FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Extending Bison with Attribute Grammars

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Mihai Calin Ghete
Matrikelnummer 00418413

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Ao.Univ.Prof. Dr. M. Anton Ertl

Wien, 28.02.2018 _____ _____

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Erklärung zur Verfassung der Arbeit

Mihai Calin Ghete
Wienerbergstraße 12/20/10,
1120 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, am 28.02.2018

Extending Bison with Attribute Grammars

Master Thesis

Mihai Calin Ghete
Advisor: M. Anton Ertl

February 28, 2018

Acknowledgement

I would like to thank my advisor, Ao.Univ.Prof. Dr. M. Anton Ertl, for his continued guidance and support throughout the development of this thesis and for the useful insights he provided to that respect.

The resulting program code is dedicated to my family and friends, who eagerly supplied plenty of encouragement and moral support.

Abstract

Knuth's attribute grammars are a formalism for specifying the semantics of context-free languages. They imply an attribute evaluation step that requires finding a topological sorting of a directed acyclic graph. Historically, various approaches have been used to shift the resulting workload to the compile-time of evaluators or to reduce it by lowering the expressiveness of the formalism. This thesis describes the conceptual background and implementation of a fully expressive dynamic (run-time) attribute evaluator within an LALR(1) and GLR parser, and analyzes the trade-offs involved in both dynamic and various compile-time evaluation approaches. The resulting evaluator uses algorithms that are linear in time and space with regard to the size of the parse tree. Benchmarking suggests it is performant enough for practical purposes.
The evaluator supports GLR attribute grammars that result in a single valid parse tree. In addition, the core of the evaluator is made available as a separate run-time library.

Zusammenfassung

Attributierte Grammatiken wurden von Knuth eingeführt und stellen ein Formalismus dar, um die Semantik von kontextfreien Sprachen zu spezifizieren. Ein wesentlicher Bestandteil dessen ist die Attribut-Evaluation, die auf einer topologischen Sortierung eines gerichteten, zyklenfreien Graphens beruht. Diverse Ansätze wurden historisch entwickelt, um den resultierenden Rechenaufwand an die Übersetzungszeit des Evaluators zu verlagern oder diesen durch einer Abschwächung des Formalismus zu reduzieren. Diese Arbeit beschreibt den konzeptuellen Hintergrund und die Implementierung eines vollständigen dynamischen (Laufzeit-) Attributevaluators innerhalb eines LALR(1) und GLR-Parsers und vergleicht dazu den dynamischen Ansatz mit solchen Ansätzen, die zur Übersetzungszeit stattfinden. Die dem dynamischen Evaluator zugrundeliegenden Algorithmen haben lineare Laufzeit und linearen Speicherbedarf in der Größe des Syntax-Baumes. Vergleichstests zeigen, dass dieser für praktische Zwecke performant genug ist.
Der Evaluator unterstützt die Attributierung von GLR-Grammatiken, welche als Ergebnis einen einzigen gültigen Syntax-Baum haben. Der Kern des Evaluators steht außerdem auch als Laufzeit-Bibliothek zur Verfügung.

Contents

Chapter 1

Introduction

Attribute grammars [Knu68; Knu71] were introduced in 1968 by Donald E. Knuth and are a formalism for defining the meaning (semantics) of context-free languages. In this formalism, symbols may have a number of attributes associated with them. Each attribute encodes a semantic property of the corresponding symbol and is accompanied by a set of rules to calculate its value; in the context of such a rule, the value of an attribute may depend on that of other attributes.

This thesis describes the implementation of an evaluator for attribute grammars realized on top of GNU Bison [1], an open-source LALR(1) and GLR parser.

Two main approaches to attribute evaluation can be found throughout literature: static and dynamic. Static approaches use the specification of an attribute grammar to generate a custom evaluator for all possible inputs accepted by the grammar. Determining if an attribute grammar is valid and thus generating an evaluator for it was shown to have exponential complexity in the size of the grammar [JOR75]. For this reason, many static approaches concentrate on subclasses of attribute grammars that are not as expressive, but which can be analyzed and evaluated more efficiently. The subclasses' membership tests range from NP-complete to linear in the size of the grammar.

Dynamic approaches do not require in-depth analysis of the grammar specification. Instead, they rely on building a dependency graph at runtime from actual input and on a dependency resolution algorithm to schedule attribute evaluation. The underlying algorithms have linear complexity in the size of the parse tree. They work for all valid attribute grammars.

GNU Bison already supports a limited subset of attribute grammars via its "semantic actions". Owing to the parsing algorithm used by Bison, "semantic actions" do not require an explicit dependency graph to be built or a separate dependency resolution step to be performed. However, they also do not offer the full flexibility of attribute grammars as they only allow information to be passed in one direction through the parse tree, which substantially limits possible applications.

The goal of the thesis is to implement full attribute grammar support in GNU Bison. A dynamic evaluation approach is preferred due to the high flexibility and low complexity involved. As the evaluator's performance plays a key role in assessing its practical significance, benchmarks and comparisons are seen as an integral part of the resulting work.

[1] `http://www.gnu.org/software/bison`

GNU Bison can parse ambiguous context-free grammars using the Generalized LR (GLR) algorithm. Another aspect of the thesis is to determine how GLR and attribute grammars can be used together, as there are few documented approaches of doing so in practice.

A final implementation aspect is to determine whether the evaluator core should be separated into an external library for easier re-use.

The main objectives of the thesis are thus summarized by the following *research questions:*

- What are the steps required to implement an attribute evaluator in an existing parser generator?
- Is a dynamic evaluator fast enough for practical purposes?
- Generalized LR (GLR) is an extension of LR for parsing ambiguous context-free grammars. How can GLR best interact with attribute grammars?
- Is it advisable to separate the evaluator runtime into an external library?

The thesis itself begins with a description of context-free grammars and ways to parse them. It then continues with an in-depth explanation of attribute grammars and their properties, showing common methods of evaluating attributes. The problem of cycles (and cycle detection) in attribute grammars is also discussed. Attributed tree traversals are mentioned as a concept not immediately related to attribute grammars, but which can be integrated easily therein.

The design chapter features a discussion of advantages and disadvantages of various evaluation techniques, explaining the rationale behind implementing a dynamic attribute evaluator. It then goes into detail into the design of the dynamic evaluator itself and concludes with a discussion of potential approaches to integrating attribute grammars with GLR.

The implementation chapter contains data that was gathered while writing the program. It documents decisions that were taken during the creation of the evaluator, details the structure of the program, how it fits into Bison and explains all optimizations that were made.

The last chapter contains the results of benchmarks run on the implementation and comparisons with an existing system.

Chapter 2

Conceptual foundation of parsers and attribute grammars

Attribute grammars were conceived as an add-on to context free grammars; this chapter explains both of them. It first introduces context-free grammars in the context of the Chomsky hierarchy, explaining their characteristics both theoretically and informally. It then continues with a description of parsing methods that can determine whether a given input belongs to a context-free language and, if so, how exactly the input matches the description of the grammar. Attribute grammars and corresponding evaluation methods are explained further on; again, an informal and theoretical description of the formalism precedes a discussion on evaluation methods. Related aspects then follow, including cycle detection in attribute grammars, handling ambiguity (in particular with regard to GLR parsing) and tree traversals. This chapter concludes with a brief survey of existing dynamic attribute grammar evaluators.

2.1 Context-free grammars

Context-free (Type-2) grammars are one of the four types of formal grammars composing the Chomsky hierarchy [Cho56; Cho59]. The hierarchy describes several types of grammars, each a superset of the next, from most expressive (Type-0) to least expressive (Type-3).

Formal grammars are described using productions; each production has a left hand side and a right hand side, both of which can contain terminal and non-terminal symbols. All strings of terminal symbols that can be generated by iteratively substituting the left hand side of a production with its right hand side, starting with a predefined start symbol, compose the language accepted by that grammar. The grammars in the Chomsky hierarchy have different constraints imposed on their productions:

Type-0 Unrestricted grammars. Productions are of the form:

$$\alpha \to \beta$$

Both α and β may contain terminals, as well as non-terminals. The languages generated by such grammars are called recursively enumerable [Cho59, p. 143].

Type-1 Context-sensitive grammars. Productions are of the form:

$$\alpha A \beta \to \alpha \gamma \beta$$

In this case, α, β and γ may contain terminals and non-terminals; α and β may be empty. A is a non-terminal. The special property of these grammars is their "context", a non-terminal (A) may have a different meaning depending on its surroundings (α and β).

Type-2 Context-free grammars. Productions are of the form:

$$A \to \gamma$$

As in Type-1 grammars, A denotes a non-terminal and γ a string of terminals and non-terminals.

Type-3 Regular grammars. Two types of regular grammars – left regular and right regular grammars – exist, both having the ability to describe the same set of languages. They are discerned by the position of the optional non-terminal (B) with regard to the terminal (a). For example, productions for a right regular grammar may be of the form:

$$A \to a \qquad \text{or} \qquad A \to aB$$

Regular grammars describe the same languages as regular expressions and finite automata, the three formalisms are thus interchangeable.

Context-free grammars are popular in language design due to their relatively high expressivity and their manageable complexity. They are equivalent in power to pushdown automata [Sip06, p. 115 ff.,193]. This enables, among others, the relatively straightforward creation of a recursive descent parser for a given language.

While the typical formal question is whether an input belongs to the language generated by a context-free grammar or not, in practice the substitution steps required to derive the input, as well as the order in which they are performed, are of equal importance.

2.1.1 Formal definition

A context-free grammar is defined as a tuple [Sip06, p. 102] [ASU86, p. 26] $G = (V, \Sigma, P, Z)$:

V is the set of non-terminals used in the grammar.

Σ is the set of terminals.

P is the set of productions belonging to the grammar. Such a production has a non-terminal on the left side and zero or more symbols (terminals or non-terminals) on the right side.

Z is a designated start symbol from the set of non-terminals.

A *derivation* represents a series of transformations applied to the start symbol and the resulting strings, in each of which a non-terminal is replaced by the right hand side of a compatible production. The derivation ends when no more non-terminals can be replaced and the resulting string contains only terminals; a *sentence* of the grammar is then said to have been derived. In contrast, strings containing both terminals and non-terminals are called *sentential forms*.

The *language* generated by a grammar is the set of all sentences that can be derived using the grammar specification. Two grammars that generate the same language are said to be equivalent; while it is possible to construct an equivalent grammar in some cases, the general question whether two given grammars are equivalent is undecidable [Sip06, p. 197].

The sample grammar below describes a notation for addition of binary digits:

$$
\begin{aligned}
V &= \{S, E, T\} \\
\Sigma &= \{\underline{0}, \underline{1}, \underline{+}\} \\
P &= \{1.\ S \to E \\
&\quad\ 2.\ E \to E \underline{+} E \\
&\quad\ 3.\ E \to \underline{0} \\
&\quad\ 4.\ E \to \underline{1}\} \\
Z &= S
\end{aligned}
\tag{2.1}
$$

$\underline{0} \underline{+} \underline{1} \underline{+} \underline{0}$ is an example sentence in the given language. It can be shown that the sentence is in the language by providing a derivation for it. We begin by looking at the start symbol; we then apply several production rules:

$$
S \overset{Rule1}{\Rightarrow} E \overset{Rule2}{\Rightarrow} E \underline{+} E \overset{Rule3}{\Rightarrow} \underline{0} \underline{+} E \overset{Rule2}{\Rightarrow} \underline{0} \underline{+} E \underline{+} E \overset{Rule4}{\Rightarrow} \underline{0} \underline{+} \underline{1} \underline{+} E \overset{Rule3}{\Rightarrow} \underline{0} \underline{+} \underline{1} \underline{+} \underline{0}
$$

The symbol $\Rightarrow$ is used to denote a single derivation step. To sum up zero or more steps, or, in other words, denote that a sentential form is derivable from another sentential form, $\overset{*}{\Rightarrow}$ can be employed.

When performing a derivation step, two choices must be made [ASU86, p. 167-169]:

- Which non-terminal to replace?
- What production to use?

With regard to the first question, two important derivation types are discussed in literature: leftmost and rightmost (also called canonical). They are performed by always replacing the leftmost (or, respectively, rightmost) non-terminal in a sentential form.

The previous example shows a leftmost derivation; a possible rightmost derivation would be:

$$
S \Rightarrow E \Rightarrow E \underline{+} E \Rightarrow E \underline{+} \underline{0} \Rightarrow E \underline{+} E \underline{+} \underline{0} \Rightarrow E \underline{+} \underline{1} \underline{+} \underline{0} \Rightarrow \underline{0} \underline{+} \underline{1} \underline{+} \underline{0}
$$

The difference between the derivations can be see in the third step; the leftmost E is replaced by a $\underline{0}$ in the first example, while the rightmost E is replaced in the second example.

More than one leftmost or rightmost derivation can exist. To exemplify this, a second leftmost derivation is given:

$$
S \Rightarrow E \Rightarrow E \underline{+} E \Rightarrow E \underline{+} E \underline{+} E \Rightarrow \underline{0} \underline{+} E \underline{+} E \Rightarrow \underline{0} \underline{+} \underline{1} \underline{+} E \Rightarrow \underline{0} \underline{+} \underline{1} \underline{+} \underline{0}
$$

In this case, rule 2 is used in the third step (on the leftmost instance of E) to expand $E \underline{+} E$ to $E \underline{+} E \underline{+} E$, diverging from the first example, where rule 3 is used to expand an E into a $\underline{0}$. Similarly, rule 2 could be applied twice in a row on the rightmost derivation example.

2.1.2 Derivations, parse trees and ambiguity

A *parse tree* is a graphical representation of a derivation. The children of each non-terminal node represent the symbols on the right side of the production used to expand that non-terminal. A parse tree is *complete* if all its leaves are non-terminals; reading the leaves from left to right yields the parsed sentence.

Parse trees do not contain information about the order in which derivation steps are performed; the parse tree for a leftmost or rightmost derivation of a certain string may be identical.

Using the same grammar as above, we analyze the parse tree for the sentence $\underline{0} + \underline{1}$. While two derivations do exist (a leftmost and a rightmost one), the parse tree is the same, as shown in Figure 2.1.

$$S \Rightarrow E \Rightarrow E + E \Rightarrow \underline{0} + E \Rightarrow \underline{0} + \underline{1}$$
$$S \Rightarrow E \Rightarrow E + E \Rightarrow E + \underline{1} \Rightarrow \underline{0} + \underline{1}$$

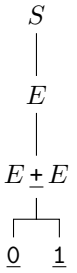


Figure 2.1: Both derivations of $\underline{0} + \underline{1}$ have the same parse tree.

A grammar is said to be *ambiguous* when there can be more than one parse tree for a sentence. This also equates to more than one leftmost or rightmost derivation being possible.

The given grammar has this property. Two parse trees for $\underline{0} + \underline{1} + \underline{0}$ are shown in Figure 2.2. In this case, both parse trees can arise from leftmost and rightmost derivations, depending on whether the first rule applied to $E + E$ derives a terminal or another instance of $E + E$.

Figure 2.2: Parse trees for $\underline{0} + \underline{1} + \underline{0}$

In applications such as language design, where semantics are tied to syntax, ambiguous grammars are avoided as this would give sentences in the language more than one meaning. Techniques to remove ambiguity from grammars range from rewriting the grammar specifications to adding special rules (for example "precedence rules") that only allow one derivation and discard the rest. The latter option is used in practice and not specified by the context-free grammar formalism.

2.1.3 Subclasses of context-free grammars

Several subclasses of context-free grammars have been researched alongside corresponding algorithms, allowing the creation/generation of more efficient parsers:

LL Left-to-right leftmost derivation. In this case, the input is parsed from the leftmost to the rightmost token by applying a production that matches the leftmost non-terminal of the current derivation string. For example, consider the input $\underline{\texttt{x}} \underline{+} \underline{\texttt{y}}$ and the grammar:

$$\begin{aligned} S &\rightarrow E \underline{+} E \\ E &\rightarrow \underline{\texttt{x}} \\ E &\rightarrow \underline{\texttt{y}} \end{aligned} \tag{2.2}$$

S is the start symbol. An LL parser would perform the following substitutions:

$$S \Rightarrow E \underline{+} E \Rightarrow \underline{\texttt{x}} \underline{+} E \Rightarrow \underline{\texttt{x}} \underline{+} \underline{\texttt{y}}$$

LL parsers are top-down parsers. They begin their derivation with the start symbol, gradually performing substitutions on it until the input string has been derived. When necessary, such parsers look ahead into the input stream to ensure the correct production is chosen for a substitution.

LR Left-to-right rightmost derivation. The input is parsed from the leftmost to the rightmost token, however, as opposed to LL grammars, a production that matches the rightmost non-terminal of the derivation string is chosen. Thus, given the previous example, an LR parser would perform the following substitutions:

$$S \Rightarrow E \underline{+} E \Rightarrow E \underline{+} \underline{\texttt{x}} \Rightarrow \underline{\texttt{x}} \underline{+} \underline{\texttt{y}}$$

LR parsers are bottom-up parsers. When parsing an input token, the algorithm derives "backwards" by substituting the already parsed symbols, together with the input token with a corresponding left hand side of a production. The last step of an LR parser thus involves obtaining the start symbol with no more input to parse.

LALR Look-ahead LR. LALR is an LR parsing method that represents a compromise between two algorithms: the compact, but less expressive simple LR (SLR) and the more space-consuming canonical LR. LALR trades in part of canonical LR's expressiveness for space efficiency and is implemented in parsers such as Bison.

GLR Generalized LR. GLR parsers [Tom84; Tom87] were introduced to handle ambiguous LR grammars. In the case of a parsing conflict (an ambiguity in the grammar), a GLR parser "splits up" and pursues all alternatives. Alternatives that lead to errors are discarded. Furthermore, optimizations are made to share common elements of the parsers' states and of the resulting parse trees.

A detailed discussion on how to parse such grammars can be found in Appendix A.1. Information on GLR in particular is available in Appendix A.1.3.

2.2 Attribute grammars

2.2.1 The concept

Attribute grammars (AG) are an extension to context-free grammars. In this formalism, grammar symbols may have *attributes*, each of which describes a certain semantic property of that

symbol. Since multiple instances of a symbol may be derived for a certain input, each instance of the symbol has its own *attribute instances*. For example, consider the following grammar (adapted from Knuth's original paper [Knu68, p. 128 ff.]):

$$\begin{aligned} L &\rightarrow LB \mid B \\ B &\rightarrow \underline{0} \mid \underline{1} \end{aligned} \tag{2.3}$$

We would like the non-terminal L to describe binary numbers and B to describe a single binary digit. For this purpose, we define *value* to be an attribute of L and of B.

For the input string $\underline{1}\,\underline{0}\,\underline{1}$, we have three instances of B, whose *value*s should be 1, 0 and 1, respectively. We also have three instances of L, whose *value*s should be 1_2, $10_2 = 2$ and $101_2 = 5$ (see Figure 2.3).

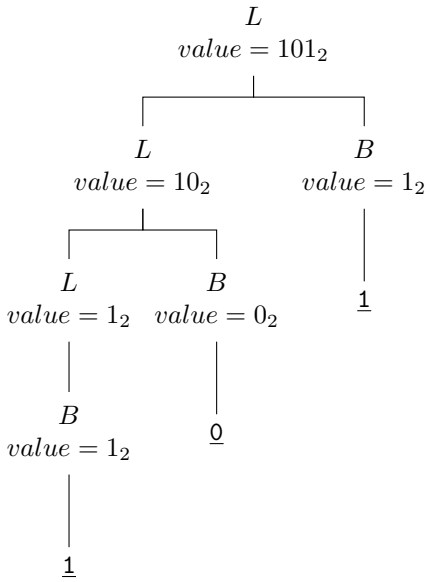


Figure 2.3: Attributed parse tree for Grammar 2.3 and input $\underline{1}\,\underline{0}\,\underline{1}$

Simply specifying that an attribute exists and giving attribute values for some fixed input is not sufficient; we need a generic way to calculate them. For this purpose, a set of semantic rules is attached to each production. Such a semantic rule describes how to calculate the value of a symbol's attribute based on attributes of other symbols in the production. For Grammar 2.3, such rules can be defined as follows [1]:

$$\begin{aligned} L_0 &\rightarrow L_1 B && \{\ L_0.value := 2 \cdot L_1.value + B.value;\ \} \\ L_0 &\rightarrow B && \{\ L_0.value := B.value;\ \} \\ B &\rightarrow \underline{0} && \{\ B.value := 0;\ \} \\ B &\rightarrow \underline{1} && \{\ B.value := 1;\ \} \end{aligned} \tag{2.4}$$

[1] L_0 and L_1 denote the same non-terminal L, but are used to disambiguate the two in the semantic rule.

The first rule, for example, specifies that the *value* of the left hand side L_0 is twice the *value* of L_1 on the right hand side, plus the *value* of B. Thus, $L_1.value$ and $B.value$ are both *dependencies* of $L_0.value$.

As we can see, semantic rules use only the local context of a production, which encompasses the attributes of all symbols in that production. By definition, there are two kinds of attributes [AM91, p. 127-128] [Knu68]:

Synthesized attributes depend on attributes of the current symbol and on attributes of its descendants in the parse tree. Within the context of a single production, synthesized attributes can only be defined for the left hand side symbol.

Inherited attributes depend on attributes of the current symbol and on those of its ancestors in the parse tree. In a production, such attributes are defined for a symbol on the right hand side.

In the previous example (AG 2.4), *value* is a synthesized attribute, since the value of an instance is always determined using attributes of child nodes. To show how inherited attributes work, the grammar is modified to use the position (or "place value") of a binary digit when calculating the digit's *value*. An attribute *position* for L and B is introduced for this purpose. The rightmost digit in a number has *position* 0, the digit to its left has *position* 1, and so on. This results in a *value* of 0 or 1 for the rightmost digit, 0 or 2 for the digit to its left, and so on.

