data structure.

The only data structure maintained in our parsers and parser/evaluators is the (possibly tree- or graph-structured) instantaneous description. Yet, it consists of the (remaining) input and the stack part.

Because activity only occurs at the top of the stack and the head of the input, it seems unprofitable to cut them into subsequent pieces and assign these to processors. So, the only fruitful parallelism may be found in the pseudo-parallelism already present. Different processors simply process the different alternatives originating from nondeterministic decision points.

5.8.1 Dividing multiple linear stacks and tree-structured stacks

Parallel implementation of Tomita's algorithm can be found in (Tanaka and Numazaki, 1989; Numazaki and Tanaka, 1990). Both divide the different alternatives occurring at nondeterministic decision points over the processors. The first uses no sharing whatsoever of the instantaneous description: several different copies of (linear) stacks are processed by different processors. The second one uses the tree-structured stack.

5.8.2 Dividing graph-structured stacks

Graph-structured stacks were introduced, because tree-structured stacks show many identical activities, because many top nodes may be labeled with the same state. As discussed earlier, sharing is best applied immediately after the shift steps that start a tst-step.

What we might do is take a processor for every state in the parser[3]. Its task is to perform one tst-step to the (shared) wait node that is labeled with its associated state.

Because all shift steps shift the same symbol, the maximum number of wait nodes resulting is the maximum number of states that can occur after a shift of a particular symbol. So, if we choose to assign a processor to all parser states, many of them will be inactive at some time. This can be improved by partitioning the parser states into a set of blocks. Every block corresponds to a grammar symbol X and contains those states that can become current state after a shift of X. Now, we take a number of processes such that, to each of them, (at most) one state in every block is assigned. This calls for a number of processors that equals the maximum block size.

5.8.3 Dividing the input

There exist other parallelizations of Tomita's algorithm. In (Lankhorst and Sikkel, 1991) the PBT (Parallel Bottom-up Tomita) algorithm is presented. In this algorithm, the input is divided over processors such that every processor processes one input symbol. The processors operate in a pipeline, communicating *marked symbols* from the end to the beginning of the string. A marked symbol consists of a grammar symbol (terminal or nonterminal) and two numbers, which indicate the left and right border of a substring of the input string, that is derivable from that grammar symbol. Actual recognition of a new marked symbol with left border i is done by the processor associated with the i^{th} input symbol.

It was stated in the beginning of this section that it seems unprofitable to cut the input (or the stack) into subsequent pieces and assign these to processors, because activity only occurs at the head of the input (and the top of the stack). Yet, this holds only for true parallelizations of Tomita's algorithm. PBT is rather another algorithm than Tomita's. The facts that PBT requires different table construction and different manipulation of the graph-structured stack support this view. Also, whereas Tomita's algorithm cannot handle hidden-left-recursive CFGs, PBT can.

6 Implementing parser/evaluators

This section discusses implementation issues of the nondeterministic parser/evaluators as presented in section 4. The underlying syntactic part

[3]That is, a processor is associated with every parser state, not with every instance of a state on the graph-structured stack.

can be made table-driven in the usual way. Additionally, in $SKLR(0)$ and $LR(0)$ parser/evaluators, high-lights have to be taken into account as well in the parser tables.

6.1 Calculating attribute values

Attribute values are computed upon syntactical steps. The (candidate) steps to be taken are chosen on purely syntactical grounds. In reduction steps, the semantic condition may prohibit the step. In other cases, the calculation of attribute values simply follows syntactical processing. The argument values are extracted from the other states on the stack (and copied from the remaining input, in case of a shift step).

6.2 Handling nondeterminism

In the parser/evaluators presented, attribute evaluation can influence parsing in two ways. First, a reduction step will be prevented if the semantic condition evaluates to `FALSE`. Second, the high-lights have syntactical consequences. Yet, the techniques of handling nondeterminism, as mentioned for parsers, can in principle all be used for parser/evaluators as well. However, implementing a graph-structured stack will not be as profitable for parser/evaluators as it is for parsers. This has two reasons.

- As opposed to the steps of the parsers, the steps of the parser/evaluator are not completely determined by the current state only. For the evaluation of attribute values, argument values are needed that reside in states further down in the stack. This causes complications when, somewhere within the sequence of states from which attribute values must be extracted, sharing has been applied, so that a new kind of semantic nondeterminism occurs. In other words, because more than one stack is associated with a top node in graph-structured stacks, different attribute values may be extracted from states deeper in the stack.