As a first step, a new start symbol, N, is added to the grammar, featuring a semantic rule that defines the initial position:

$$\begin{array}{lcll} N & \to & L & \{ \quad N.value := L.value; \quad \} \\ & & & \{ \quad L.position := 0; \quad \} \end{array}$$

The semantic rules for $L.value$ are modified to use simple addition instead of multiplication by two:

$$\begin{array}{lcll} L_0 & \to & L_1 B & \{ \quad L_0.value := L_1.value + B.value; \quad \} \\ L_0 & \to & B & \{ \quad L_0.value := B.value; \quad \} \end{array}$$

The rules for $B.value$ are also modified to take the position of the digit into account:

$$\begin{array}{lcll} B & \to & \underline{0} & \{ \quad B.value := 0; \quad \} \\ B & \to & \underline{1} & \{ \quad B.value := 2^{B.position}; \quad \} \end{array}$$

Additionally, rules to increment and propagate *position* are added:

$$\begin{array}{lcll} L_0 & \to & L_1 B & \{ \quad \ldots \quad \} \\ & & & \{ \quad L_1.position := L_0.position + 1; \quad \} \\ & & & \{ \quad B.position := L_0.position; \quad \} \\ L_0 & \to & B & \{ \quad \ldots \quad \} \\ & & & \{ \quad B.position := L_0.position; \quad \} \end{array}$$

The attribute grammar has thus been successfully modified to use the inherited attribute *position* when calculating its value. AG 2.5 contains the full listing:

$$
\begin{array}{lcll}
N & \to & L & \{ \quad N.value := L.value; \quad \} \\
 & & & \{ \quad L.position := 0; \quad \} \\
L_0 & \to & L_1 B & \{ \quad L_0.value := L_1.value + B.value; \quad \} \\
 & & & \{ \quad L_1.position := L_0.position + 1; \quad \} \\
 & & & \{ \quad B.position := L_0.position; \quad \} \\
L_0 & \to & B & \{ \quad L_0.value := B.value; \quad \} \\
 & & & \{ \quad B.position := L_0.position; \quad \} \\
B & \to & \underline{\texttt{0}} & \{ \quad B.value := 0; \quad \} \\
B & \to & \underline{\texttt{1}} & \{ \quad B.value := 2^{B.position}; \quad \}
\end{array}
\tag{2.5}
$$

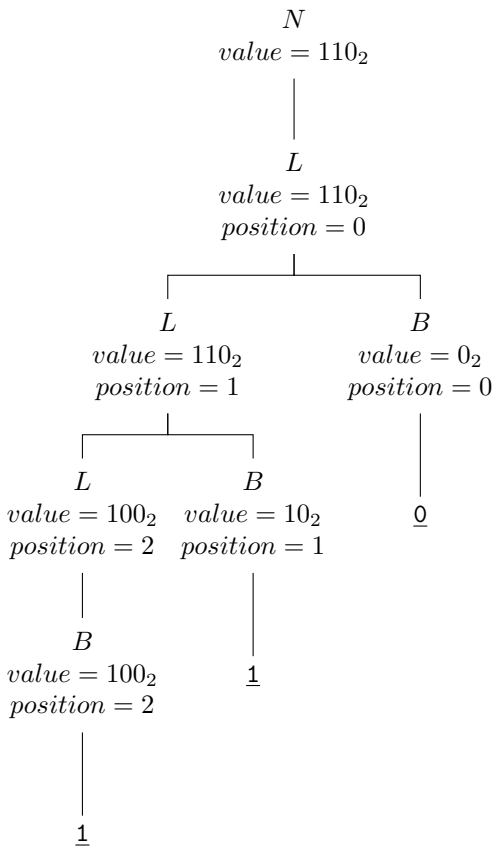


Figure 2.4: Attributed parse tree for grammar 2.5 and input $\underline{\texttt{1}}\ \underline{\texttt{1}}\ \underline{\texttt{0}}$

Figure 2.4 shows the complete attributed parse tree for input $\underline{\texttt{1}}\ \underline{\texttt{1}}\ \underline{\texttt{0}}$. While the parse tree does list the values of all attribute instances, it does not show the order in which the values were calculated. In order to synthesize $L.value$, an instance of L first has to inherit *position* and obtain the *value* of its children. A possible evaluation order is shown in Figure 2.5. There, *position* is first inherited to all instances of L and B, after which $B.value$ and $L.value$ are calculated in a bottom-up fashion.

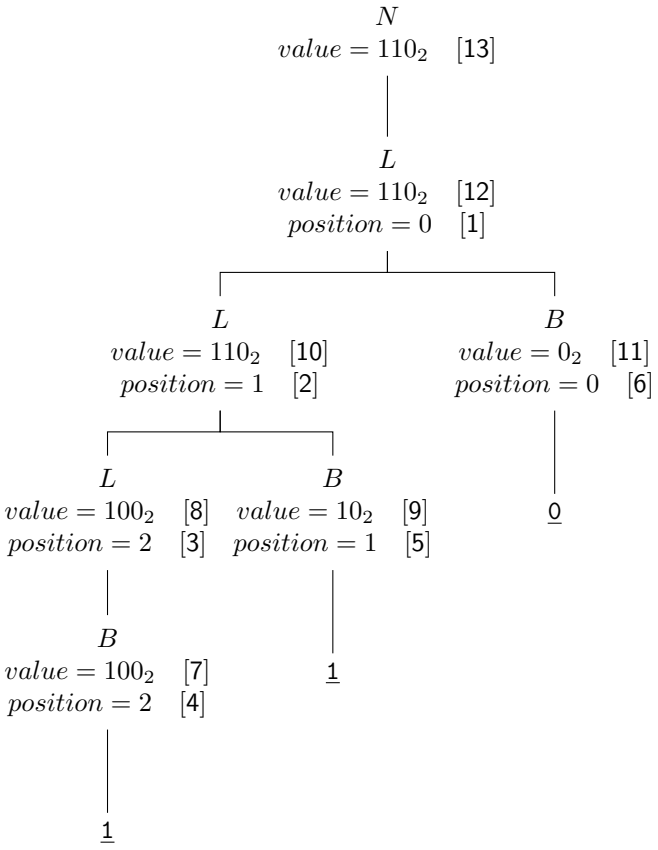


Figure 2.5: Attributed parse tree for grammar 2.5 and input 1 1 0, with evaluation order in brackets

The precise order in which the attributes are calculated is irrelevant, as long as the dependencies specified by the semantic rules are fulfilled.

2.2.2 Formal definition

Formally, an attribute grammar is a tuple [AM91, p. 2][Knu68, p. 131 ff.] $AG = (G, SD, AD, R, C)$ where:

$G = (V, \Sigma, P, Z)$ is a context-free grammar as described in Section 2.1.1.

$SD = (\mathit{TYPE\text{-}SET}, \mathit{FUNC\text{-}SET})$ is a so-called *semantic domain*. More precisely, *TYPE-SET* contains the definition of possible attribute types and is structured as a set of sets $\{type_1, type_2, \ldots type_n\}$. *FUNC-SET* is a set of functions with the signature $type_{i_1} \times type_{i_2} \times type_{i_n} \rightarrow type_{i_0}$, $n \geq 0$ that are used within semantic rules.

$AD = (A, I, S, \mathit{TYPE})$ is an attribute definition composed of several sets. For each symbol $X \in V$, $A(X)$ is defined to be the set of all attributes of X; $I(X)$ and $S(X)$ are the (by definition disjoint) sets of inherited and synthesized attributes, respectively. $\mathit{TYPE}(a) \in \mathit{TYPE\text{-}SET}$ is the set of all possible values of an attribute $a \in A(X)$. Within this formal definition, attributes are unique to symbols, so, for example, two attributes of non-identical symbols $X.a$ and $Y.a$ are different [AM91, p. 2].

$R(p)$ is the set of *semantic rules* associated with production $p \in P$. Each element of $R(p)$ has the form [AM91, p. 3]:

$$(a_0, p, k_0) := f((a_1, p, k_1), \ldots, (a_m, p, k_m))$$

Here, a_0 and $a_1, \ldots, a_m$ are attributes belonging to the symbols present in production $p = X_{p0} \rightarrow X_{p1} X_{p2} \ldots X_n$. An *attribute occurrence* (a, p, k) denotes the attribute instance of a for the symbol X_{pk} of production p. In this case, (a_0, p, k_0) is said to *depend* on $(a_1, p, k_1), \ldots, (a_m, p, k_m)$.

C is described as a "set of semantic conditions associated with production p" [AM91, p. 4]. A condition can be used as a constraint on the sentences accepted by the production and is specified as a function of attribute occurrences. This function yields *true* if the sentence is semantically correct and $false$ if it is not.

The attribute occurrences for a production p are denoted with $AO(p)$; this set is further split into two disjunct sets, $DO(p)$ and $UO(p)$. $DO(p)$ contains the *defined attribute occurrences* within p and $UO(p)$ contains the *used attribute occurrences*, as featured in semantic rules.

More precisely, $DO(p)$ encompasses all synthesized attributes of the left hand side and all inherited attributes of the right hand side:

$$DO(p) = \{(a, p, 0) \mid a \in S(X_{p0})\} \cup \{(a, p, k) \mid a \in I(X_{pk})\}$$

$UO(p)$ encompasses all inherited attributes of the left hand side and all synthesized attributes of the right hand side:

$$UO(p) = \{(a, p, 0) \mid a \in I(X_{p0})\} \cup \{(a, p, k) \mid a \in S(X_{pk})\}$$

The set of semantic rules $R(p)$ is restricted such that all elements (a_0, p, k_0) on the left side of a rule belong to $DO(p)$. Furthermore, if all elements on the right hand side belong to $UO(p)$,

for all rules of all productions, the grammar is said to be in *normal form* [AM91, p. 3]. Using a series of substitutions, all elements belonging to $DO(p)$ on the right hand side can be expressed using elements from $UO(p)$, thus, attribute grammars can be easily transformed to normal form.

2.3 Evaluating attribute grammars

The semantic rules of an attribute grammar are said to be *well-defined* if all attribute instances can be calculated using that set of rules for all possible parse trees [Knu68, p. 133]. More precisely, each production has to be annotated with semantic rules such that each of the synthesized attributes of the left hand side symbol, as well as all of the inherited attributes of the right hand side symbols can be calculated. Furthermore, no cyclical dependencies between attributes may arise. If the rules of an attribute grammar are well-defined, they can be applied to the parse tree obtained from a certain input. This is called *evaluating* the attribute grammar.

During the evaluation process, a so-called *attributed parse tree* is constructed. For each node N of a parse tree T, let $X \in V$ be the *label* of N, the symbol associated with it. N has a set of *attribute instances* $a_0 \dots a_m$ connected to it, where each attribute instance corresponds to an attribute in $A(X)$. Furthermore, *attribute evaluation instructions* [AM91, p. 4] of the form:

$$N_k.a := f(N_{k_1}.a_1, \dots, N_{k_m}.a_m)$$

are instantiated for each semantic rule:

$$(a, p, k) := f((a_1, p, k_1), \dots, (a_m, p, k_m))$$

in $R(p)$. If all attribute instances for all nodes have been calculated, the parse tree is said to be *attributed* and the evaluation process complete.

Dependencies within attribute evaluation rules need to be transferred when creating attribute instances. For a parse tree T, its *dependency graph* $D(T)$ is defined as follows:

- The vertices of $D(T)$ are the attribute instances contained in T.
- An edge $(N_i.a, N_j.b)$ exists for $D(T)$ iff $N_j.b$ depends on $N_i.a$.

An attribute instance $N.a$ can only be calculated if all of its dependencies ($N_i.a_i$ where $(N_i.a_i, N.a) \in E(D(T))$) have been calculated. If the graph is acyclic, the edges impose a partial order on the vertices of the dependency graph.

A *topological sort*, a total order of the vertices in $D(T)$ that also satisfies the partial order, can then be calculated. The topological sort specifies a valid evaluation order of the attribute evaluation instructions; the instructions are executed in the order their associated attribute instances appear.

The two steps that an attribute evaluator must execute are [DJL88, p. 9]:

1. Building a *plan*, that is, the construction of an evaluation order.
2. *Attribute computation*, the actual attribution of the parse tree.

In practice, these steps are often interleaved.

2.3.1 Analyzing the parse tree

One can analyze the behaviour of attribute dependencies within a given parse tree as follows [AM91, p. 11] [2]:

IS Given a subtree "T/N" of T with the root N, the dependency graph $D(T/N)$ can be calculated analogously to $D(T)$. The "i-to-s" behaviour of N, $IS(T/N)$ is defined as a directed graph with the attributes of X, N's label, as its vertices. Edges exist from a vertex corresponding to an inherited attribute i of X to a vertex corresponding to a synthesized attribute s of X if and only if there is a path in $D(T/N)$ from $N.i$ to $N.s$.

SI Given a parse tree T with the subtree T/N subtracted from it (except for the node N itself), a dependency graph $D(T - T/N)$ can be defined for the result, "$T - T/N$". The "s-to-i" behaviour of N, $SI(T - T/N)$, is defined as a directed graph with the attributes of X, N's label, as its vertices. Edges exist from a synthesized attribute s to an inherited attribute i if and only if there is a path in $D(T - T/N)$ from $N.s$ to $N.i$.

By analyzing all possible trees T for a given attribute grammar, one can define the sets *IS-SET*(X) and *SI-SET*(X) for the grammar's symbols by deriving them from the IS and SI graphs of all nodes labeled with X:

$$\begin{aligned} \textit{IS-SET}(X) &:= \{IS(T/N) \mid T = \text{parse tree}, X = \text{label of } N\} \\ \textit{SI-SET}(X) &:= \{SI(T - T/N) \mid T = \text{parse tree}, X = \text{label of } N\} \end{aligned}$$

Since, for all symbols of the grammar, the IS and SI graphs have a number of vertices limited by $A(X)$, both the *IS-SET* and *SI-SET* are finite (although the number of graphs contained within a set can be exponential with relation to the number of vertices).

Both the *IS-SET* and the *SI-SET* can be calculated using a fixed point algorithm, whereas *SI-SET* also depends on *IS-SET* [AM91, p. 13-14]. In order to understand the calculation and the further analysis and classification of attribute grammars, more definitions have to be introduced:

DG_p is the dependency graph of a single production. The nodes of this dependency graph represent the attribute occurrences within the production, its edges are given by the dependency rules specified for $DO(p)$.

$DG_p[D_0,\ldots,D_{n_p}]$ is an extension of DG_p where edges are added for two attribute occurrences (a, p, i) and (b, p, i) if the graph D_i contains that corresponding edge, meaning $(X_{pi}.a, X_{pi}.b) \in E(D_i)$. Here, D_i is not given explicitly, it may be any arbitrary graph with $A(X_{pi})$ as vertices. n_p is the number of symbols on the right hand side of production p.

$DG_p - k[\ldots]$ is used to denote $DG_p[D_0, \ldots, D_{k-1}, D_{k+1}, \ldots, D_{n_p}]$, which has the same effect as setting D_k to a graph with no edges.

$DG_p - k^*[\ldots]$ is based on $DG_p - k[\ldots]^*$, the *transitive closure* of $DG_p - k[\ldots]$. It only contains the attributes of X_k as vertices. If an edge $((a, p, k), (b, p, k))$ exists in $DG_p - k[\ldots]^*$, then an edge (a, b) exists in $DG_p - k^*[\ldots]$.

[2] Different notations are used throughout literature, such as $sd_t(X_0)$ for $IS(t/X_0)$, $id_t(X_u)$ for $SI(t - t/X_0)$ and $USD(X)$ for *IS-SET*(X), $UID(X)$ for *SI-SET*(X) [DJL88, p. 5-6]

Calculation of *IS-SET* and *SI-SET*

IS-SET is initialized with an empty set for all non-terminals and with a set containing a graph with no edges for all terminals. During an iteration, for productions where the *IS-SET*s of all symbols on the right hand side are non-empty, a new graph is constructed by using graphs from each of the right hand side symbols' *IS-SET*s and by combining them with the local dependency graph to form $DG_p - 0^*[\ldots]$. The graph is then added to the production's *IS-SET* [AM91, p. 13].

```
for all X ∈ V:
  IS-SET(X) := {}
for all X ∈ Σ:
  IS-SET(X) := {Empty graph with vertices A(X)}

do until no more graphs can be added:
  for each production p := X_0 → X_1 ... X_n:
    if IS-SET(X_1) ... IS-SET(X_n) are non-empty:
      for i := 1 to n:
        D_i := Random graph from IS-SET(X_i)

      G := DG_p − 0*[D_1, ..., D_n]
      if G ∉ IS-SET(X_0):
        Add G to IS-SET(X_0)
```

Algorithm 2.1: Calculation of the *IS-SET*s

Analogously, *SI-SET* can be calculated iteratively by starting with an empty set for all symbols, except for the start symbol, whose set is initialized with a single graph with no edges. In an iteration, all productions whose left-hand-side symbol's *SI-SET* and right hand side symbols' *IS-SET*s are non-empty are considered. The k-th symbol of the production is picked and for it $DG_p - k^*[D_0, \ldots, D_{k-1}, D_{k+1}, \ldots, D_{n_p}]$ is calculated and added to *SI-SET*(X_k), where D_0 is a graph from *SI-SET*(X_0) and D_i $(1 \leq i \leq n)$ is a graph from *IS-SET*(X_i).

```
for all X ∈ V:
  SI-SET(X) := {}

SI-SET(Z) := {Empty graph with vertices A(Z)}

do until no more graphs can be added:
  for each production p := X_0 → X_1 ... X_n:
    if SI-SET(X_0) is non-empty:
      Pick a graph D_0 from SI-SET(X_0)
      Pick a k between 1 and n
      for i := 1 to n:
        if i ≠ k:
          D_i := Random graph from IS-SET(X_i)

      G := DG_p − k*[D_0, ..., D_{k−1}, D_{k+1}, D_n]
      if G ∉ SI-SET(X_k):
        Add G to SI-SET(X_k)
```

Algorithm 2.2: Calculation of the *SI-SET*s

Use of *IS-SET* and *SI-SET*

These sets can be used to determine whether a grammar is circular or to calculate a fixed evaluation order.

A way of determining the circularity of a grammar by using *IS-SET*s is given in Algorithm 2.3. For each production, each combination of *IS-SET* graphs $D_1, \ldots, D_n$ corresponding to the right hand side symbols $X_1, \ldots, X_n$ is tested; if $DG_p - 0[D_1, \ldots, D_n]$ contains a cycle, then the grammar is circular and thus not well-defined.

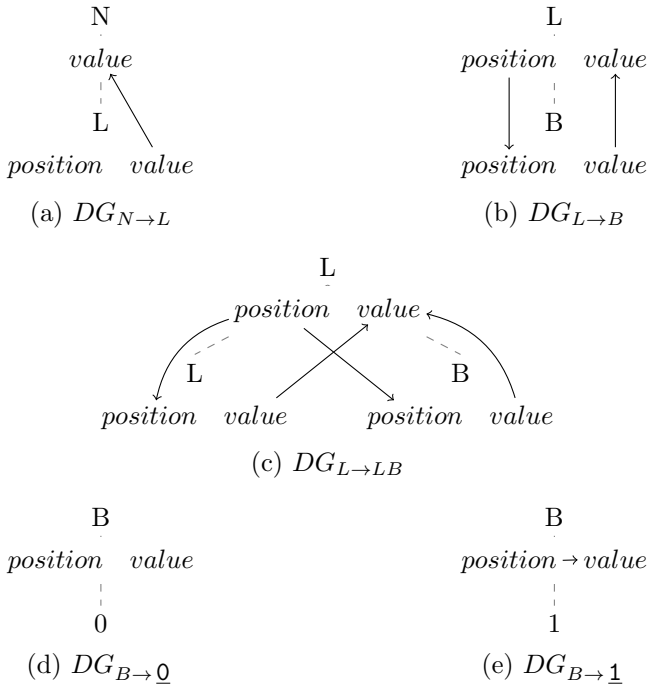
```
for each production p := X_0 → X_1 ... X_n :
      for each D_1 ∈ IS-SET(X_1), ..., D_n ∈ IS-SET(X_n) :
        if DG_p − 0[D_1, ..., D_n] contains a cycle:
          fail /* Grammar is circular */

success /* Grammar is non−circular */
```

Algorithm 2.3: Membership test for well-defined grammars [AM91, p. 14]

The problem of circularity is addressed in more detail in Section 2.4.

Example

Given AG 2.5, one can first obtain the local dependency graph for each production as given in Figure 2.6.



(a) $DG_{N \to L}$ (b) $DG_{L \to B}$

(c) $DG_{L \to LB}$

(d) $DG_{B \to \underline{0}}$ (e) $DG_{B \to \underline{1}}$

Figure 2.6: The local dependency graphs, DG_p, for AG 2.5

From the local dependency graphs, *IS-SET*(X) can be calculated for all symbols (see Figure 2.7). In this case there are two possible "i-to-s" behaviours of B, one in which *value* depends

on *position* (production $B \to \underline{1}$) and one in which it does not (production $B \to \underline{0}$). These behaviours are reflected in *IS-SET* (B) and propagate further to *IS-SET* (L).

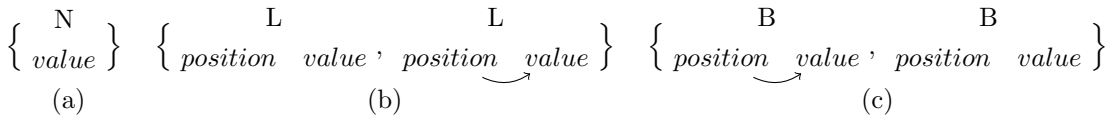


Figure 2.7: The *IS-SET*s of the symbols in AG 2.5

In order to check if the grammar is well-defined, for each production, edges from all possible combinations of *IS-SET* graphs of the right hand side symbols are inserted into the dependency graph of that production and the resulting graphs are checked for cycles. The graphs for AG 2.5 are shown in Figure 2.8. In this case, a total of ten graphs must be analyzed.

As none of the graphs exhibits a cycle, the grammar is well-defined.

2.3.2 Static evaluation of attribute grammars

A number of static analysis techniques has been devised for attribute grammars [DJL88; EF89]. Such approaches use the attribute grammar specification to determine possible behaviours of attributes in parse trees.

These behaviours can be used to determine if an attribute grammar is well-defined and to then create an evaluator for it. Various types of evaluators exist, including such based on *visits*, *passes* and *sweeps*.

Due to the exponential complexity involved in handling all well-defined attribute grammars, several subclasses of AGs have been searched for and analyzed throughout literature.

Static analysis techniques are detailed further in Appendix A.2.

2.3.3 Dynamic attribute evaluation

In a parse tree, each node is associated with a matching symbol from the context-free grammar. Attribute instances associated with a certain node correspond to the attributes defined for its symbol; their dependencies arise from the local dependency graph DG_p applied to the node (and the nodes directly above or below in the parse tree).

Since attribute instances can depend on other attribute instances, one can build a *dependency graph* encompassing all the instances that have to be calculated at runtime, with the edges given by the dependencies between them. A partial order of the nodes can be found such that if A depends on B, $B < A$. From this, a topological sort of the graph, which is a total order based on $B < A$, can be derived. Evaluation rules for calculating the attributes' values may then be called as per the total order, finalizing the evaluation process.

Dynamic evaluation does not concern itself with the preprocessing and grammar analysis techniques hinted at in the previous subsection (and detailed in Appendix A.2) or with the construction of a predefined plan for the grammar. Instead, it relies on the dependency resolution algorithm to evaluate the attribute instances in the right order either during or after parse tree construction.

The dependency graph of an entire parse tree can be assembled from all local dependency graphs of the productions applied on the nodes of the parse tree. In the context of a dependency graph, attribute instances are referred to directly as *nodes* of the dependency graph.

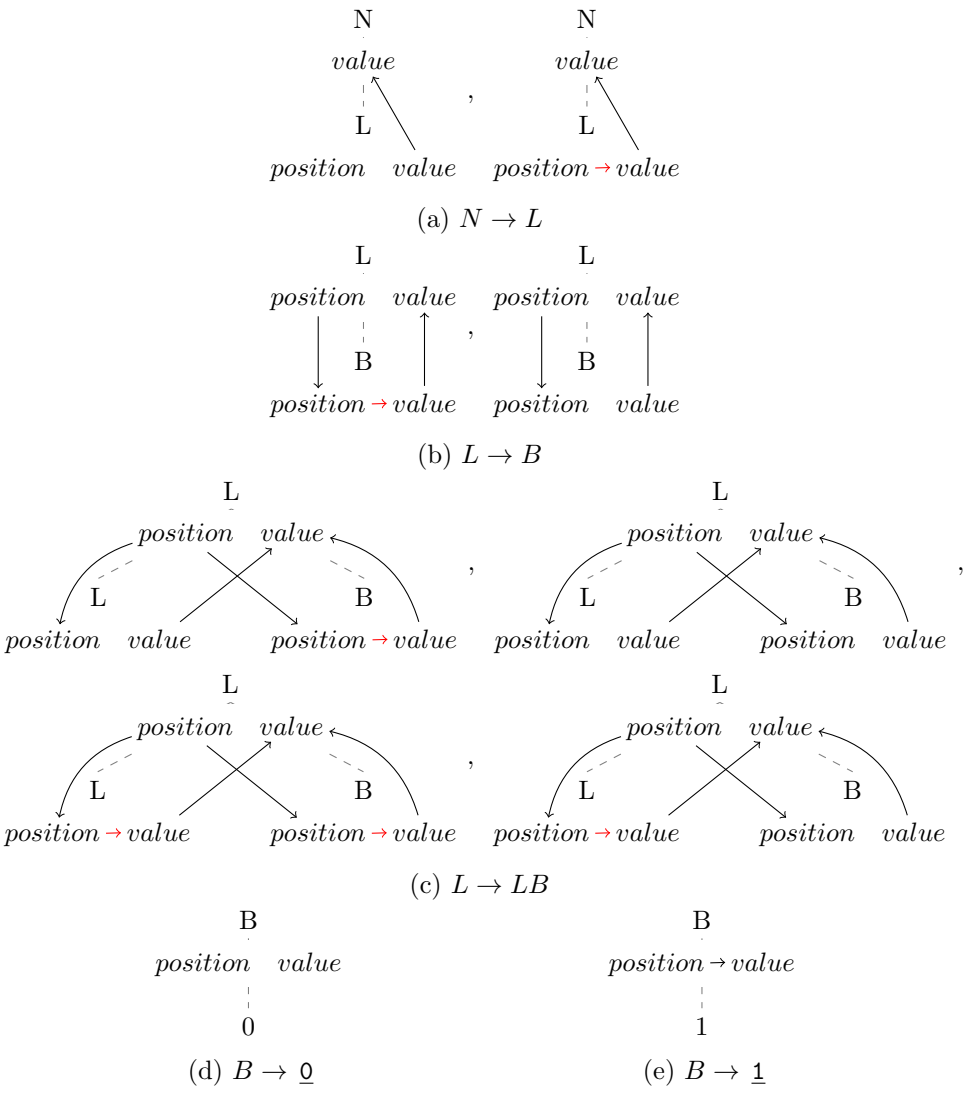


Figure 2.8: Dependency graphs from Figure 2.6 with combinations of edges from the *IS-SET* graphs in Figure 2.7

The parse tree from Figure 2.4 is reproduced here, with the dependency graph overlaid. Edges from DG_p (Figure 2.6) are added whenever a node plus its children match production p. For example, the labels of the root N_I and its child L_I match the symbols of production $N \to L$, so all edges from $DG_{N \to L}$ (Figure 2.6 (a)) are transferred. In this case, there is only one edge: $L_\mathrm{I}.value \to N_\mathrm{I}.value$.

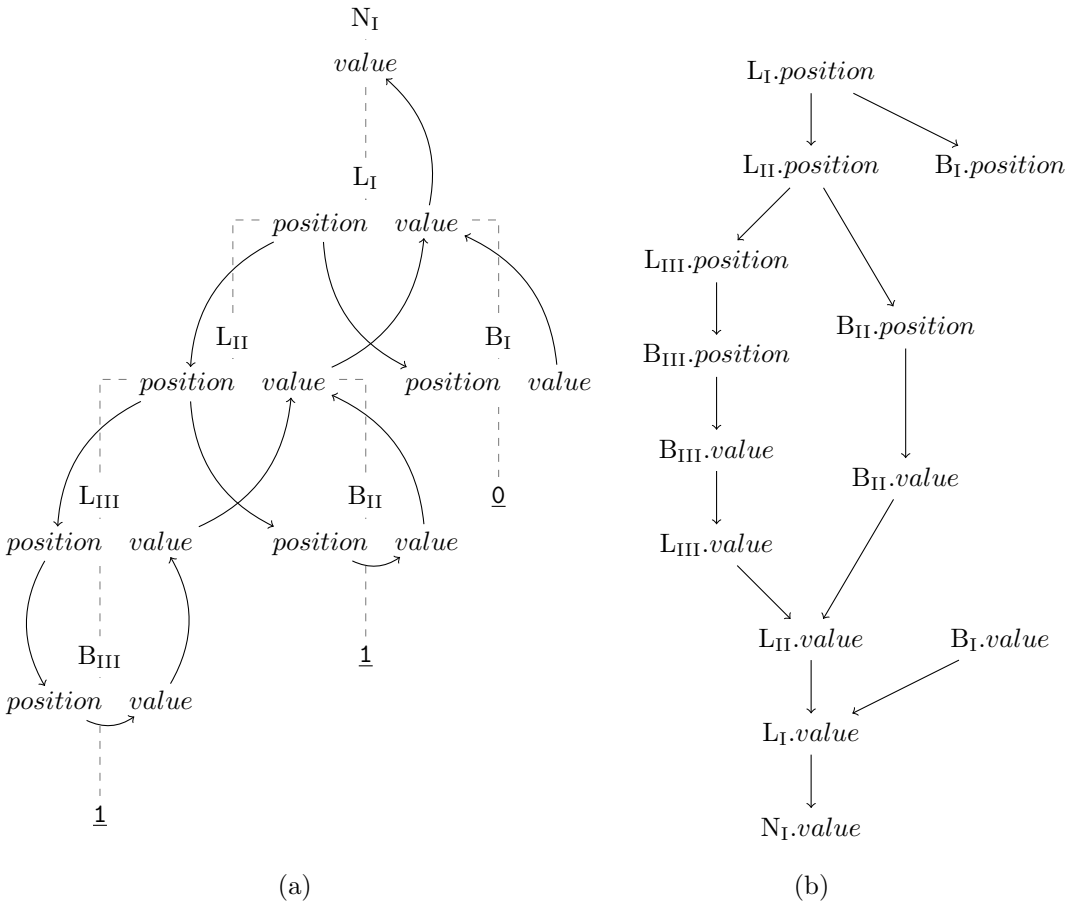


Figure 2.9: Attributed parse tree for AG 2.5 and input 1 1 0, with dependency graph overlaid (a) and resulting dependency graph (b).

There are two main approaches for dynamic evaluation: *eager* (or supply-driven) and *lazy* (or demand-driven). They differ in the order they traverse the dependency graph – eager algorithms begin with nodes that have no incoming edges, meaning their dependencies have already been fulfilled, while lazy algorithms are given a set of required nodes and from there work their way iteratively through all nodes that they depend on.

The lazy algorithm is based on depth-first search and can be easily implemented as a recursive procedure (see Algorithm 2.4). In this case, the function *evaluate* is called for each node in the dependency graph; if only a certain set of nodes have to be calculated, the number of calls can be reduced.

```
1 T := {}
2
```

```
3  function evaluate(n):
4    if n ∈ T:
5      return
6
7    for each node m where the edge m → n exists:
8      evaluate(m)
9
10   append n to T
11
12 /* Evaluate all nodes in tree */
13 for each node n:
14   evaluate(n)
15
16 /* T contains the evaluation order */
```

Algorithm 2.4: Lazy (demand-driven) attribute evaluation

Figure 2.10 shows the algorithm at work. In order to calculate $N_{\mathrm{I}}.value$, the function *evaluate* must be called for each attribute node in part. The numbered arrows denote the order in which nodes are visited (calls to $evaluate(a)$ for each node a). When a node is appended to T, the sequence of this event is marked using a circled number near the node.

Backtracking steps (returns of the *evaluate* function) are not listed.

The eager algorithm involves iteratively finding nodes with no edges and removing them from the graph:

```
1  S := set of all nodes;
2  T := {};
3
4  while S contains nodes with no incoming edges:
5    for all nodes n with no incoming edges:
6      for all nodes m where the edge n → m exists:
7        remove n → m;
8      S -= n;
9      T += n;
10
11 /* T now contains the evaluation order */
```

Algorithm 2.5: Eager (supply-driven) attribute evaluation

An example of eager evaluation is given in Figure 2.11, with the evaluation order indicated by a circled number next to each node. The two nodes with no incoming edges, $B_{\mathrm{I}}.value$ and $L_{\mathrm{I}}.position$, are evaluated first – as nodes ① and ② – and removed from the graph together with their outgoing edges. Nodes with no incoming edges in the new graph, in this case $L_{\mathrm{II}}.position$ and $B_{\mathrm{I}}.position$, are evaluated next – as nodes ③ and ④. A single evaluation step of the algorithm can thus affect more than just one node. The order in which nodes are evaluated within this step is interchangeable, so $L_{\mathrm{I}}.position$ could conceivably be evaluated before $B_{\mathrm{I}}.value$. The first node in each step is marked with a more prominently circled number in Figure 2.11.

The advantages and disadvantages of these algorithms are discussed in more detail in the design chapter. A discussion of cycle detection during dynamic evaluation is given in Section 2.4.1.

Dynamic evaluation can be interleaved with parsing. Whenever a node is created in the parse tree, its attribute instances are added to the dependency graph. The resulting (partial) dependency graph can be subjected to preliminary evaluation in an eager fashion; all attributes with dependencies met, for which the corresponding parse tree nodes have also been constructed, can be calculated. Such techniques are discussed in more detail in the design chapter as well.

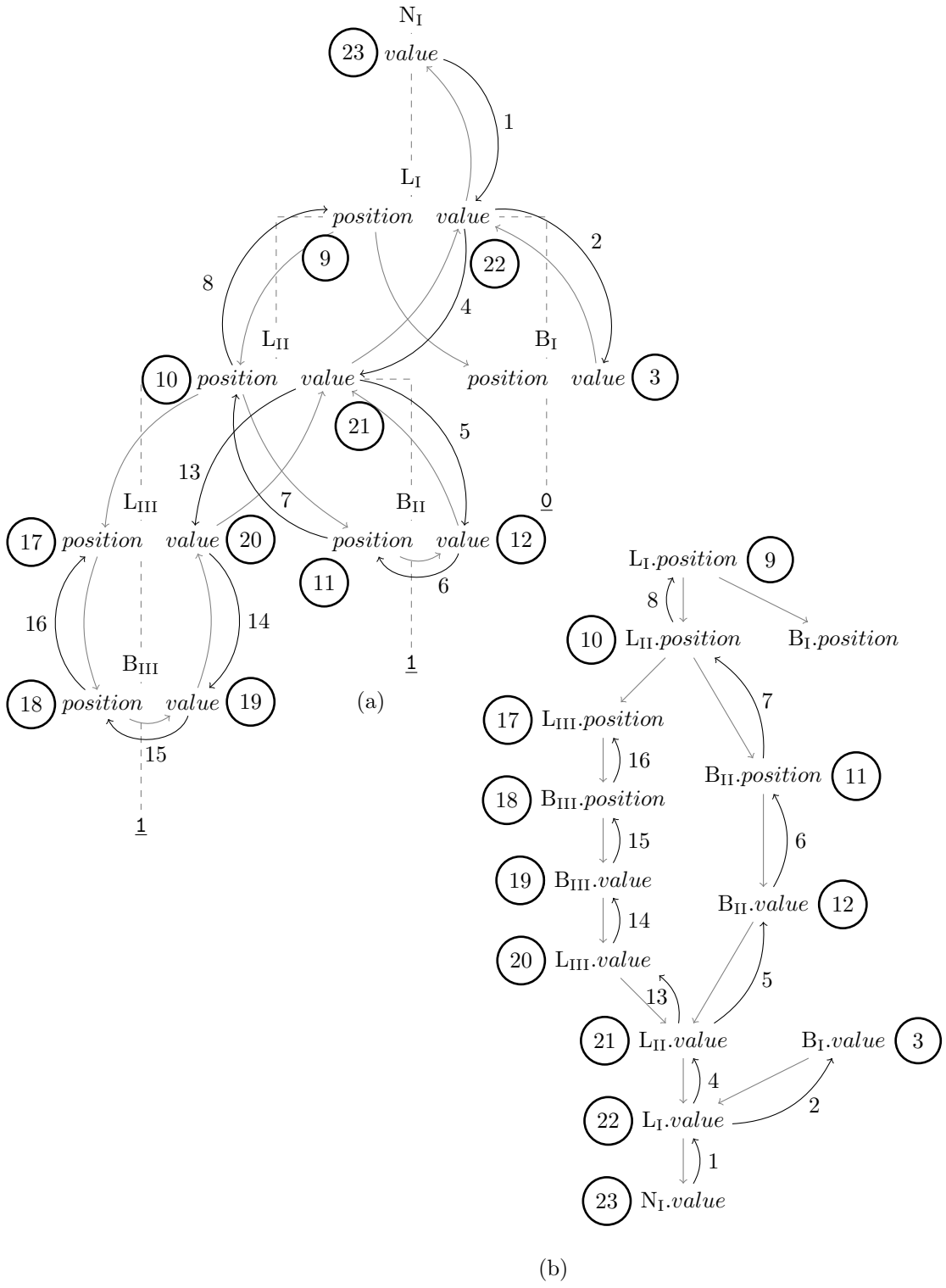


Figure 2.10: Lazy evaluation of input $\underline{1}\ \underline{1}\ \underline{0}$ for AG 2.5 shown in attributed parse tree (a) and isolated dependency graph (b).

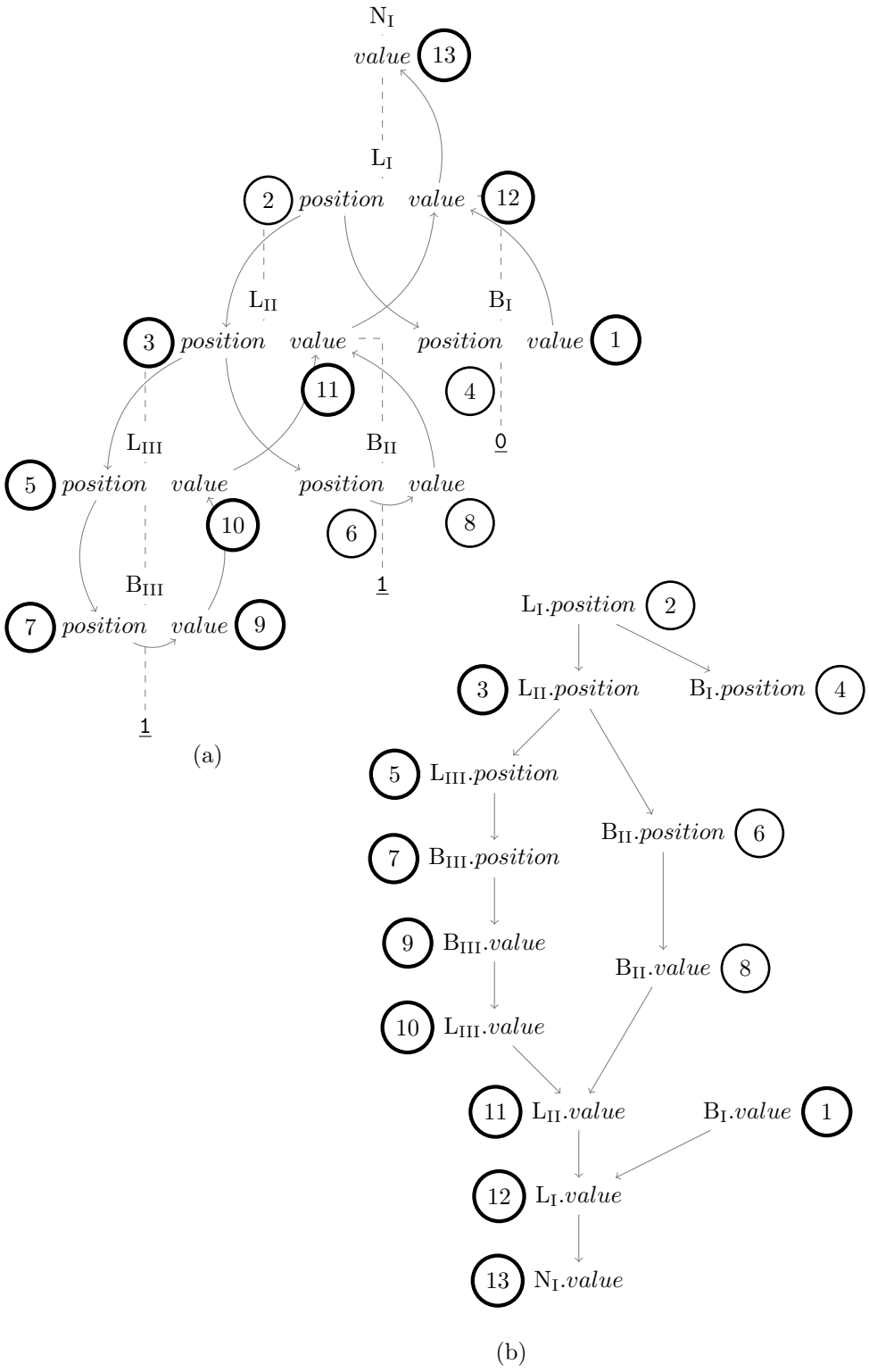


Figure 2.11: Eager evaluation of input 1 1 0 for AG 2.5 shown in attributed parse tree (a) and isolated dependency graph (b).

2.4 Cycles in attribute grammars

As an attribute grammar cannot be evaluated if it contains cycles, a circularity test is useful for providing a guarantee that the evaluation will terminate for any possible input. Originally, Knuth specified an incomplete circularity test [Knu68, p. 134], which was then corrected, the complexity of the underlying algorithm, however, increased from polynomial to exponential as a side effect [Knu90; Knu71]. Circularity tests for attribute grammars were shown to be of exponential time complexity by Jazayeri et al. [JOR75] – this was one of the first practical problems which turned out to be so difficult [JOR75; Knu90]:

> The upper and lower bounds on the time complexity of the problem are, respectively, $2^{c_1 n}$ and $2^{c_2 n/\log n}$ where c_1 and c_2 are constants and n is the length of the description of the attribute grammar. [JOR75, p. 128]

Jazayeri used attribute grammars to define a specification of a linear bounded automaton and of a writing pushdown acceptor, thus reducing cycle detection in an attribute grammar to the membership problem of these automata [3] and being able to infer the lower bound. The upper bound was given by a refinement of Knuth's circularity test based on constructing a context-free grammar from an attribute grammar such that, in the case of circularity, the resulting context-free grammar is non-empty.

It has been further shown [Wu04] that the problem is EXPTIME-complete, meaning it requires exponential time and polynomial space to solve on an alternating Turing machine. Still, an implementation of the given alternating Turing machine itself on a computer requires exponential time and space.

2.4.1 Cycle detection in dynamic evaluators

The problem of detecting cycles at runtime is more simple, as it can be performed during the dependency resolution algorithm itself, with just a few additions to the basic algorithm.

Algorithm 2.4 specifying lazy evaluation can be modified to also track a list of visited nodes V, mirroring the parameters in the call stack of *evaluate*. If evaluation reaches a node n already present in V, a cycle has been detected. The sublist of V from the last occurrence of n to the most present entry shows the encountered cycle.