- States may only be shared if they are identical. Since dynamic states now include attribute values, these have to be identical as well. It is doubtful whether the time saved by sharing identical states outweighs the time lost in verifying that states are identical, given the dynamic nature of attribute values.

Both tree- as well as graph-structured stacks suffer from the fact that a larger portion of the stack is inspected for each step. This may cause more read conflicts in a parallel implementation. Still, a tree-structured stack seems feasible.

6.3 Parallel parser/evaluators

This section will discuss ways to implement the defined parser/evaluators in parallel.

Because parsers are the skeleton of our parser/evaluators, we review the opportunities for parallelism, which are present in our parsers, and discuss whether they can be applied to parser/evaluators as well.

Division of multiple linear stacks and tree- and graph-structured stacks can be carried over to parser/evaluators. However, there are two difficulties.

- Graph-structured stacks are hardly useful for parsing/evaluation (as discussed earlier). Therefore, the parallel parsing technique for dividing the graph-structured stack over a (statically bounded) number of processors is not applicable.

- In tree-structured stacks (as well in graph-structured stacks) the sharing of parts of the data structure introduces the possibility of read conflicts.

Unfortunately, dividing the input seems unsuitable as well, because the L-attributedness of ULAGs imposes a very sequential nature on attribute dependencies. Only when a left-to-right dependency is absent, possibilities for parallelism appear. This occurs, for instance, in case there are no inherited attributes, but only synthesized ones. However, if we allow ourselves to use dynamic evaluation techniques, this changes, because we may postpone evaluation. This is the subject of the next subsection.

6.4 Dynamic attribute evaluation

Although dynamic attribute evaluation is not a main topic of this paper, we will spend some remarks on it.

Dynamic attribute evaluation abandons the need to (completely) evaluate attribute values before they are used. Generally, a directed graph is maintained in which nodes are labeled with semantic functions and have outgoing arcs to all their arguments. Let us call such a graph an *attribute graph*. This graph can essentially be handled by two techniques.

- The first technique calculates an attribute value in the attribute graph only when all its argument values are completely evaluated. Let us call this the *evaluation-before-use* technique.
- In the other technique, the attribute graph is viewed as a syntactic description of a value. Rewrite rules are applied to it whenever this is possible. A rewrite rule may, in principle, be performed to any part of the graph. This is the *graph-rewriting* technique.

In both techniques, the attribute graph grows whenever the parser enters a new state and shrinks whenever a new attribute value is calculated, or a rewrite rule has been applied.

Dynamic evaluation may be helpful in parsing/evaluation in two ways. First, it allows for postponing (complete) evaluation of attribute values. Second, related with that, it allows for a less rigid approach to infinity problems in case of left recursion. Apparently, a combination of generalized parsing/evaluation techniques and dynamic evaluation may be desirable.

The fact that dynamic evaluation partly frees attribute evaluation from problems caused by syntactical processing also offers new possibilities for parallelism.

For a more elaborate discussion, see Oude Luttighuis and Sikkel (1992).

7 Related approaches

The only GLR parser generator that we know of is the incremental parser generator IPG (Heering et al., 1990; Rekers, 1992), from the ESPRIT project GIPE (generation of interactive programming environments). They chose Tomita's algorithm as a starting point for their incremental parser generation because it provides a good mix of generality and efficiency. One of the reasons for accepting arbitrary context-free languages is that the grammar is allowed to be *modular*, and none of the classes of LR-grammars is closed under composition. Rekers' thesis does not discuss attribute evaluation in the incremental parser generator.

Affix Grammars over a Finite Lattice (AGFLs) (Weijers, 1986) are a sub-class of affix grammars specifically designed for natural languages. The formalism can be located somewhere between attributed grammars and feature-structure grammars (Shieber, 1986). There are two basic ways to parse an AGFL: (1) Evaluate the affix values on-the-fly during the construction of the parse forest, or (2) compute a parse forest according to the underlying context-free grammar and decorate it with affix values afterwards. Koster (1991) claims that the first approach is more practical, provided that his "Recursive Backup" algorithm (Koster, 1975) is used, rather than an Earley or Tomita parser (despite the exponential worst-case complexity of the Recursive Backup algorithm). Nederhof and Sarbo (1993) claim the reverse, however. They discuss how ambiguity can be handled practically in an interactive environment.