Algorithm 2.5 can also detect cycles with ease; if at some point S is non-empty, but all nodes have at least one incoming edge, then one of the nodes must reference itself, either directly or indirectly, producing a cycle. There may be more cycles within the remaining graph, a standard algorithm for detecting strongly connected components such as Tarjan's algorithm [Tar72] can be used to find them.

2.5 Attribute grammars and GLR

Generalized LR (GLR) extends LR parsing with the ability to process ambiguous context-free grammars. When the parser encounters an LR conflict, it splits into two sub-processes that follow both alternatives. Sub-processes that encounter an error are discarded, and if two sub-processes reach the same state after processing the same input, they are merged back into one. The method is explained in more detail in Appendix A.1.3.

The interaction between attribute grammars and ambiguous parsers such as GLR is not discussed often in literature. While the two are technically compatible, using them in combination

[3] The membership problem denotes whether a certain input will be accepted by the automaton.

exposes a few problems. In particular, applying semantic rules in an ambiguous context can result in several different attributed parse trees being generated for one particular input. In the case of GLR, where compatible parse tree nodes can be merged back into one, it is possible for one parse tree node to have different possible sets of attributes.

A method of handling such concerns efficiently, especially in the case of GLR, is not available within the context of "ordinary" attribute grammars. Thus, an extension of the formalism is required.

Research in this area is sparse and mostly focuses on pruning erroneous parses using conditional attributes [DDPS09]. In this approach, a boolean attribute called "ok" is defined for a parse tree node; the value of the attribute determines if the node (and its corresponding parse tree) will be kept.

Bison implements the option to merge "semantic actions" (Bison's in-built synthesized attribute per node) from multiple tree derivations [FSF15a, Sec. 1.5.2]. This can be done either using precedence rules specified within the grammar or by employing a user-defined merge action. A merge action will receive all values from the potential derivations as input and calculate a new value based on them.

Furthermore, Bison provides a construct similar to conditional attributes via its "semantic predicates" [FSF15a, Sec. 1.5.4]. Such predicates are evaluated immediately when a corresponding rule is reduced in one of the GLR subprocesses; if the result is 0 (`false`), the subprocess terminates and the corresponding parse tree is discarded. At the time of writing however, semantic predicates do not have access to the value of semantic actions while the parse tree is ambiguous, so they cannot be used in conjunction with them.

This thesis proposes a set of approaches for merging attribute grammars and GLR in Section 3.3. The dynamic evaluator implemented as part of the thesis supports evaluation for all well-defined attribute grammars on unambiguous parse trees (see Section 4.2.3).

2.6 Traversals

Traversals [ALSU07, p. 56-60] allow nodes of a tree to be visited in an explicit and predictable order as specified by the *type* and *direction* of the traversal.

Due to their predictable nature, traversals are a useful complementary technique to attribute grammars, for which execution order can depend on many factors and is most safely regarded as an implementation detail. Traversals are particularly useful when performing sequential operations involving the entire parse tree, such as outputting generated code.

In the context of attribute grammars, traversals are ideally performed on the attributed parse tree once evaluation is complete. This gives traversals access to the values of all attribute instances. Multiple traversals of various types can then be executed in sequence.

Several *types* of traversals exist; two of the most common ones – postorder and preorder – are detailed below. They are variants of depth-first search and can be easily described using recursive procedures.

In a **postorder traversal**, a node is visited after its children:

```
1 function postorder(n):
2   for all children m of n:
3         postorder(m)
4
5   visit(n)
```

In a **preorder traversal**, a node is visited before its children:

```
1 function preorder(n):
2   visit(n)
3
4   for all children m of n:
5         preorder(m)
```

The *direction* in which child nodes are visited is determined by the iteration order in lines 2 and 4 respectively; commonly, the direction is either left-to-right or right-to-left.

Although they are orthogonal concepts, traversals and attribute grammars are often encountered together. An evaluator based on traversals can be generated statically for subclasses of attribute grammars such as multi-pass AGs (see Appendix A.2.1). In the absence of attribute grammar support, it is common for compiler implementations to use one or more explicit traversals over a parse tree to perform similar kinds of analyses.

2.7 Existing dynamic attribute grammar evaluators

A brief overview of existing systems that support dynamic attribute grammar evaluation is given in this section.

2.7.1 Ox

Ox is an attribute grammar evaluator toolkit written by Kurt M. Bischoff [Bis93]. It preprocesses scanner and parser definitions written for the Lex and Yacc generators, turning attribute definitions into "semantic actions" understood by the generators. In effect, Ox can be seen as a dynamic evaluator generator for well-defined grammars or, more precisely, as a program that transforms any well-defined attribute grammar into an S-attributed one, which is then evaluated using Yacc's semantic actions.

The toolkit requires for attributes to be defined as a C-like `struct` in the preamble of the Yacc parser definition. Within the parser and scanner specifications, for each terminal or production, there may be an *attribute reference section* specifying attribute evaluation rules (*attribute definitions*) as annotated C/C++ code. A *definition mode annunciator*, which is part of each attribute definition, specifies how to obtain the attribute's dependencies – they can either be given explicitly outside the C/C++ code block, "guessed" from their occurrences in the code itself or a combination thereof.

The generator implemented in the thesis has a similar feature set to that of Ox, as it has proven useful in practice. However, there are some differences, most notably the presence of GLR support and of other, more intuitive, mode annunciators; these differences are discussed in the implementation chapter (Section 4.1.2).

2.7.2 JastAdd

JastAdd [EH07] is a Java-based system for extensible compiler construction. JastAdd supports dynamic evaluation of attribute grammars in a lazy (on-demand) fashion. The toolkit expects the entire tree to be parsed before evaluation, but also offers a multitude of other features such as references, circular attributes and tree rewriting.

2.7.3 Happy

Happy[4] is a parser generator for Haskell featuring attribute grammar support. Happy leverages Haskell's functional lazy evaluation mechanism.

[4] https://www.haskell.org/happy/doc/html/sec-AttributeGrammar.html

Chapter 3

Design of the attribute evaluator

This chapter details the rationale behind implementing a dynamic evaluator. It also explains the choice of algorithm and explains potential approaches to integrating attribute evaluation with GLR.

3.1 Static versus dynamic approaches

The idea was to create a conceptually simple evaluator that could handle any well-defined attribute grammar. Dependency resolution would not be necessary until the parsing stage.

Evaluation of attribute grammars is an old problem which has been researched in great depth. However, most of this research has concentrated on static analysis of subclasses of well-defined attribute grammars, such as purely synthesized AGs, l-ordered AGs and absolutely non-circular AGs [AM91; DJL88]. The problem, as cited with respect to general approaches that handle all well-defined attribute grammars, is exponential with regard to grammar size [DJL88; JOR75]. In addition, some subclasses are interesting because it is possible to construct efficient evaluators for them based on tree-walk approaches (see Appendix A.2.1).

For **well-defined attribute grammars**, the complexity of static analysis stems from the need to analyze the dependency graph of each production in the context of all possible parse trees. The procedure is outlined in Section 2.3.1 and refers to the calculation of *IS-SET* s and *SI-SET* s and to the membership test in particular. Figure 2.8 hints at the potential complexity involved.

Absolutely non-circular grammars attempt to avoid the exponential complexity by modifying the membership test. They are a superset of most other researched subclasses of AGs [AM91, p. 110] and are discussed in Appendix A.2.1.

The example given in Appendix A.2.1 shows that even simple grammars can be well-defined but not absolutely non-circular.

l-ordered grammars are the largest class that can be statically evaluated (see Appendix A.2.1). They are a subclass of absolutely non-circular grammars and as such inherit their limitations.

In practice, many attribute grammars can be statically evaluated and do not incur significant runtime costs. As such, static approaches should not be discounted immediately due to two main advantages:

- The membership test, implying cycle detection, can be a useful tool during development since it is usually performed at evaluator construction time and is thus valid for all possible input.

- The ability to construct an efficient evaluator after analyzing the grammar can optimize use of computing resources.

Nevertheless, it is possible to create grammars that do not conform to a specific subclass, and results in practice show that it may not be obvious when this happens or how to deal with the issue [NGIHK99].

With regard to evaluating all well-defined attribute grammars, static approaches can thus be employed as optimizations of the dynamic evaluation technique. If a grammar does not pass the membership test for a specific subclass offering an optimized method of evaluator construction, it can fall back to another subclass or to dynamic evaluation.

If, however, the dynamic evaluator is fast enough in practice, the complexity of a static approach may be unwarranted. It is therefore of interest to see how a dynamic approach fares, especially given the computational resources available nowadays. Recent research suggests that dynamic evaluation approaches work well in practice [EH07; SKV10].

It is to be noted that some static analyses can be performed in linear time in the size of the grammar. This concerns verifying that attributes of a symbol are consistently used as inherited or synthesized, and that they are calculated in all productions where the symbol appears on the right or left hand side respectively. This analysis ensures the grammar is well-defined – except for cycle detection – and is useful at evaluator construction time.

3.2 The dynamic evaluator

3.2.1 Topological sorting complexity

Topological sorting, when done during the parsing stage, requires a linear amount of time with regard to the number of nodes and edges of the dependency graph. A more detailed analysis is given here:

Nodes Since the nodes represent attribute instances, their number is limited by the nodes of the parse tree times a constant factor which, in worst case, is the maximum number of attributes that a symbol has.

Edges Synthesized attributes can only depend on attributes of the same node or of a node below in the parse tree. The number of vertices belonging to a synthesized attribute node is thus limited by the maximum number of symbols in a production times the maximum number of attributes a symbol has. Inherited attributes of a node can only depend on other attributes of that node, attributes of the parent node or attributes of its siblings. Again, the limiting factor is the number of symbols in a production times the maximum number of attributes per symbol.

Asymptotically, the number of edges can be larger than the number of nodes. Given the following factors:

N ... The number of nodes in the parse tree.
P_s ... The maximum number of symbols in a production rule.
S_a ... The maximum number of attributes a symbol has.
P_a ... The maximum number of attributes in a production rule.

the worst-case complexity of topological sorting can thus be expressed as:

$$O(N \cdot S_a + N \cdot S_a \cdot \underbrace{S_a \cdot P_s}_{P_a})$$

The algorithm remains linear with regard to the size of the parse tree. The two values that play an additional role in the complexity formula (P_s and S_a) are constants given a concrete grammar. The only interesting question that remains is the relationship between the size of the input (given by the number of input tokens) and the number of nodes in the parse tree. As LR parsing requires linear time [Knu65, p. 638][Tay02], this relationship is linear as well. The space required by the algorithm is also linear with respect to the nodes and edges of the dependency graph, as nothing else but a copy of the graph is used by the algorithm.

3.2.2 Lazy and eager evaluation

Depending on the starting point of a topological sorting algorithm, the corresponding evaluator can be classified as either "eager" or "lazy". An eager evaluator begins at nodes that have no dependencies and gradually works its way to the other nodes, computing a node as soon as it becomes available. By contrast, a lazy evaluator is given a set of nodes that have to be computed; the evaluator then recursively follows the nodes' dependencies through the graph, marking the nodes in its way, until there are no more dependencies to follow.

Both evaluation methods have advantages and disadvantages. A prominent advantage of lazy evaluation is that a subset of nodes can be evaluated selectively; nodes that are not related to the subset remain untouched. Eager evaluation cannot guarantee the latter. This feature, however, is not relevant when all attributes have to be evaluated.

An advantage of the eager algorithm is its intrinsic ability to calculate attributes as soon as possible. This is of relevance, for example, when evaluating purely synthesized attribute grammars, since all attributes will be calculated by the time the parser is done, rendering an extra loop over all nodes unnecessary. Deferring attribute evaluation requires information about dependencies to be stored in memory, memory that needs not be used if it is known that all dependencies are met and evaluation can proceed immediately.

An eager algorithm was chosen for the implementation of the evaluator because it possesses the useful features outlined above and does not have any disadvantages with regard to algorithmic complexity.

3.2.3 The eager evaluation algorithm

The standard eager algorithm [Kah62] gradually refines the set of all nodes S by removing nodes with no incoming edges (dependencies) and all their outgoing edges. As such, nodes that had no other dependencies come next, the order of removal is the same as the topological sort. The algorithm can easily detect cycles – they occur when some nodes are still left in the set, but no node can be removed:

```
1  S := set of all nodes;
2  T := {};
3
4  while S contains nodes with no incoming edges:
5    for all nodes n with no incoming edges:
6      for all nodes m where the edge n → m exists:
7        remove n → m;
8      S -= n;
9      T += n;
10
11 if S is not empty:
12   error(); /* cycle detected */
13
14 /* T now contains the evaluation order */
```

The set data structure. While the basic idea of the algorithm remains the same, some operations need additional consideration in order to be performed efficiently.

A naive implementation would employ an unordered list for S. This would mean that finding all nodes with no incoming edges (line 5) would require traversing the entire list in each iteration. The worst case complexity of the two loops from lines 4 and 5 would thus be $O(n^2)$, which would occur when the last node in the list matches on every iteration of the outer loop.

Fundamentally, the two loops together cannot be more efficient than $O(n)$ since all n nodes have to be added to T and only one node can be added at a time. However, it is possible to reduce the complexity of the two loops by partitioning the nodes according to the number of incoming edges; when an outgoing edge is removed, the node it pointed to is moved one partition to the left. As a result, all nodes with no incoming edges can be found in the first partition and so the inner loop never needs to go beyond it.

A partition-based implementation. The enhanced algorithm described here uses d partitions, where d is the maximum number of dependencies an attribute can have. While P_a is a worst case value for d, the actual number can be determined while parsing the grammar specification.

```
1  for i := 0 to d:
2    S_i := all nodes with i dependencies;
3
4  T := {};
5
6  while S_0 is not empty:
7    for all nodes n in S_0:
8      for all nodes m where the edge (n → m) exists:
9        remove (n → m);
10       move m one partition to the left;
11
12     S_0 -= n;
13     T += n;
14
15 if S_1 ... S_d are not empty:
16   error(); /* cycle detected */
17
18 /* T now contains the evaluation order */
```

An issue with implementing separate partitions as sets is that nodes have to be constantly moved from one partition to another, which incurs some overhead. To mitigate this, partitions $S_1, \ldots, S_d$ can be merged into a single partition S_{NR} of nodes whose dependencies have not yet been resolved. The dependency count (*deps*) of nodes in S_{NR} becomes a property of that node, which is decremented until it reaches 0 instead of moving the node to another partition.

```
1  S_0 := {};
2  S_NR := {};
3
4  for all nodes n:
5    n.deps := dependency count of n;
6    if n.deps = 0:
7      S_0 += n;
8    else:
9      S_NR += n;
10
11 T := {};
12
13 while S_0 is not empty:
14   pick a node n from S_0:
15     for all nodes m where the edge (n → m) exists:
16       remove (n → m);
17       decrease m.deps by 1;
18       if m.deps = 0:
19         S_0 += m;
20
21     S_0 -= n;
22     T += n;
23
24 if S_NR is not empty:
25   error(); /* cycle detected */
26
27 /* T now contains the evaluation order */
```

3.2.4 Evaluation at runtime

In order to save memory, evaluation can be performed during the parsing stage, before the dependency graph is known in its entirety. Essentially, all attributes from the local dependency graph of a symbol are submitted to the evaluation routine whenever that symbol is found (reduced); if the attribute instance's dependencies have already been evaluated, that attribute instance is also evaluated immediately; otherwise, it is "enqueued" within the dependency graph structure. Assuming no circular attribute dependencies, all attribute instances within the dependency graph will have been evaluated after the start symbol is reduced. The algorithm, which is set to be called after every *reduce* operation of the parser with the obtained parse tree node as parameter, is described in Algorithm 3.1.

To concretize the implementation of the dependency graph, more precisely of the query "all nodes m where the edge $(n \to m)$ exists", a property *dependees* is added to each node (n), containing links to other nodes that depend on it. *dependees* can thus be viewed as the adjacency list of n encompassing outgoing edges.

The initialization of *n.dependees* must happen before any node m that depends on n is processed. Any such node m must belong to $DO(p)$ (see Subsection 2.2.2). This encompasses synthesized attributes of the production's left hand side symbol and inherited attributes of the

production's right hand side symbols. Since the parser's *reduce* operation traverses the tree in a bottom-up fashion (children first), all references to attributes of right hand side symbols have with certainty been initialized in a previous call to *reduce*. The initialization of (synthesized) attributes referenced by the left hand side symbol is ensured by marking them as non-calculated at the beginning of *reduce_attributes*, before any dependency analysis operation.

```
1  T := {};
2
3  /* p is the parse tree node created after reduction.
4     It contains the attribute instances of the node
5     resulting from the production's left hand side,
6     as well as links to child nodes (for symbols on
7     the right hand side).
8  */
9
10 function reduce_attributes(p):
11   S0 := {};
12
13   for all attribute instances n of p:
14     n.dependees = {};
15
16   for all attribute instances n of p:
17     n.deps = 0;
18
19     for all dependencies d of n:
20       if d not in T:
21         increase n.deps by 1;
22         d.dependees += n;
23
24     if n.deps = 0:
25       S0 += n;
26
27   while S0 is not empty:
28     pick a node n from S0:
29       for all nodes m in n.dependees:
30         decrease m.deps by 1;
31         if m.deps = 0:
32           S0 += m;
33       discard n.dependees;
34
35       S0 -= n;
36       T += n; /* (evaluate n) */
37
38 ... call reduce_attributes(p); after each reduction ...
39
40 if a node n exists with n.deps > 0:
41   error(); /* cycle detected */
42
43 /* T contains the evaluation order */
```

Algorithm 3.1: A runtime eager evaluator.

The dependency graph structure is thus reduced to a single counter *deps* per node representing the incoming edges and an adjacency list *dependees* representing outgoing edges. The number of

dependees is kept low, as they are only added if a node's dependencies have not been calculated yet. In best case (as would be, for example, with a synthesized-only grammar featuring properly ordered attributes), *dependees* would always remain empty.

On line 36, n can be evaluated instead of being added to T. The check whether d is in T on line 20 can be worked around by adding a special "evaluated" flag to the node properties (or by saving, for example, a negative value in *deps*). Maintaining the list T is thus not required at all.

3.3 Integrating attribute grammars and GLR

GLR ("Generalized LR") is an extension of LR parsing that can handle ambiguous context-free grammars (see Appendix A.1.3). In practice, GLR can be used to generate both a parse forest containing all possible parse trees or, in combination with semantic rules, to generate one or more valid parse trees. In case a single parse tree is generated, GLR can be seen as effectively increasing the expressiveness of the LR variant used.

This section offers an overview of possible ways to integrate attribute grammars with GLR and details the reasoning behind choosing a specific approach for the implementation.

3.3.1 Potential approaches

Generating an attributed parse forest

Like with other parsing techniques, the resulting parse trees can be later attributed; thus a simple (but blunt) way of extending GLR to support attribute grammars is to obtain all parse trees and subsequently perform attribute evaluation on each one. Depending on the level of ambiguity of the grammar, an exponential amount of parse trees with regard to the size of the input may have to be processed.

Using GLR's compact parse forest representation

In one of the initial papers on GLR, Tomita describes a method of generating compact parse forests by employing two optimizations: sub-tree sharing and local ambiguity packing [Tom85]. The result is a directed acyclic graph from which all possible parse trees can be derived. These optimizations are based on the equivalence in behaviour of equal parser states that are reached after processing the same input.

If attributes come into play, this equivalence no longer necessarily holds. As an example, consider AG 3.1 below describing addition of binary digits. Here, the attribute *code* is used to translate the input into reverse Polish notation.

$$
\begin{array}{lcll}
S & \to & E & \{ \quad S.code := E.code; \quad \} \\
E_0 & \to & E_1 \ \underline{+} \ E_2 & \{ \quad E_0.code := concatenate(E_1.code, E_2.code, \underline{+}); \quad \} \\
E & \to & B & \{ \quad E.code := B.code; \quad \} \\
B & \to & \underline{0} & \{ \quad B.code := \underline{0}; \quad \} \\
B & \to & \underline{1} & \{ \quad B.code := \underline{1}; \quad \}
\end{array}
\tag{3.1}
$$

For input $\underline{0} \ \underline{+} \ \underline{1} \ \underline{+} \ \underline{0}$, the GLR parser will split to pursue the ambiguity in rule $E_0 \to E_1 \ \underline{+} \ E_2$. Both parse trees are merged back and treated as one as soon as the topmost E is derived, but $E.code$ now has two possible values: $\underline{0} \ \underline{1} \ \underline{+} \ \underline{0} \ \underline{+}$ and $\underline{0} \ \underline{1} \ \underline{0} \ \underline{+} \ \underline{+}$ (see Figure 3.1). This shows that a

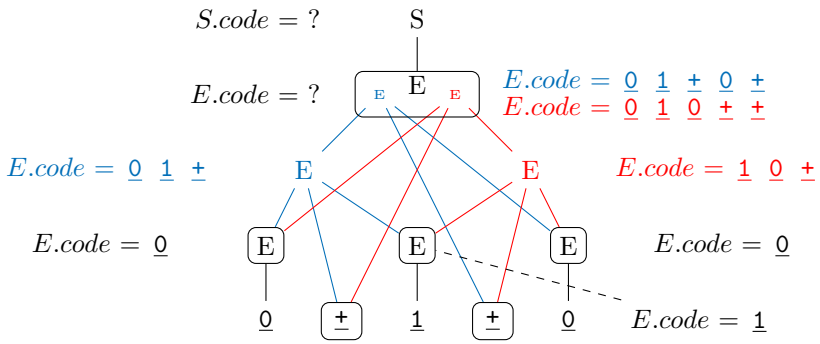


Figure 3.1: A compacted GLR parse forest for AG 3.1 and input `0 + 1 + 0`. Red and blue nodes belong to different parse trees, black nodes are shared between them.

one-to-one relationship between compacted GLR nodes and attributed nodes cannot be assumed. Note that this also affects nodes compacted via sub-tree sharing: for example, if one were to calculate the *depth* of the tree for each node labelled E by inheriting it from the topmost E onward, then the first and last bottom-most nodes labelled E in the example would each have two different possible values.

In order to handle this issue, two courses of action are possible:

- Ensuring that the equivalence between compacted nodes and attributed nodes does indeed hold by extending attribute grammars with "merge actions" that produce unique attribute instance values for nodes shared by multiple parse trees. In the example above, the merge action would simply select one of the alternatives.

  Such an approach is used, for example, for Bison's semantic values: by adding a `%merge` clause one can calculate the semantic value of a merged node based on values from the derivations leading to the merge.

  A merge action in the case of well-defined attribute grammars is needed both when calculating synthesized attributes of merged parent nodes (as is the case with Bison's semantic values) and when calculating inherited attributes of child nodes in common sub-trees (where the parent node is part of a GLR split).

  This method offers a clean mapping of attributes to GLR parser structures. However, it can be tedious and error-prone to define merge actions for all affected attributes.

- Relaxing the compacted representation and determining node equivalence based on distinctive features of attributes, be it their values or – particularly in the case of dynamic evaluation – their dependencies within the dependency graph.

  This approach retains maximum flexibility at the expense of efficiency and/or complexity. While it is conceivable that the approach can be optimized to not affect GLR parsing and to calculate only the minimum amount of attributes required, it is yet uncertain whether such expressivity is required in practice.

Generating a single attributed parse tree

A relatively straightforward approach is to generate a single valid parse tree from the GLR grammar and then perform attribute evaluation on it.

This approach is particularly useful when only one valid parse tree is actively sought for; examples can be parsing programming languages or using GLR to increase the expressiveness of an underlying LR parser. The approach entails starting with a single parse tree, pursuing multiple sub-trees in the case of a GLR split, but then merging (or picking, or discarding) parse trees into a single one during a GLR merge. In this case, attribute evaluation can be performed after parsing or while only one parse tree is present.

This method also requires the use of "merge actions" or precedence rules in order to choose the desired parse tree in face of ambiguity. However, the scope is more limited in this case and only applies to the parse tree itself, not to individual attributes. The parser may be furthermore configured to pick a random alternative by default if so desired.

3.3.2 Chosen approach and rationale

Bison already supports various mechanisms for selecting one of several possible parse trees. This includes precedence rules, semantic predicates and `%merge` clauses [FSF15a, Sec. 1.5].

Furthermore, Bison expects all ambiguities to be resolved before parsing ends. For this reason, performing attribute evaluation on a single valid parse tree is seen as the most sensible approach.

Bison's deferred semantic action mechanism can be used to construct the dependency graph and perform eager evaluation only at times when the parse is unambiguous. This saves processing time, as attributes are not evaluated by sub-parsers that will be later discarded.

Chapter 4

Implementation in Bison

As part of the thesis, an attribute evaluator was implemented on top of Bison, a popular LALR(1) and GLR parser generator. This chapter focuses on how the implementation was done, in particular with respect to the syntax additions made to support attribute grammars, to the portions of the Bison source code affected and to various architectural and user-centric decisions. The chapter also details the implementation of `libxnag`, a library encapsulating the core of the resulting dynamic evaluator. It furthermore explains how the code was tested and lists optimizations made to the evaluator during the course of the implementation.

4.1 Concepts and their syntax

Bison [FSF15a] is an open-source LALR(1) and GLR parser generator. Its syntax is compatible to that of Yacc, another LALR(1) parser generator [1], and allows specifying context-free grammars which can be compiled into C, C++ or Java parsers.

Rules consist of the name of a non-terminal on the left hand side, followed by a colon and a list of symbol names for the right hand side (which can be non-terminals or tokens). Rules end in a semicolon, vertical bars can however be used to separate alternative right hand sides.

A "semantic action" can be specified for each rule. Such actions allow for one synthesized attribute to be calculated per symbol and are easily implemented as part of the *reduce*() operation. The parser has access to all necessary data, as nothing else is needed beyond the synthesized attributes of the right hand side symbols (which are saved on the parser stack) and the actual code to be executed (saved together with the rule). No dependency problems can arise, as there are no inherited attributes; the grammar is *S-attributed* (see Appendix A.2.1).

After reading all rules, settings and their associated actions, Bison proceeds by generating a temporary data file containing the LR parsing table, all rules and action code. This data file is used in conjunction with a template called a "Bison skeleton" to generate the actual parser in the programming language of choice, optionally using a different algorithm instead of LALR. Code required for several debug and compatibility options is also handled by the skeletons.

In order to make attribute evaluation possible, certain add-ons had to be made to Bison:

%attributes. Since the languages supported by Bison are statically-typed, a listing of all attributes of a symbol, together with their corresponding types, is required. This listing is implemented as a new Bison option called `%attributes` and has the following syntax:

[1] `http://dinosaur.compilertools.net/yacc/`

`%attributes` {*attributes*} *symbols*

attributes is an annotated listing of attributes, separated by comma or semicolon, also containing their types, matching the definition of a C/C++ `struct`. This construct is detailed in Subsection 4.1.1, together with the two related options `%autosyn` and `%autoinh`.

Attribute actions and specifiers. In addition to the "semantic actions" supported by Bison itself, a rule of the grammar may have several "attribute actions", formally equivalent to "semantic rules" in an attribute grammar. Attribute actions allow the use of inherited attributes, as well as more than one synthesized attribute. Each attribute action specifies code that is to be executed to obtain the value of one or, in some special cases, several attributes.

In contrast to Bison's semantic actions, attributes defined this way remain available after the parsing stage for traversals.

Attribute specifiers are used to prefix attribute actions (or actions that are to be executed during traversal). At the same time, they allow the user to specify which attributes are being calculated, together with their dependencies. The attributes and dependencies can be entirely or partially inferred from the position of attribute references in attribute actions by using the correct specifier (`@i`, `@e`, `@m`). Alternatively, the `@d` specifier can be used to infer dependencies based on the declared types of the attributes.

A more in-depth description can be found in Subsection 4.1.2.

Traversals. While attributes themselves are sufficient to execute all the necessary operations in the semantic phase of a compiler, explicit traversals of the parse tree have also been added as a way to use calculated attribute values more effectively. Traversals are useful for debugging purposes or for such tasks as code generation, when the entire parse tree is traversed and instructions calculated using attributes are output for each node.

Traversals come into being by adding a corresponding `%traversal` option to the header of the Bison grammar specification:

`%traversal` *order direction name*

order is either `%preorder` or `%postorder`, *direction* is either `%ltr` (left-to-right) or `%rtl` (right-to-left). *order* and *direction* can be swapped. *name* is a designation for the traversal, "@*name*" can be used as an attribute specifier to run an attribute action as part of the traversal when the node corresponding to the left hand side is reached. For more details, see Section 4.1.3.

Attribute assignment in Flex. In order to allow initialization of attribute values associated with tokens, attribute support was added to Flex, a scanner generator commonly used with Bison. An attribute *a* of a token `TOK` can be referenced in a Flex action returning a `TOK` via `$TOK.a`. More details are given in Section 4.1.4.

4.1.1 %attributes

The attributes of one or more symbols can be specified using the `%attributes` option:

`%attributes` {*attributes*} *symbols*

The names and types of all *attributes* given in braces are associated with the given *symbols*. The entire contents of *attributes*, without any annotations, is translated into a C/C++ `struct` definition and represents the actual type corresponding nodes will have in the attributed parse tree. As an example, a definition of:

```
%attributes { char *name; int length; } Identifier
```

will cause the C skeleton to yield:

```
struct yyattr1 { char *name; int length; }
```

which will be used every time the attribute values of an `Identifier` are referenced.

The attribute names must be known to the system as to ensure proper usage in attribute specifiers and attribute actions. Furthermore, information not required for the C/C++ output (annotations) can be associated with attributes. Each attribute exhibits the following properties:

The attribute name. Names match the format of C identifiers – strings of alphanumeric characters including underscore, but not starting with a digit. A name is detected as the last such string before a comma or semicolon.

The attribute kind. Attributes can be declared as "inherited" or "synthesized" by adding `%inh` or `%syn` before their name, respectively. For example:

```
%attributes { %inh char *name; %syn int length; } Identifier
```

Using `%inh` and `%syn` is optional. In their absence, the information is inferred from specifiers and attribute actions in which the specific attribute occurs. However, if an attribute is erroneously used as both inherited and synthesized, being able to fall back to the declaration can yield a more useful error message.

Note that the `@d` specifier also depends on `%inh` or `%syn` being used; the specifier cannot be employed unless the declaration is present (see Section 4.1.2).

Propagation options. Attribute grammars often require information to be passed several levels up or down the parse tree, resulting in the usage of a series of "copy rules". These copy rules can be generated automatically from a template, but may cause confusion if they are not specified by the user in advance. For this reason, attributes that are to be "passed-through" in one direction or the other for a symbol or set of symbols can be annotated using a preceding `%autoinh` if inherited or `%autosyn` if synthesized. In this case, `%inh` and `%syn` are implied.

Note that propagation options for an attribute can be specified outside of `%attributes` as well, using the dedicated Bison options `%autoinh` and `%autosyn` respectively. Their syntax is:

`%autoinh` *attributes*
`%autosyn` *attributes*

attributes is a whitespace-separated list of attribute identifiers.

The dedicated options `%autoinh` and `%autosyn` only apply to `%attributes` declarations following them.

A symbol can be referenced in at most one `%attributes` declaration (this is a limitation of the current implementation). The following will result in an error, as the attributes for `Identifier` are declared more than once:

```
%attributes { int i; } Term Identifier
%attributes { char *name; } Identifier
```

4.1.2 Actions and attribute specifiers

Bison allows specifying a context-free grammar using a series of rules. Such rules consist of a left hand side (`Expression`) and a right hand side (`Term '+' Term` and `Term '-' Term`, respectively), separated by a colon:

```
Expression:  Term '+' Term ;
Expression:  Term '-' Term ;
```

Rules must end in a semicolon.

If the left hand side of several rules is the same, they can be joined together using vertical bars:

```
Expression:  Term '+' Term
           | Term '-' Term
           ;
```

Since the first syntax is less complicated, it is used more prominently further on. The distinction is handled transparently by Bison; semantic actions and attribute specifiers work in both cases.

A "standard" Bison semantic action is specified by inserting it between the last term of the right hand side and the semicolon [2]:

```
Expression:  Term '+' Term { $$ = $1 + $3; }
           | Term '-' Term { $$ = $1 - $3; }
           ;
```

`$$` references the value of the parent, `$1` and `$3` refer to the values of the first and third right hand side symbol. Symbols can be named to increase the descriptiveness of semantic actions:

```
Expression[e]:  Term[t] '+' Term[u] { $e = $t + $u; } ;
```

Attribute actions have a similar syntax. Assuming a synthesized attribute called "value" is defined for `Expression` and `Term`, an attribute action serving the same purpose as the code above would look like this:

```
Expression[e]:  Term[t] '+' Term[u]
                @i { $e.value = $t.value + $u.value; } ;
```

Attribute actions are easily recognizable as they are always preceded by an attribute specifier (in this case `@i` – explained below). The attribute action itself exhibits the use of references to attributes, such as `$t.value`. Symbols are referenced in the same way as in Bison semantic actions. The identifier after the last dot of the reference is extracted as the attribute name, meaning `$x.a.y` is parsed as `($x.a).y` – the attribute `y` of the symbol `$x.a`. The separation is made for error checking purposes and in order to infer dependencies, if requested by the user.

Attribute specifiers like `@i` above are used to determine the dependencies of the attribute action. They follow the same semantics as the "definition mode annunciators" in Ox (see Section 2.7.1) except `@d`, which Ox does not support. To keep Bison's grammar unambiguous, attribute specifiers start with an "`@`". The following specifiers exist:

[2] Bison also supports "mid-rule actions", which are evaluated after part of a rule has been parsed. They are only useful in cases when semantic attribute (and attribute actions) are unavailable. Therefore "mid-rule attribute actions" are not implemented in the attribute evaluator extension.

@i Implicit mode. This specifier lets the first encountered attribute depend on the rest. This specifier is useful for assignments using the assignment operator:

```
A: B '+' C  @i { $B.y = $A.x + $C.z; } ;
```

In this example, `B.y` depends on `A.x` and `C.z`.

@m *attributes* Mixed mode. This specifier has one or more parameters, separated by space, designating "dependees". Attributes specified as parameters depend on all other attributes encountered in the action. This is useful when more complex code is executed in the attribute action and `@i` would not have the desired effect:

```
A: B '+' C  @m B.y { int tmp = $A.x + $C.z; $B.y = tmp; } ;
```

Here, `B.y` depends on `A.x` and `C.z`.

@e *dependees***:** *dependencies* Explicit mode. This specifier requires both the dependees and the dependencies as parameters.

```
A: B '+' C
   @e B.y: A.x C.z { $B.y = $A.x; if (0) {/* use C.z */} } ;
```

Here, `B.y` depends on `A.x` and `C.z`.

@d Declaration mode. This specifier uses the attribute declaration to find out whether an attribute is synthesized or inherited and thus associate with it the role of dependency or dependee.

```
%attributes { %syn int val; } Expression Term
...
Expression: Term[t] '+' Term[u] @d {
  int tmp = $t.val; $$.val = tmp + $u.val;
} ;
```

In this example, `Expression.val` depends on `Term[t].val` and `Term[u].val`.

More attribute actions may be given for a rule, but an attribute specifier must appear every time:

```
A: B '+' C  @i { $A.x = $B.y + $C.z; }
            @m C.f {
              if ($B.g)
                $C.f = $A.h;
              else
                $C.f = -$A.h;
            } ;
```

Section 4.2 contains more information about how attributes are saved and how attribute values are actually calculated.

Traversal specifiers also start with an "`@`" and use this mechanism; they are detailed below.

4.1.3 Traversals

Traversals are used to perform operations on the nodes of a parse tree in a predefined order, after the parsing process is complete. Each node for which a traversal action is set is visited and its traversal action is executed. All attribute values are readily available in the traversal phase.

Traversals are defined using the `%traversal` option:

`%traversal` *order direction name*

Traversal order. *order* can be either `%preorder` or `%postorder` – in a preorder traversal, a node is visited before all of its children; in a postorder traversal, the children are visited first.

Traversal direction. *direction* can either be `%ltr` (left-to-right) or `%rtl` (right-to-left). This determines the order in which children are visited and works both with `%preorder` and `%postorder`.

Name. Each traversal must have a name. This name is used to distinguish traversal actions – a traversal specifier with `@name` must precede traversal actions for the traversal called `name`.

The position of *order* and *direction* may vary. An example traversal definition would be:

```
%traversal %preorder %ltr output
```

A traversal action for `output` may look like this:

```
Expression: Term '+' Term @output { print($$.value); }
```

Traversal actions and attribute actions may follow each other:

```
Expression[e]: Term[t] '+' Term[u]
               @i { $e.value = $t.value + $u.value; }
               @output { print($e.value); }
```

An actual parse tree with the forward and backward links necessary to traverse it in any direction given a node is constructed during parsing. After both parsing and attribute evaluation are complete, traversals are executed, one after another, in the order that they were defined in the `.y` file.

4.1.4 Attribute assignment in Flex

The evaluator offers the ability to define attribute actions for tokens in a corresponding scanner (`.l`) file in a way similar to Ox (see Section 2.7.1). Attribute support is enabled in the modified version of Flex by adding:

```
%option attributes
```

to the initial section of the input file or by calling `flex` with the `-A` command line flag.

The syntax is relatively straightforward and involves referencing the attribute data via a `$` followed by the token name; this can be done in a regular Flex action.

In the following example, the attribute *val* of nodes labelled `NUM` matching `0x[0-9a-fA-F]+` is set to the value of the scanned string parsed as a hexadecimal number:

```
0x[0-9a-fA-F]+ {  $NUM.val = strtol(yytext, NULL, 16);  return NUM; }
```

The evaluator runtime checks if the returned token identifier is compatible with the token that was used in the attribute assignment. Token names used in `%attributes` are exported by Bison as `#define`s prefixed with `YYA_` that reference the corresponding `%attributes` struct, so using an invalid token will result in a compiler error.

4.2 Implementation details

4.2.1 Scanners and parser

Bison uses a bootstrapping process to generate a parser for its own input file format. By default, a compiled version of the grammar for Bison's input format is included with the distribution; when the distribution is rebuilt, a previously created Bison executable is used to regenerate the Bison parser from the specification. All relevant files can be found in the `src/` directory of the distribution; `parse-gram.y` is the grammar specifying input files accepted by Bison, while `parse-gram.c` is the compiled version.

The scanners employed by Bison are also not hand-coded – they are created at build time using Flex, an open source scanner generator. The corresponding definition files have the extension `.l`, as more scanners are employed for particular purposes. The challenge met when adding attribute grammar support was to reuse as much of the existing code and tokens as possible, integrating new features "naturally" yet retaining a certain degree of independence, such that unrelated changes to the grammar in the future do not influence the functionality of the attribute grammar code.

A summary of the definition files including their use and additions made to support attribute grammars follows:

scan-gram.l is used to produce input tokens for Bison's main grammar, including identifiers, option names, comments, strings, bracketed and unbracketed code.

Tokens corresponding to the new Bison options (`%attributes`, `%traversal`, `%preorder`, `%postorder`, etc.) were added. Attribute specifier-related tokens (`@i`, `@e`, `@m`, `@d`, as well as every token encountered in the scanner's initial state starting with an "`@`") are also handled. In particular, the scanner recognizes strings that start with "`@`" and end before a "`{`" as a single token; a simple parser is then used to extract the attribute / traversal specifier and its parameters from the string.

scan-attributes.l is the scanner used to parse attribute declarations inside `%attributes` options. Beside identifying and removing comments, the scanner generates:

- A string that can be used as the body of a C/C++ `struct` definition by the skeleton. This is done by removing annotations (everything starting with a "`%`") and escaping certain characters, as required by the skeleton's macro language.
- An `attribute_decl_list` structure that maintains all attributes listed in the declaration. Attribute names are detected as C identifier tokens preceding commas or semicolons. The structure contains the name of the attribute and flags indicating whether the attribute is synthesized or inherited and whether this is done automatically.
- An `attributes_props` structure that collects all of the gathered information, including the `attribute_decl_list`, the number of attributes, the struct code string and the location of the declaration in the Bison input file. Information from this structure is transferred by the parser to each symbol referenced by the `%attributes` option.

scan-code.l handles semantic action code. This scanner was enhanced to also understand attribute actions and traversal actions; this was done by introducing a new Flex state which handles references containing a dot separately. The function responsible for performing the actual work, `handle_action_attribute`, checks whether the attribute name is valid for the referenced left or right hand side symbol and if that is the case, translates it into

a `b4_lhs_attribute` or `b4_rhs_attribute` macro, respectively, to be turned into special programming language code later during the skeleton stage. It generates an error message if the referenced item or attribute is unknown or cannot be used.

All attributes encountered in the code are saved in a list to be used for determining dependencies, together with the attribute specifiers.

scan-skel.l is a scanner that is used to interpret special directives and escape codes output by the instantiation of a skeleton. No changes were made to this file.

In order to make deployment of the attribute-enabled Bison version easier, changes to the Bison grammar had to be compatible with Bison versions without attribute support. As such, new features are implemented using a combination of Bison's semantic actions and global variables:

Options Bison options relevant to attribute grammars are implemented and handled like any other option. Their tokens are defined at the beginning of the grammar and used in a `prologue_declaration`. The non-terminal `attribute_declaration` is used for attribute blocks, a semantic action that outputs the structure and attribute declarations needed by the skeleton is present here. The action also updates the `sym_content` of referenced symbols with structure and attribute information via a call to `symbol_attribute_data_set`.

Traversal specifications for the skeleton are output when a `%traversal` is encountered.

Semantic actions The definition of a RHS in Bison is recursive – a list of RHS symbols and actions is created this way. The relevant line for attribute actions and traversal actions is:

```
rhs: rhs attribute_specifier "{...}" ;
```

Thus, one or more attribute actions for a RHS can be given. Each such action must begin with an `attribute_specifier`. A specialized function called `grammar_current_rule_attribute_append`, detailed later on, is called to parse the attribute specifier and add the action to the structure belonging to the current rule, which is managed as a global/member variable by Bison.

Attribute specifiers are, too, saved in global variables during parsing and used/cleaned up immediately when the corresponding `rhs` rule is reduced. `current_specifier_type` determines how dependencies and dependees of the attribute action will be calculated (based on the specifier token), while `current_specifier_option_type` together with `current_specifier_options` (which is handled by the `gramar_current_specifier_*` functions) manage a string or list of strings which hold the traversal name or the dependencies of an `@e` or `@m` specifier respectively.

4.2.2 Attribute actions

The function `grammar_current_rule_attribute_append` is called each time a rule corresponding to an attribute/traversal is reduced. This function creates a new entry in the (user input) rule's `attribute_props_list` structure containing the specifier, the rule type and preallocated space for data that is filled in when `scan-code.l`'s scanner is called. This is done after all input has been parsed.

The function `code_props_translate_attribute_code` fills in the aforementioned data. It is called in `reader.c` by `translate_attributes_code`, which is one of the operations performed on the "symbol list" representation of the grammar. `code_props_translate_attribute_code` generates both properly escaped code for use by the skeleton and the list of attribute references found in the code. The list of attribute references is then used along with the specifier to

obtain a concrete list of dependencies that can be processed directly by the topological sorting algorithm. Checks that ensure the user gave a definition of all attributes are also executed during `translate_attributes_code`:

- The start symbol must not have any attributes declared as inherited or auto-inherited, as there is nowhere to inherit them from.
- Each attribute of a symbol must be used consistently as either inherited or synthesized in all rules of the grammar in which the symbol is present. If an attribute is explicitly declared as inherited or synthesized, this must be reflected in its usage. Attributes declared as `%autoinh` or `%autosyn` are assumed to be inherited and synthesized, respectively.
- For each rule of the grammar, all synthesized attributes of the left hand side symbol and all inherited attributes of all the right hand side symbols must be defined (appear as "dependees") in exactly one attribute action. Inherited attributes of the left hand side symbol and synthesized attributes of the right hand side symbols may not be defined in any of the rule's attribute actions. For attributes that have not been explicitly declared as inherited or synthesized, the attribute kind is inferred from the presence of either zero or one attribute actions in which they are defined. This is cross-referenced with the inferred attribute kind from other rules featuring the same attribute; any conflicts result in an error.