A generalization of an LC parser, based on the recursive backup method mentioned above, is used for the closely related formalism of Extended Affix Grammars in (Meijer, 1986).

A generalized LR parser for a query language for logical databases is described in (Lang, 1988).

8 Conclusions

Contributions of this paper are

- a systematic treatment of generalized *LL* and *LR* parsing,
- the description of *SKLR* parsing as an intermediate form of these,
- a correspondingly systematic treatment of attribute evaluation during *LL*, *SKLR*, and *LR* parsing, with a precise identification of problematic issues in the case of *SKLR* and *LR* parsing,
- (partial) solutions to these problems, viz. high-lights, string rewrite systems, and selection functions,

- the technique of attribute evaluation during generalized *SKLR* parsing, which is to be preferred to evaluation during generalized *LR* parsing, because it avoids some technical complications while nondeterministism is only marginally increased.

Future work may be done on the following problems.

- What can be gained by adding the use of look-ahead information to *SKLR* parsing?
- What is the relationship between *SKLR* and *LC* parsing?
- Can our evaluation techniques be efficiently combined with dynamic evaluation techniques?
- Is the use of string rewrite systems a real gain in practice?

References

Heering, J. — P. Klint — J. Rekers (1990). Incremental generation of parsers. *IEEE Transactions on Software Engineering*, SE–16:1344–1351.

Knuth, D.E. (1965). On the translation of languages from left to right. *Information and Control*, 8:607–639.

Koster, C.H.A. (1975). A technique for parsing ambiguous grammars. In D. Siefkes, editor, *GI — 4. Jahrestagung.* Lecture Notes in Computer Science 26.

Koster, C.H.A. (1991). Affix grammars for natural languages. In H. Alblas and B. Melichar, editors, *Proceedings of the International Summer School on Attribute Grammars, Applications and Systems (Prague, Czechoslovakia, June 4–13, 1991)*, pages 469–484, Berlin, Germany. Springer-Verlag. Lecture Notes in Computer Science 545.

Lang, B. (1988). Datalog automata. In *Proceedings of the 3rd International Conference on Data and Knowledge Bases: Improving Usability and Responsiveness (Jerusalem, Israel, 1988)*, pages 389–404.

Lankhorst, M. — K. Sikkel (1991). PBT: A parallel bottom-up Tomita parser. Memoranda Informatica INF 91–69, Department of Computer Science, University of Twente, Enschede, The Netherlands, September.

Meijer, H. (1986). *Programmar: A Translator Generator.* PhD thesis, University of Nijmegen, Nijmegen, The Netherlands.

Nederhof, M.-J. — J.J. Sarbo (1993). Efficient decoration of parse forests. In H. Trost, editor, *Feature formalisms and linguistic ambiguities*, pages 95–109, Chicester, U.K. Ellis Horwood.

Numazaki, H. — H. Tanaka (1990). A new parallel algorithm for generalized LR parsing. In *Proceedings of the 13th International Conference on Computational Linguistics (Helsinki, Finland, 1990) (Vol. 2)*, pages 304–310.

Oude Luttighuis, P. — K. Sikkel (1992). Attribute evaluation during generalized parsing. Memoranda Informatica 92–85, Department of Computer Science, Enschede, The Netherlands.

Rekers, J. (1992). *Parser Generation for Interactive Environments.* PhD thesis, University of Amsterdam, Amsterdam, The Netherlands.

Shieber, S.M. (1986). An introduction to unification-based approaches to grammar. CSLI Lecture Notes 4, Center for the Study of Language and Information, Stanford University, Stanford, California, USA.

Sippu, S. — E. Soisalon-Soininen (1990). *Parsing Theory*, volume II LR(k) and LL(k) Parsing. Springer-Verlag, Berlin, Germany.

Tanaka, H. — H. Numazaki (1989). Generalized LR parsing based on logic programming. In *Proceedings of the International Workshop on Parsing Technologies (Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 1989)*, pages 329–328.

Tomita, M. (1985). *Efficient Parsing for Natural Language.* Kluwer Academic Publishers, Boston, Massachusetts, USA.

Weijers, G.A.H. (1986). Affix grammars over finite lattices. Report No. 94, Department of Computer Science, University of Nijmegen, Nijmegen, The Netherlands.