Error handling

Error reporting is an important usability aspect of any software, in particular that of libraries and tools. Thus, particular attention is given to the context in which error messages are output and to the information reported to the user. Any conflicts or inconsistencies in the attribute grammar are reported. The goal behind the error messages is to pinpoint inconsistencies in an understandable way.

Error handling and reporting is done in a single pass over the grammar. The main functions that perform error handling are called `check_unused_attributes_in_rule` and `symbol_attribute_usage_set`. The following error conditions will be detected:

- Redefinitions of the same attribute in the same rule.
- Missing or conflicting definitions in all affected rules if the attribute was explicitly declared as inherited or synthesized. For example, given the following Bison grammar that features a missing declaration of `B.y` in the rule `B: '1'`:

  ```
  %attributes { %syn int y; } B
  %%
  N: B;
  B: '0' @i { $B.y = 0; };
  B: '1';
  ```

  an error message such as this will be output:

  ```
  syn-missing.y:5.4-6: error: synthesized attribute ‘y’ of symbol
  'B' is undefined here
   B: '1';
      ^^^
  ```

Similarly, if the synthesized attribute features in a rule where it would be inherited:

```
%attributes { %syn int y; } B
%%
N: B @i { $B.y = -1; };
B: '0' @i { $B.y = 0; };
```

an error message such as this follows:

```
inh-conflict.y:3.9-22: error: synthesized attribute ‘y’ of symbol
‘B’ inherited here
 N: B @i { $B.y = -1; };
         ^^^^^^^^^^^^^^
```

- Rules with a `@d` specifier in which no attribute is calculated and all attributes are interpreted as dependencies due to their types. In order to elicit a more helpful response in the case of user error – such as having used the wrong attribute type in an attribute definition or having placed the rule at an incorrect position – such rules are not allowed. For example, the specification:

```
%attributes { %syn int y; } B
%%
N: B @d { $B.y = 1; };
B: '0' @d { $B.y = 0; };
```

yields an error message such as this:

```
no-parents.y:3.9-33: error: none of the attributes have types (inherited or
synthesized) that suggest they could be calculated here
 N: B @d { $B.y = 1; };
         ^^^^^^^^^^^^^
```

- Mismatches between the inferred type of an attribute from one rule and the way it is used in another rule, if the attribute was not explicitly declared as inherited or synthesized. Error messages emphasize the mismatch itself, but assume a rule in which an attribute is used to more likely represent the attribute type:

```
%attributes { int y; } B
%%
N: B;
B: '0' @i { $B.y = 0; };
B: '1';
```

will yield:

```
rhs-missing.y:4.11-23: error: missing definition: attribute
‘y’ of symbol ’B’ is synthesized here,
 B: ’0’ @i { $B.y = 0; };
           ^^^^^^^^^^^^^
rhs-missing.y:5.4-6: error: but not synthesized in this rule
 B: ’1’;
    ^^^
```

An equally valid interpretation of the grammar would be that the attribute `B.y` is inherited, that the declaration in `N: B` is missing and the declaration in `B: ’0’` is superfluous. This would mean that the rule `B: ’1’` is correct.

- Note that in order to obtain relevant error messages when an attribute is used incorrectly, it is recommended to declare attributes as inherited or synthesized in the `%attributes` definition.

  As a further example, given the grammar:

  ```
  %attributes { int y; } B

  N: B;
  B: ’0’;
  B: ’1’;
  ```

  it is unclear whether `B.y` is inherited or synthesized. As such, the error handler will first assume that the attribute is synthesized (since it is undefined for `N: B`), then report a conflict when `B: ’0’` and `B: ’1’` are processed.

  Declaring the attribute as inherited would yield a single and more descriptive error, namely that the inherited attribute `B.y` is undefined in the rule `N: B`.

4.2.3 The skeleton

The `%attribute` structures and their size, all attribute/traversal action code, identifiers of attributes defined by tokens and their dependencies are saved as key-value pairs into a table, the contents of which is used as input data together with the skeleton to obtain the actual parser using the macro language `M4`.

Multiple skeletons exist for various combinations of output programming language and parsing algorithm. Attribute evaluation was implemented for the C language skeleton in the standard (LALR) and GLR variants.

A detailed description of how this is accomplished is given below.

yacc.c

`yacc.c` is the skeleton file for C language LALR parsers. It contains additional variable definitions and macro calls that enable attribute evaluation.

- Two additional variables – `yynval` and `yylnval` – hold the parse tree nodes belonging to the current symbol and lookahead symbol respectively, in a manner similar to `yyval` and `yylval` (which are used by Bison to hold “semantic values”). Furthermore, the parse tree node stack is maintained in variables prefixed with `yyns`, analogously to `yyvs` for “semantic values”.

- Calls to the macros `b4_node_init_tok` and `b4_node_cleanup_tok` are used to initialize parse tree nodes for tokens.
- Calls to the macro `b4_node_init` are used to initialize parse tree nodes during a reduce operation. For each attribute action defined in the scope of the reduced rule, a call to `b4_node_dependency` is emitted, which adds the necessary dependency initialization code for that action.
- A call to the macro `b4_evaluate_ready_set` is performed after initializing the above. The macro emits code that will evaluate attribute instances with no remaining dependencies.
- `b4_support_structs` outputs C `struct` declarations for the data structures used internally by the evaluator.
- `b4_support_header` outputs variable declarations that need to be present in the generated C headers.
- `b4_support_functions` outputs function definitions used internally by the attribute evaluator and by traversal code.
- `b4_attr_yyattrs` outputs code to be executed after parsing has been completed, which includes the circularity check and traversal execution.

bison.m4

`bison.m4` contains language-independent `M4` macros. The macro for checking if attribute support is enabled (`b4_attr_if`), as well as macros that output user code for attribute actions ((`b4_attribute_actions`) and traversal actions (`b4_trav_actions`) are defined here.

glr.c

`glr.c` is the skeleton file for C GLR parsers; this file is also used by the C++ GLR skeleton.

Like `yacc.c`, it contains variable definitions and macro calls that add support for attribute grammars inside the parser.

- `struct yyGLRState`, which embodies a GLR state, is extended to hold a pointer to `yynval`, an attributed parse tree node.
- The set of attribute instances ready for evaluation, `yyai_ready_list`, is added to `struct yyGLRStack`.
- Since the GLR parser uses different variable names and data structures than the LALR parser, access to the current parse tree node and its children is performed differently. Macros that abstract away this aspect (`b4_lhs_attribute_node`, `b4_rhs_attribute_node` and `b4_root_attribute_node`) are defined for this purpose.
- Several functions declarations, such as `yyglrShift`, `yydoAction`, `yyresolveAction` and `yyuserAction` are amended to accept a pointer to the active `yynval`. With this, parse tree nodes can be initialized when shifting tokens or evaluating (deferred) actions.
- `yyuserAction` is extended with the code necessary for node initialization and attribute evaluation.
- The `yyparse` function is modified to execute all traversals after successful parsing.

Code related to attribute evaluation is executed when a deferred action is resolved. This is generally the case when the parse is not – or no longer – ambiguous. Only the attributes of the accepted parse tree are evaluated. An exception is the evaluation of a user-defined GLR merge action, in which case both possible parse trees are generated internally.

c-attr.m4

`c-attr.m4` isolates the evaluator-specific C code from `yacc.c` and `glr.c` (except for some variable definitions) in order to increase legibility and support code re-use. It also contains definitions of other macros that are used throughout the code, including in attribute/traversal actions and when outputting `%attribute` structures. The most important macro definitions are listed below.

- `b4_attribute_actions` outputs all attribute actions of the generated parser as a list of C `case` statements. Each attribute action is assigned a unique identifier.
- Analogously, `b4_trav_actions` outputs all actions associated with traversals as C `case` statements, with each traversal action being assigned a unique identifier.
- `b4_node_init` is used when a new parse tree node is generated. It outputs code to allocate memory for the node and link all child nodes.
- `b4_node_init_tok` and `b4_node_cleanup_tok` are used when a new pare tree node is generated for a token; such nodes can be handled more efficiently as they do not have children or attribute dependencies, and in some cases may be completely superfluous.
- `b4_node_dependency` is expanded once for each attribute action in a rule, after the macro `b4_node_init`. The attribute's dependencies within the context of the rule are passed to the macro as parameters. The code adds the attribute instance to the dependency graph and enqueues it for evaluation if possible.
- `b4_evaluate_ready_set` iteratively evaluates actions of all attribute instances that have no unresolved dependencies.
- `b4_lhs_attribute` and `b4_rhs_attribute` expand to code that accesses the data of the given attribute; calls to these macros are inserted into the attribute action code by `code_props_translate_attribute_code`.
- `b4_copy_rule` generates a copy rule for a specific attribute instance; such rules are added for automatically inherited / synthesized attributes in the absence of user-specified rules.
- `b4_attr_struct` is used to output a `%attributes` declaration as a C/C++ `struct`.
- `b4_attr_struct_alias` is used to make the type name of a token's `%attributes struct` available via `YYA_<token name>`. Flex can use that alias when outputting an attribute action without requiring knowledge of the `structs`' internal numbering.
- `b4_attr_support_structs` outputs all data structures and functions used internally by the evaluator, the most important of which are described below.

The attribute evaluator uses several data structures (C `structs`) internally.
The parse tree is saved in a recursive data structure of type `YYNTYPE`.

```
1 typedef struct YYNTYPE
2 {
3   struct YYNTYPE **children;
4   int num_children;
5
6   struct YYNTYPE *parent;
7   int idx_in_parent;
8
9   int trav_actions@{]b4_trav_count[@};
10
11  struct YYAITYPE *attr_instances;
12  int num_attr_instances;
13
14  void *attrs;
15 } *YYNTYPE;
```

This data structure contains the following:

- Pointers to child nodes (`struct YYNTYPE **children`) and the number of children (`int num_children`). These pointers are used to look up attribute instances and to execute traversals.
- Pointers to the parent node (`struct YYNTYPE *parent`), as well as the index within the parent (`int idx_in_parent`). These values are used to traverse the tree without the need for a stack.
- The action (code) to be executed during each traversal (`int trav_actions[]`).
- An array of attribute instance nodes associated with the current parse tree node (`struct YYAITYPE *attr_instances`) and their count (`int num_attr_instances`). While the attribute instance nodes are in principle independent of the parse tree, they are stored here for retrieval dependency resolution purposes and for simplicity.
- `void *attrs`, which points to the actual data of the attribute instances at runtime. This is cast to one of the `struct yyattr*` data types on use, as detailed in Section 4.1.1.

Attribute instance nodes are saved in a recursive data structure of type `YYAITYPE`:

```
1 typedef struct YYAITYPE
2 {
3   int action;
4   struct YYNTYPE *context;
5   int deps;
6   struct YYAILIST *dependees;
7 } *YYAITYPE;
```

The data structure is based on the requirements of Algorithm 3.1 and contains:

- The action (code) to be executed when the attribute instance can be evaluated (`int action`).
- The context in which the attribute instance is to be evaluated (`struct *YYNTYPE context`). The code linked to by `action` thus references either `context->attrs` or one of `context->children[...]->attrs`.
- The number of currently unsolved dependencies of the attribute instance (`int deps`), or a negative value if the attribute instance has been calculated.

- Pointers to all attribute instance nodes that depend on the current object. Once the object has been evaluated, the `deps` field of all `dependees` is decremented and any dependees with no unresolved dependencies are enqueued for evaluation.

`YYAILIST` is a single-linked list used for holding both the dependees of an attribute instance and the set of attribute instances with no unresolved dependencies (S_0).

```
1 typedef struct YYAILIST
2 {
3   struct YYAITYPE *data;
4   struct YYAILIST *next;
5 } *YYAILIST;
```

4.2.4 The algorithm

For the practical purposes of the implementation, several steps – building the dependency graph, performing the actual topological sort and evaluating the semantic rules – become intermingled.

In an LALR parser, nodes will be added to the parse tree either when a token is encountered, producing a leaf, or during a *reduce* operation, producing a node for the left hand side and linking the corresponding child nodes to it.

The basic workings of the algorithm are detailed in Section 3.2.4 (Algorithm 3.1). As mentioned in the text, the set T is not explicitly needed and is thus not included in the code.

Elements of the algorithm are transformed into code as follows:

- The set S_0 of attributes with no incoming edges is implemented by `YYAILIST yyai_ready_list`. `yyai_ready_list`, being a single linked list, is usually accessed as a stack. This does not add any unnecessary complexity, as all operations on S_0 can be performed in constant time on `yyai_ready_list` by accessing the first element of the list.
- The function `reduce_attributes(p)` is split into two parts; initialization and generation of the dependency graph structure (lines 13 – 25) is performed in the macros `b4_node_init` and `b4_node_dependency` and expanded for each possible rule of the grammar in part. Evaluation of instances in S_0 (lines 27 – 37) is implemented within the macro `b4_evaluate_ready_set`, which is expanded after the user actions' switch/case statement.
- The check on line 20 whether an attribute instance has been evaluated (so it doesn't have to be added as a dependency) is done by comparing the value of its `deps` field with the constant `YYAI_DEPS_CALCULATED`; this marker is set immediately after the attribute action is run.

The algorithm, as implemented in code, contains some additional complexity as it performs special handling for attribute actions that generate more than one attribute. In such a case, one attribute instance is designated as parent and is calculated by performing the action after its dependencies have been resolved; the others are treated as separate nodes in the graph with a single dependency on the parent and with an empty attribute action.

4.3 Implementation of the evaluator core as a separate library

The thesis aims to determine whether it is beneficial to implement the core of the evaluator as a separate library.

A C library (`libxnag` [3]) was developed in order to shed light on this aspect. The use of the library for providing dynamic attribute evaluation in Bison has both advantages and disadvantages; these are discussed below following an overview of the library's implementation.

4.3.1 Implementation overview

The main tasks to handle while performing dynamic evaluation are:

- Initializing the parse tree structure.
- Handling operations on the LR stack (shift and reduce).
- Building the dependency graph structure.
- Performing dependency resolution.
- Running attribute actions in the right context.
- Running traversals.

Initialization

Before parsing can begin, the data structure that maintains the parse tree, dependency graph and evaluator state must be initialized.

In the Bison implementation, this is handled by using the `yyns` stack and `yyai_ready_list`, respectively.

In the library, this task is encapsulated into a function that allocates the memory required and returns a pointer to the datastructure (`XNAG`).

```
1 /* Initialize the XNAG data structure. Used the passed function to
2 evaluate attributes (function is passed action ID and context node).
3 Also pass number of traversals used. */
4 XNAG nag_init(
5     void (*action_function)(int, XNAG_Node),
6     unsigned int num_travs);
```

Handling the parse tree structure

When a *shift* is performed, a parse tree node must be created and pushed onto the stack. When performing a *reduce* operation, a new node must be created and, based on the reduced rule, a specific number of symbols at the top of the parse tree node stack must be removed and linked as the new node's children. Finally, the new node can be pushed onto the stack.

Creation of the node ideally involves allocating memory for attribute data based on the node's label.

Maintaining the stack and the nodes can either be done in the library or in the generated parser directly. In the Bison implementation, these actions are emitted using the macros `b4_node_init` and `b4_node_init_tok`. An initial version of the library required an initialization step declaring all symbols of the grammar, including sizes of their attribute data structures, evaluation functions and dependencies of attribute actions. A *shift* or *reduce* would thus be followed

[3] Note that while the abbreviations "nag" and "XNAG" are used throughout the code, this library is not related in any way to the NAG numerical library (`https://www.nag.com`).

with a single call to the library's evaluation routine with the name of the new symbol. This would perform node and dependency graph initialization as well as eager attribute evaluation.

The actual, revised version also maintains the underlying stack, nodes and dependency graph itself; however, operations on them are performed explicitly using separate function calls. This has both the benefit of simplifying the library (no need to maintain the rule data structure at runtime) and of making the evaluator abstraction easier to understand.

The first function allows initialization of a node with children directly on the stack.

```
1 /* Push a new node with the given attributes onto the stack.
2    Pop its children from the stack and link them. Allocate
3    memory if needed. */
4 XNAG_Node nag_node_push(
5     XNAG nag, unsigned int children,
6     unsigned int num_attributes, unsigned int data_size);
```

Another set of functions is available for the token version; an `XNAG_Node` is allocated outside of the library, then initialized via reference and copied to the stack when needed.

```
1 /* Initialize an XNAG node that is not linked to a stack.
2    Allocate a node yourself and pass it, the call will modify it. */
3 void nag_node_init_unlinked(XNAG_Node node);
4
5 /* Push a node initialized outside of the XNAG stack onto the stack.
6    This allocates memory for the copy on the stack.
7    You have to free the original node yourself. */
8 void nag_node_push_unlinked(XNAG nag, XNAG_Node node);
```

Building the dependency graph

As shown in Algorithm 3.1, a *reduce* operation triggers the creation of new dependency graph nodes and edges based on the attributes that can be enqueued in that step.

Dependencies are part of the input required by the library. After a node has been pushed onto the stack, attribute instances can be added to it. Each attribute instance has an action associated with it, as well as a series of dependencies.

In the Bison implementation, this is handled by the `b4_node_dependency` macro.

```
1 /* Add an attribute instance to evaluate to the given node */
2 XNAG_AttributeInstance nag_instance_add(
3     XNAG_Node node, unsigned int symbol_index,
4     unsigned int attribute_index,
5     int action);
6
7 /* Add a dependency to parent_instance, i.e. parent_instance
8    can only be calculated when all its dependencies added this
9    way are evaluated. */
10 void nag_dependency_add(
11     XNAG_Node node, XNAG_AttributeInstance parent_instance,
12     unsigned int symbol_index, unsigned int attribute_index);
13
14 /* Call after all dependencies have been added
15    with nag_dependency_add. */
16 void nag_dependencies_finish(
17         XNAG nag, XNAG_AttributeInstance instance);
```

The library also supports attribute actions with more than one attribute generated from them. This can be handled by calling a separate function after declaring all dependencies:

```
1 /* Add a second attribute instance that will automatically be marked
2    as calculated along with the original instance. */
3 XNAG_AttributeInstance nag_computed_add(
4     XNAG_Node node,
5     unsigned int new_instance_symbol_index,
6     unsigned int new_instance_attribute_index,
7     unsigned int original_instance_symbol_index,
8     unsigned int original_instance_attribute_index);
```

Dependency resolution, running attribute actions

Eager dependency resolution involves maintaining a set of attribute instances with no unresolved dependencies, evaluating them and then possibly obtaining a new set of instances without unresolved dependencies.

Adding instances to this set can be done either when the dependencies of an instance have been completely defined or during evaluation.

In the Bison implementation, this is split among `b4_node_dependency` and a distinct call to `b4_evaluate_ready_set` after all attribute instances of a rule have been initialized.

The library follows a similar approach, with the call to `nag_dependencies_finish` potentially enqueueing a particular attribute instance and `nag_evaluate` performing the iterative evaluation and enqueueing of any newly resolved instances.

```
1 /* Evaluate the set of ready attributes. */
2 void nag_evaluate(XNAG nag);
```

In Bison, evaluation code is inlined. When using the library, the `action_function` passed during initialization is called with the identifier of the action to evaluate and the corresponding parse tree node.

Running traversals

After performing attribute evaluation, the library offers the option to run traversals on the attributed parse tree (see Section 4.1.3).

In the Bison implementation, traversal actions are saved along with each parse tree node using a call to `b4_node_traversal`. A set of functions to execute traversals of various types is available; these are called after parsing is complete.

Similarly, the library offers the ability to define traversal actions for each node, for a predefined number of traversals. The total number of traversals is passed to `nag_init`, and each traversal action is defined using `nag_node_traversal_add`:

```
1 /* Associate the traversal with the given index,
2    within the specified node, with the given action. */
3 void nag_node_traversal_add(XNAG_Node node, unsigned int trav_index,
      int action);
```

After evaluation is complete, each traversal can be called in part using `nag_traverse`:

```
1 /* Perform a traversal on all nodes starting with root. Calls the
      action_function for each node, passing the action for the given
      traversal set with the node and the evaluation context.
2    Supports left-to-right, right-to-left, postorder, preorder
```

```
3     (set corresponding XNAG_ORDER | XNAG_DIRECTION flags). */
4 void nag_traverse(
5     XNAG_Node root, int traversal_index,
6     void (*action_function)(int, XNAG_Node), int flags);
```

4.3.2 Integration in Bison

The library can be integrated in the Bison skeletons by initializing the `XNAG` data structure when parsing begins, modifying the macros `b4_node_init`, `b4_node_init_tok`, `b4_node_dependency` and `b4_evaluate_ready_set` to output function calls instead of code directly, and by outputting separate functions containing attribute actions and traversal actions instead of inlining them.

As the macros and changes to the skeleton are still required (albeit less code is output), Bison integration remains largely unchanged and does not become less complex except for the encapsulation provided by library functions. Stack and parse tree memory handling – for example – is done by the library, which largely removes this concern from the Bison skeleton. However, this also means that the Bison-internal memory allocation mechanisms (including the use of `YYMALLOC`, `yyoverflow` etc.) are not employed.

Without the library, generalization in Bison can be solved at skeleton level, with `c-attr.m4` containing macros for these purposes that can be reused in more skeletons.

Additionally, integrating the library adds an external dependency to the generated parser. This is unusual, as Bison parsers are generally self-contained, with the entirety of code required for compilation being generated from the skeleton. An approach that keeps the "attribute evaluation library" inside the skeleton in an isolated fashion would, of course, be possible. This is, however, not much different from a direct integration.

As developed, the library does, nevertheless, allow other applications to integrate dependency resolution and attribute evaluation functionality more easily. It also makes it possible to tackle other concerns such as modification of the evaluator itself, or implementations of logging or performance measurements, without modifying the Bison skeleton.

4.4 Testing

The evaluator is supplemented by a number of machine-executable tests. A test consists of a Bison input file (possibly accompanied by a Flex `.l` file and other support code) together with a set of testcases defining input and expected output of the resulting parser.

The tests fall into various categories and include:

- Specialized tests for errors – such as when an an attribute is undefined, or an attribute is being used as both inherited and synthesized.
- Attribute grammar assignment submissions from the Compiler Construction course at the Vienna University of Technology. These grammars define simple programming languages and exhibit a variety of features, most notably the use of both inherited and synthesized attributes, of various attribute specifiers and of traversals. As the grammars are written in Ox syntax, they are first converted into a format the evaluator can understand. Grammars that happen to depend on Ox-specific behaviour – such as an implicit evaluation order – or on features not implemented by the evaluator (macros or actions executed in reverse traversal order) are discarded.
- Performance tests as detailed in Chapter 5. The main purpose of these tests is to measure the evaluator's behaviour for input of various lengths. The tests, however, also generate

specific output that is verified as part of the benchmark run. This is particularly useful for testing large input and any code triggered by it, such as Bison's stack memory reallocation mechanism.

4.5 Optimizations

A few optimizations were made to the evaluator during its development. Implementation started with a version based on "lazy" evaluation that was not pursued further; the version, as implemented, uses sub-optimal data structures. The new version (described in the other sections) switches evaluation mode to "eager" and also addresses the choice of data structures. A brief overview is given below.

4.5.1 First version – "lazy" evaluator

The first implemented version performs topological sorting based on depth-first search. It maintains a "result" list containing the ordered elements and a stack containing the elements to be processed. Attribute evaluation is performed after all elements have been sorted.

Both the stack and the "result" list are implemented using single linked lists.

While the stack is not empty, the first element is read from the stack. If all of the element's dependencies (which are stored in a dependency list inside the element) are in the "result" list already, the element is removed from the stack and placed onto the "result" list as well. Otherwise, the element remains on the stack and children of the element that have not been calculated yet are also placed on the stack.

Determining whether an element is in the "result" list requires linear time.

This version is slow and does not exhibit linear runtime mainly due to the use of linked lists for searching. This, however, is an implementation detail and does not stem from any inherent complexity.

This version was not developed further because its main advantage, that of not calculating attributes unless explicitly required, was not put to use. Other advantages offered by the "eager" evaluator described below were more compelling, in particular the ability to interleave evaluation and parsing.

4.5.2 Second version – "eager" evaluator

The second version implements a variant of Kahn's algorithm [Kah62] as presented in Algorithm 3.1. Each attribute instance element maintains a list of attribute instances that depend on it and an unresolved dependency counter. After an attribute instance is evaluated, the unresolved dependency counter of all attribute instances depending on it is decreased by one; if the counter reaches zero for a particular attribute instance, it is enqueued for evaluation.

A worklist (implemented as a single linked list) is used to keep track of attribute instances with no unresolved dependencies.

Chapter 5

Benchmarks

This section explains the methods used to measure the performance of the evaluator and gives an overview of the results.

5.1 Benchmark code

In order to test the attribute evaluator under multiple conditions, a set of three different benchmarks is used. The first two benchmarks are meant to analyze simple behaviours of the evaluator, while the third aims to emulate a practical example.

5.1.1 Bincount

The first benchmark is based on a simple grammar with two synthesized attributes. The generated parser counts the number of ones and zeroes present in a given input and then outputs these values.

```
%{
#include <stdio.h>
#include <stdlib.h>
%}

%attributes {
    %autosyn int zeroes;
    %autosyn int ones;
} b;

%traversal out

%%

start: b @out {
    printf("zeroes: \%d ones: \%d\n", $b.zeroes, $b.ones);
};

b: b '0' @d { $$.zeroes = $1.zeroes + 1; };
```

```
b: b '1' @d { $$.ones = $1.ones + 1; };
b: /* nothing */ @d { $$.zeroes = $$.ones = 0; };

%%

#include "../charlexer.c"
```

The parser is run with strings of various lengths as input. Given that this grammar is S-attributed, it can be theoretically processed in a single pass during LR parsing. Complex dependency resolution is not necessary.

5.1.2 Multipass

The second benchmark uses attribute grammars that, if they were evaluated by performing passes over the attributed parse tree, would require a specific (fixed) amount of passes. The benchmark itself consists of a number of attribute grammars constructed using the same technique but requiring an increasing number of implied passes.

An example attribute grammar that can be parsed using two passes of the parse tree is given below:

```
%{
#include <stdio.h>
#include <stdlib.h>
%}

%autosyn attr_1s
%autoinh attr_1i
%autosyn attr_2s
%autoinh attr_2i


%attributes {
    long attr_1i, attr_1s, attr_2i, attr_2s;
} A;

%traversal out

%%

S: A
@i { $A.attr_2i = $A.attr_1s; }
@i { $A.attr_1i = 1; }
@out { printf("%ld\n", $A.attr_2s); }
;
A: A '1' A '1'
@i { $$.attr_1s = $1.attr_1s + $3.attr_1s; }
@i { $$.attr_2s = $1.attr_2s + $3.attr_2s; }
;
A: '0'
```

```
@i { $A.attr_1s = $A.attr_1i; }
@i { $A.attr_2s = $A.attr_2i; }
;

%%

#include "../charlexer.c"
```

Each "pass" uses a synthesized and an inherited attribute – for example `attr_1s` and `attr_1i` in the first pass (see Figure 5.1). The inherited attribute (`attr_1i`) is passed down from the root of the parse tree down to all nodes matching the production `A: '0'`, where its value is assigned to a corresponding synthesized attribute (`attr_1s`) and passed back up to the root. The attribute action in the intermediate production (`A: A '1' A '1'`) calculates the sum of the children's respective synthesized values and passes it to the parent. When the value reaches the root, it becomes the input for the inherited attribute of the next pass (`$A.attr_2i = $A.attr_1s;`), or is output if there are no more passes.

The example above shows a grammar configured for two passes (`attr_1i`, `attr_1s`, `attr_2i`, `attr_2s`) with the intermediate production being used to expand two non-terminal child nodes (`A: A '1' A '1'`).

Both the number of passes (determined by the attribute count), as well as the number of non-terminal child nodes within the intermediate production are parameters of the constructed attribute grammars.

Each parser generated from such an attribute grammar is run with input restricted to a specific number of expansions of the rule `A: '0'`. The lowest possible tree depth for such an input is used.

5.1.3 AG2010

The third benchmark is based on an example attribute grammar from the Compiler Construction course at the Vienna University of Technology. The benchmarked code performs syntax and semantic checks on programs written in a simple constructed language. In particular, attribute grammars are used to determine the visibility of variables and named fields within nested scopes.

Randomly generated programs of various lengths (up to one million lines of code / 26 megabytes) are used as input. All generated programs use random identifiers as variable, field and function names, are syntactically and semantically valid; furthermore, the generator attempts to expand all rules of the grammar.

The source code of the benchmark and a short example input file are listed in Appendix B.1.

5.2 Measurements and results

5.2.1 Timing of evaluator phases

The first set of measurements is made to determine the amount of time spent in each phase of the evaluator. This places attribute evaluation and tree traversal in perspective with regard to other phases such as parsing and scanning.

Parser instrumentation

An instrumented parser is generated from each benchmark's parser specification(s). The parser uses several calls to the `clock_gettime(CLOCK_PROCESS_CPUTIME_ID, ...)` function to mea-

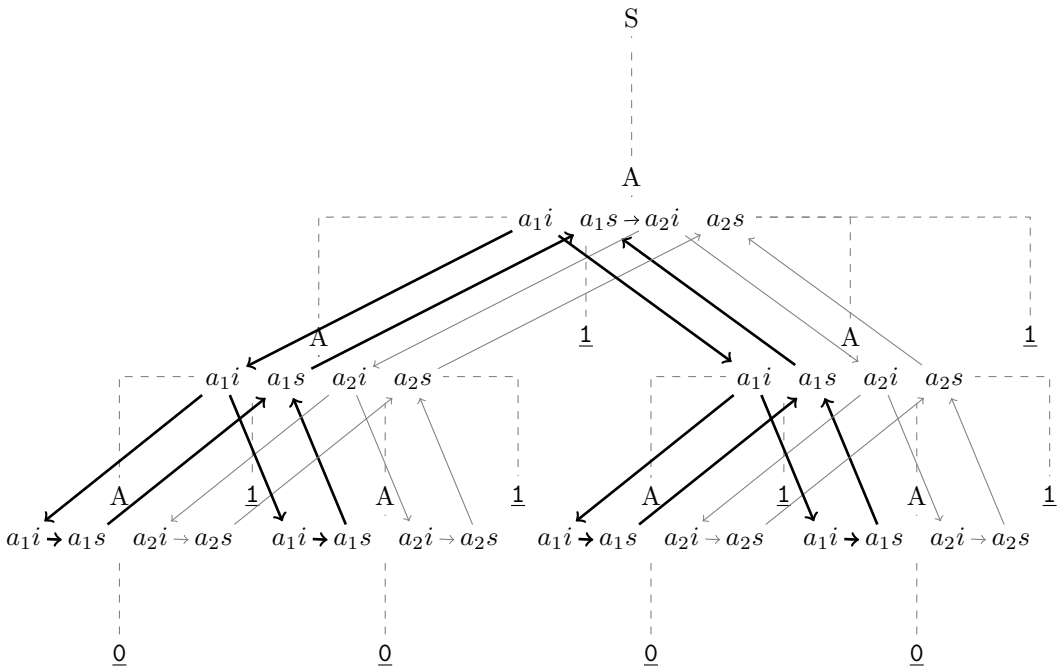


Figure 5.1: Attributed parse tree for input `0101101011` with the first "pass" highlighted. Attribute names are shortened for brevity (`attr_1s` becomes a_1s and so on).

sure the following:

- The total runtime of the parser (call to `yyparse()`), including the scanner and the attribute evaluator.
- The runtime of the scanner alone (calls to `yylex()`) – referred to as "Scanner" in the graphs below.
- The total runtime of the attribute evaluator, including:
  - The dependency resolution algorithm which builds the foundation of the topological sort. This includes construction of the parse tree.
  - Attribute action execution.
  - Time spent in tree traversals, including traversal actions.
- The runtime of the attribute actions alone – referred to as "AG actions" in the graphs below.
- The runtime of the tree traversal and traversal action code alone – referred to as "AG traversals" in the graphs below.

Samples are taken before and after entering code pertaining to one of the above, and totals are output after both parsing and attribute evaluation have completed.

Furthermore, the number of parse tree nodes (both terminals and non-terminals) is recorded.

The results are then post-processed in order to obtain:

- The parser overhead without scanning and attribute evaluation – referred to as "Parser" in the graphs below).
- The runtime of attribute evaluation (dependency resolution) alone, without time spent in actions and traversals – referred to as "AG evaluation" in the graphs below.

`clock_gettime(CLOCK_PROCESS_CPUTIME_ID, ...)` measures the CPU time used by the parser process. This function reports nanosecond accuracy on the test machine, which is an Intel® Core™ i5-6200U CPU @ 2.30GHz, running Ubuntu 16.04 with a Linux 4.4.0 kernel in 64 bit mode.

`clock_gettime()` is defined in the POSIX specification [POSIX09, p. 667-670]; support for `CLOCK_PROCESS_CPUTIME_ID` is optional [POSIX09, p. 668] but available on Linux.

Measuring instrumentation overhead

Instrumentation incurs some overhead that must be taken into account when calculating the proportions of evaluator phases to each other.

This overhead is measured using a version of the benchmark that counts only the number of measurements per phase – without actually performing the measurements themselves. The difference in execution time between this version and the regular one is subtracted proportionally from the result of each phase. In order to increase accuracy, the results are averaged over three runs of the parser.

`perf`-based timing for `libxnag`

In addition to measurements using `clock_gettime()`, it is possible to use a sampling approach based on the `perf` command[1] to analyze time spent in each function of the library version of the evaluator (for a list, see Section 4.3). The sampling approach has less overhead, does not require code instrumentation and allows for a more detailed insight into the attribute evaluator.

The command used to perform sampling is:

```
perf record -e cycles -F3000 --call-graph=dwarf <BENCHMARK>
```

For this purpose, the corresponding benchmarks are compiled with debug information (`-ggdb`). The percentage of time spent in each phase is obtained from the output of `perf report --stdio`.

Results: Bincount

Figure 5.2 shows the results of the "Bincount" benchmark for inputs of up to 20 000 characters. This benchmark run primarily allows for a comparison between the first (unoptimized) version of the evaluator – in this case built on lazy evaluation (see Section 4.5.1) – and the later ones (embedded in Bison and using `libxnag`). The first version spends almost its entire time on attribute evaluation and quickly exhibits non-linear behaviour. Both eager versions, on the other hand, scale linearly with input size. The split among parser phases is largely similar for these two (although the `libxnag` version is slightly slower in total). The lazy version requires almost 20 seconds to parse 19 900 characters, while the eager versions require around 6 and 10 milliseconds respectively.

Figure 5.3 shows the same benchmark for inputs of up to 10 million characters, this time only for the eager versions. The embedded version takes about 2.5 seconds to process 9 500 000 characters, while the `libxnag` version takes about four seconds.

A detailed comparison of the amount of time spent in each parser phase for "Bincount" is given in Figure 5.4. The percentage remains constant even when input size increases. The detailed breakdown using `perf` on the right hand side shows that of roughly 63% of time spent in "AG evaluation", 44% is spent constructing the parse tree and not actually performing topological sorting. Attributes are usually enqueued for immediate evaluation without having to rely on the dependency graph, which is evidenced by the large percentage of roughly 14% spent in `nag_dependencies_finish`. At 22.5%, traversals also take a substantial amount of time. This is because although only one traversal action exists in the tree, all nodes are actually visited by the traversal. This shows one opportunity for potential future optimization.

Results: Multipass

Figures 5.5 and 5.6 show results of the "Multipass" benchmark, which is run for various numbers of passes and tree widths as detailed in Section 5.1.2. The first figure shows timing information for the Bison-embedded version of the evaluator. The second figure shows percentage of time spent per phase and is based on measurements of the `libxnag` version with `perf`.

Unlike "Bincount", where dependency resolution is very simple, topological sorting and building a dependency graph are crucial components of this benchmark. This is evidenced by the split among attribute evaluator phases in Figure 5.6. Here, `nag_dependency_add` and `nag_evaluate` are the two more resource-intensive parts of evaluation, particularly in the case of multiple passes (the rightmost plots). Similarly to the scanner and LR parser, parse tree construction takes proportionally less time when more "passes" over the grammar are involved. Percentage of time

[1] `https://perf.wiki.kernel.org/index.php/Main_Page`

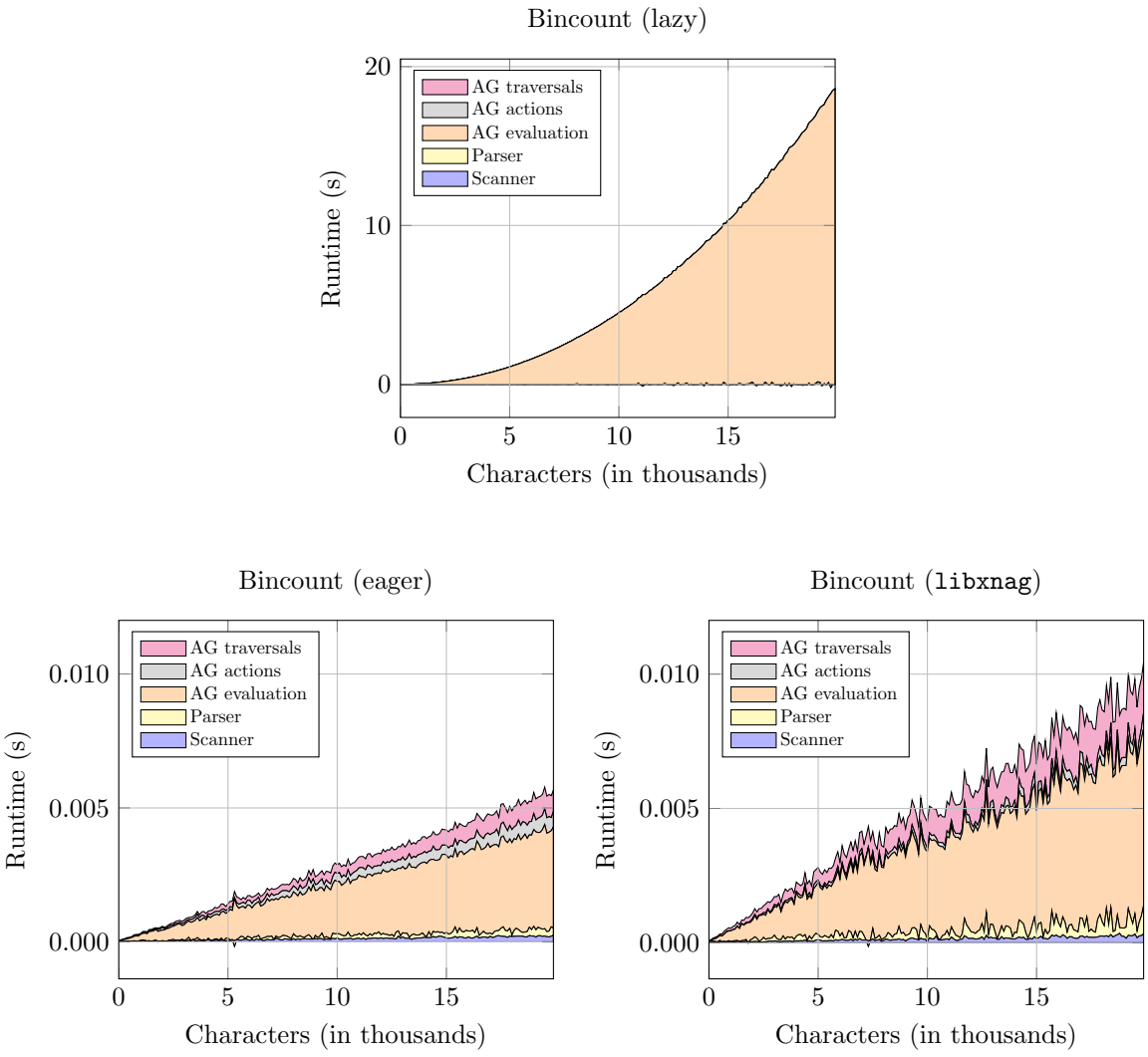


Figure 5.2: Results of “Bincount” for up to 20 000 characters.

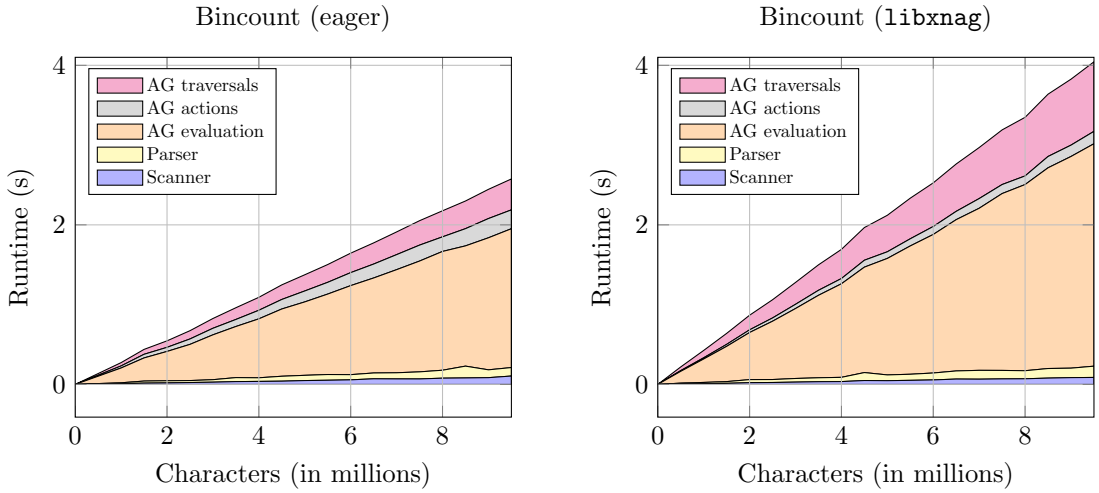


Figure 5.3: Results of "Bincount" for up to 10 million characters.

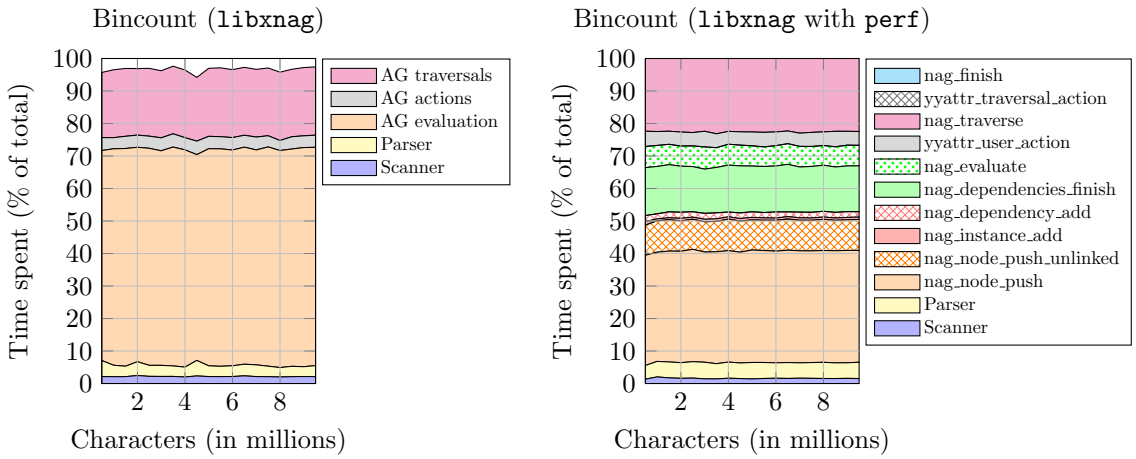


Figure 5.4: Percentage of time spent in evaluator phases for "Bincount" measured using `clock_gettime()` (left) and `perf` (right).

spent in traversals appears to be constant regardless of the number of passes, however stack trace logging reveals that a substantial amount of time is spent in `malloc_consolidate` which is triggered as part of the traversal's `printf` call. Within `nag_traverse`, actual time spent traversing the tree is proportional to tree size, not to number of attributes.

Results: AG2010

Figure 5.7 shows results of the "AG2010" benchmark and illustrates the behaviour of a more balanced attribute grammar, similar to grammars that can be found in practice.

Figure 5.8 shows the percentage of time spent in each parser phase. In this case, the percentage is constant except for that of traversal actions (which are user-specified). This can be seen more clearly in the bottom plot, where traversals and user actions are not shown.

Compared to "Bincount", more time is spent in the scanner (Bincount uses a simple call to `getchar()` while AG2010 uses a Flex scanner) and in the parser, less time is spent on creating parser nodes, and the proportion between directly enqueueing attributes and evaluating them later is skewed in favour of the latter – calls to `nag_dependencies_finish` take very little time and most time is split among `nag_dependency_add` and `nag_evaluate`.

Compared to "Multipass", the scanner and parser again are more prominent. The proportion of parse tree construction to evaluation is higher than in all "Multipass" benchmarks, suggesting a more parse tree-centric workload in this case.

In total, ignoring tree traversals and user actions, time is spent as follows:

- Scanning takes about 18%
- LR parsing takes about 8%
- Parse tree construction takes about 47%
- Dependency graph creation and attribute evaluation takes about 27%

5.2.2 Comparison with Ox

The AG2010 benchmark is used to compare execution time of the evaluator with that of an Ox-generated parser (see Section 2.7.1) obtained from the same specification.

The parsers are compiled with `-O2` and run without additional instrumentation. As Ox preallocates the memory used by the evaluator, its maximum memory requirements are specified at compile time:

```
ox -Yn1000000000 -Yc100000000 -Yr500000000 -Yt5000000 parser.y scanner.l
```

The execution time is measured using the UNIX `time` command.

Results are shown in Figure 5.9. Runtime of both the eager evaluator embedded in Bison and the `libxnag` version is roughly twice that of Ox. For a practical input of 1 013 971 lines of code (26 megabytes) that expands to 13 106 154 symbols, execution time of the implemented evaluator lies at around 6 seconds.

Given the order of magnitude involved, the evaluator can be considered fast enough for practical purposes. It is likely that this result could be improved further by implementing a more intricate memory allocation strategy.

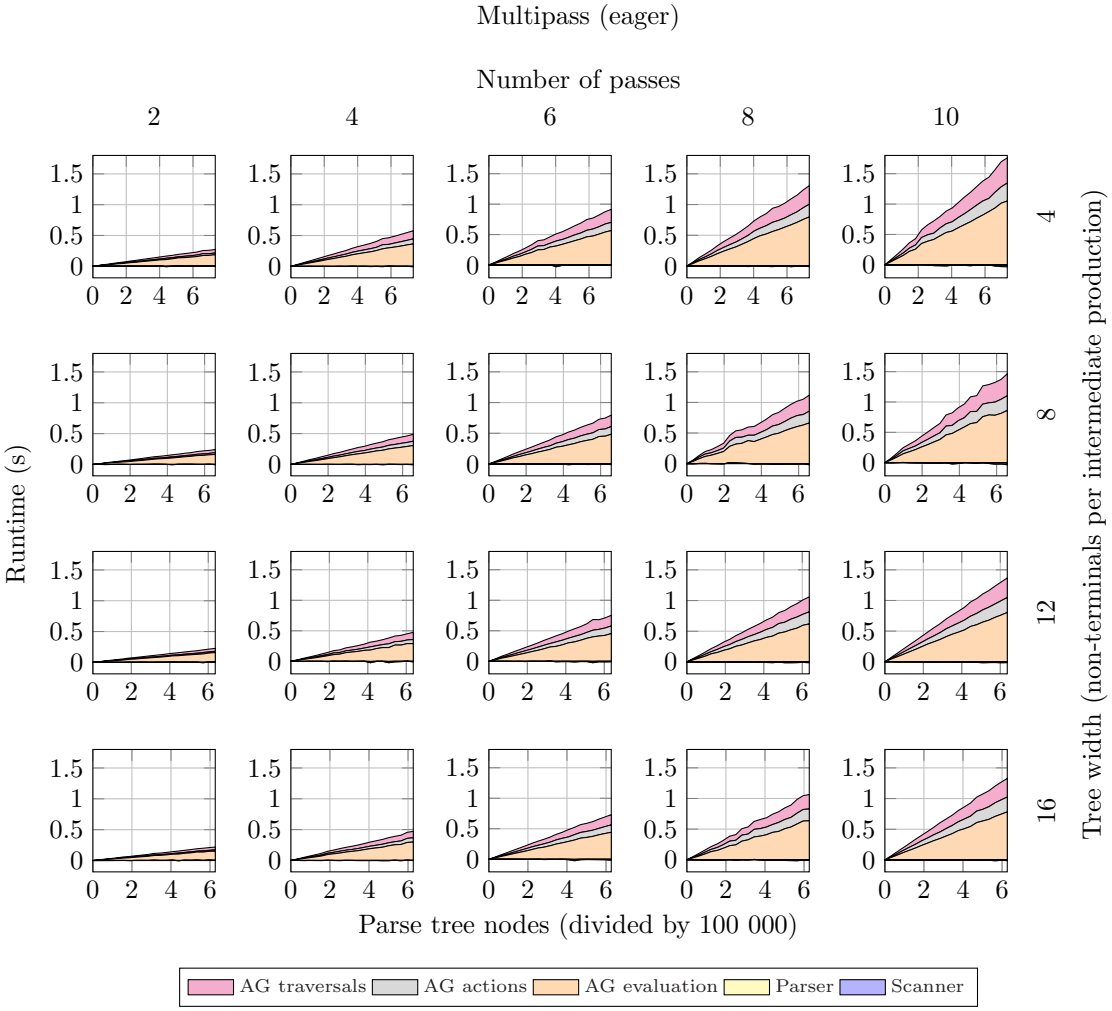


Figure 5.5: Runtime (in seconds) of eager "Multipass" benchmarks, varied by number of simulated passes (left to right), tree width (top to bottom) and input size.

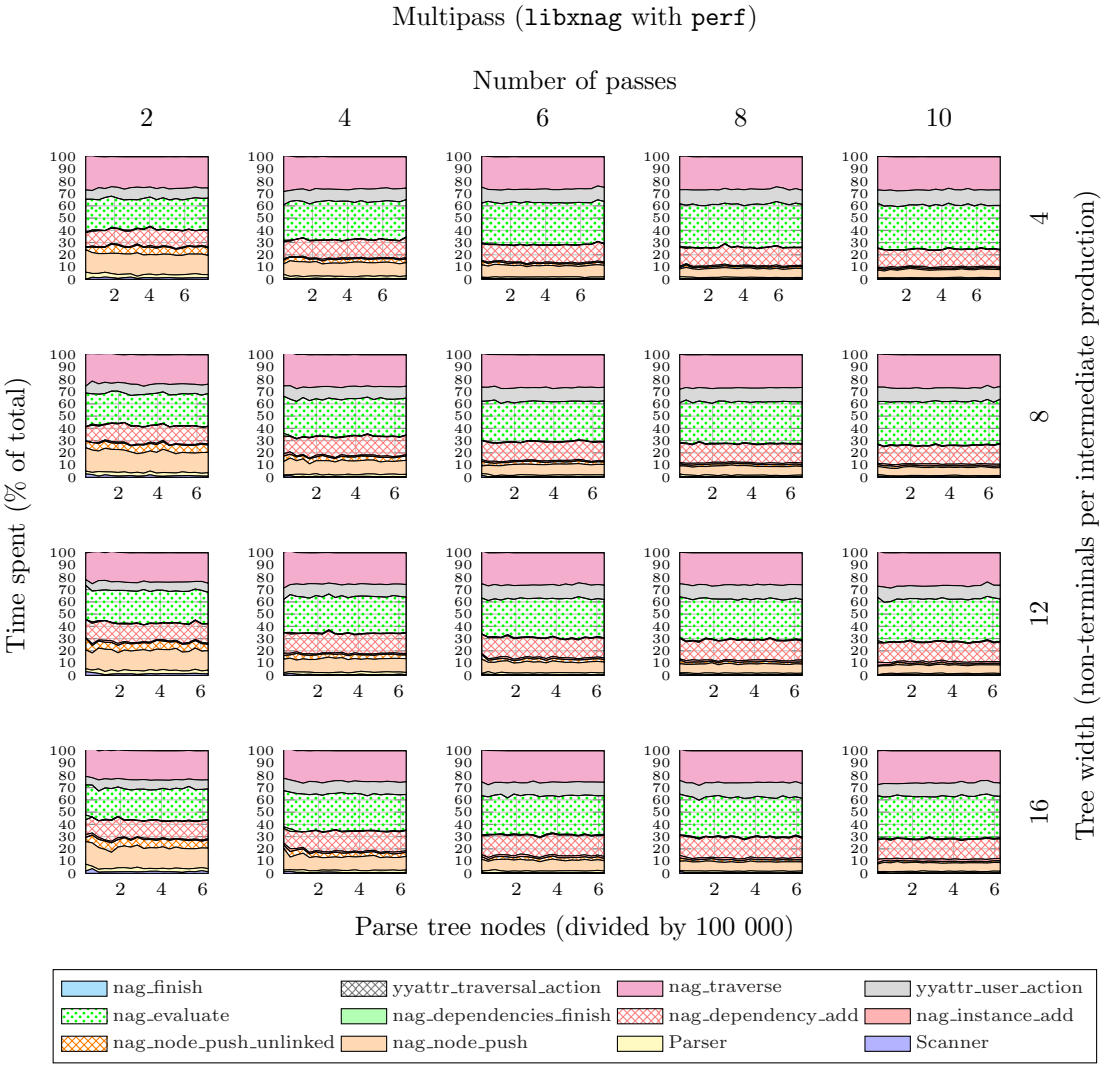


Figure 5.6: Percentage of time spent in parser phases for "Multipass", varied by number of simulated passes (left to right), tree width (top to bottom) and input size.

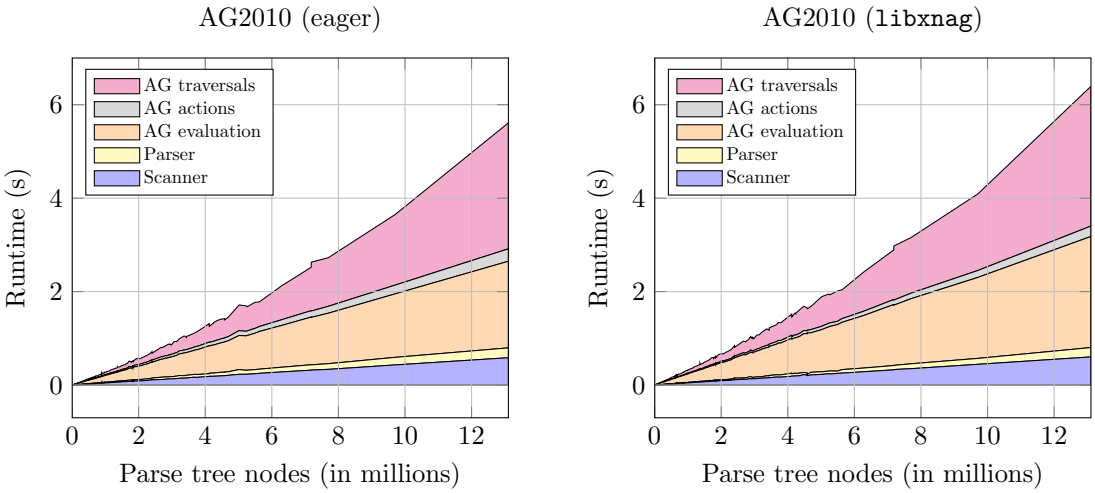


Figure 5.7: Results of “AG2010”.

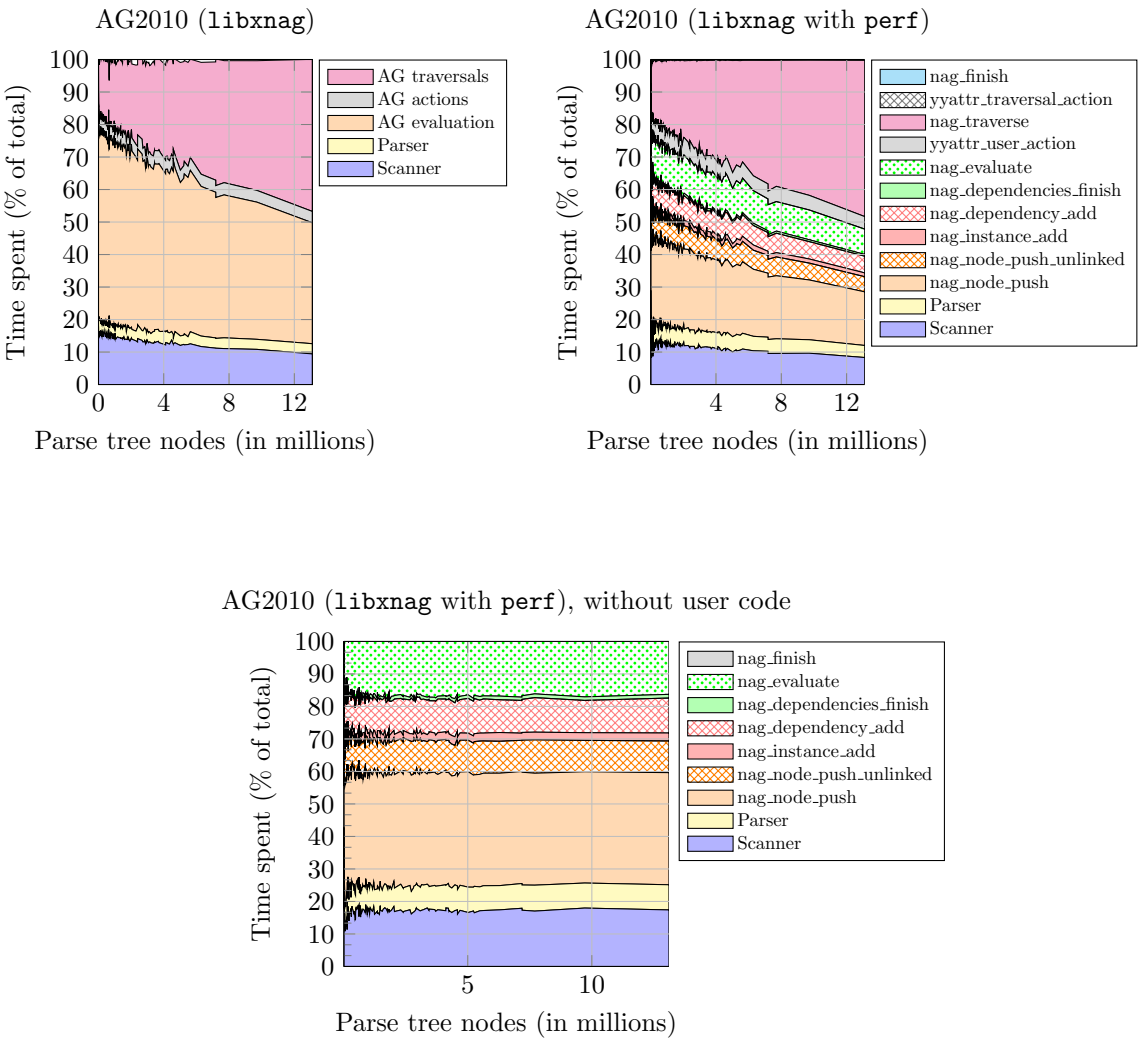


Figure 5.8: Percentage of time spent in evaluator phases for "AG2010" measured using `clock_gettime()` (upper left) and `perf` (upper right) and percentage of time spent in phases other than attribute actions / traversals (bottom).

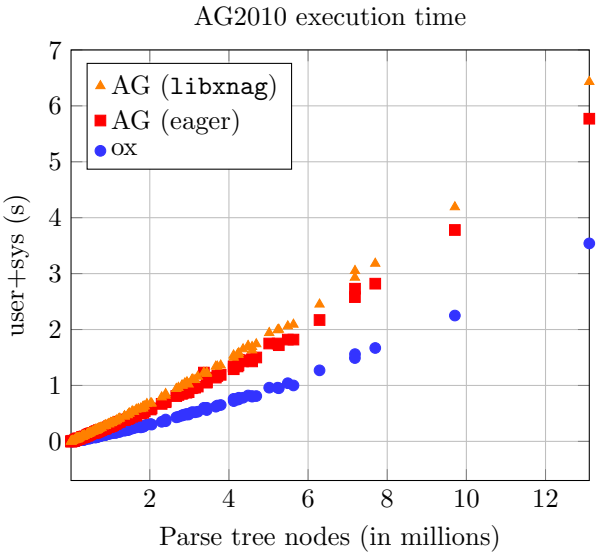


Figure 5.9: Comparison of AG2010 execution time between Ox, Bison-embedded and `libxnag` evaluator versions.

Chapter 6

Conclusion

This thesis describes the **implementation of a dynamic attribute evaluator in Bison**, an LALR(1) and GLR parser generator. A suitable and efficient topological sorting algorithm is the cornerstone of the implementation. A set of extensions to Bison's syntax allow the user to specify any well-defined LALR(1) / GLR attribute grammar. The implementation also includes tree traversal support. The syntax is similar to that of the Ox preprocessor [Bis93] but makes use of Bison's in-built way of referencing symbols and also exhibits other notable improvements, in particular the ability to explicitly declare attributes as inherited or synthesized.

Attribute grammar support is implemented for the C LALR(1) and GLR parsers.

Performance. The evaluator is fast enough for practical purposes.

Benchmarks suggest the runtime of the evaluator is linear with regard to the size of the parse tree; this aligns with the computational complexity of the underlying algorithms.

Separation of evaluator runtime into an external library. Bison uses so-called "skeleton files" written in the macro language M4 as templates to generate parsers. Most code belonging to the evaluator is embedded in a separate M4 skeleton. Existing skeletons reference sections of it to enable attribute grammar support.

The evaluator runtime also exists as an externally linkable library. The architecture of Bison, however, negates some of the advantages of using it. In particular, the skeleton approach already provides structuring and encapsulation. Much of the code present in the skeleton is still needed when interfacing with the library. The ease of using Bison-specific memory management code and the fact that runtime library dependencies are uncommon for Bison-generated parsers also speak in favour of a Bison-embedded version.

The developed evaluator runtime library `libxnag` is nevertheless flexible, offers useful encapsulation and can serve as a building block for other projects.

GLR. Various methods of combining attribute grammars and Generalized LR (GLR) are shown. The best method to apply depends on the use case, be it generating the entire parse forest or one or more valid parse trees.

A simple method that works in the case of a single valid parse tree is to perform attribute evaluation on portions of the parse tree as soon as they become unambiguous. This method is also implemented in the evaluator, where disambiguation is performed through Bison mechanisms such as precedence rules, merge functions or semantic predicates.

Appendix A

Annex

A.1 Parsing context-free grammars

Traditionally, attribute grammar evaluators have been viewed as extensions to syntax parsers. Existing systems usually interleave the two for performance reasons as for some classes of attribute grammars, evaluation can be done together with parsing, rendering extra passes of the syntax tree and as such, a mandatory representation of the syntax tree in memory, unnecessary.

This section gives an overview of the more commonly used parsing algorithms, together with their strengths and weaknesses.

A.1.1 Top-down parsers

The concept of top-down parsers [ASU86, p. 181 ff.] is very straightforward; a special kind of top-down parsers (recursive descent parsers) can even be built manually with ease. The idea of a top-down parser is to begin with the start symbol and complete the parse tree down to the leaves, akin to a preorder traversal.

Recursive decent parsers

Recursive descent parsers use one function per non-terminal which calls functions corresponding to the symbols on the right hand side of the rule. Given the following grammar with start symbol A:

$$\begin{aligned} A &\rightarrow \underline{\texttt{x}}\,BC \\ B &\rightarrow \underline{\texttt{y}} \\ C &\rightarrow B\,\underline{\texttt{z}} \end{aligned} \tag{A.1}$$

a recursive descent parser would look like this:

```
function parse():
  parse_A();

function parse_A():
  consume(x); parse_B(); parse_C();
```

```
function parse_B():
  consume(y);

function parse_C():
  parse_B(); consume(z);
```

The function *consume*() consumes a token (terminal) from the input, while the other functions are simply the mapped rules. If *consume*() does not encounter the expected token in the input string, the input does not belong to the given language.

Lookahead. For some grammars, it is impossible to tell which rule to choose without looking "ahead" into the input stream [1]:

$$\begin{aligned} A &\rightarrow \underline{\mathtt{w}}\,B \mid \underline{\mathtt{x}}\,C \\ B &\rightarrow \underline{\mathtt{y}} \\ C &\rightarrow B\,\underline{\mathtt{z}} \end{aligned} \tag{A.2}$$

The function $parse_A()$ has to look at the first input token to determine whether to pick $\underline{\mathtt{w}}\,B$ or $\underline{\mathtt{x}}\,C$:

```
function parse_A():
  if lookahead == w:
    consume(w); parse_B();

  else if lookahead == x:
    consume(x); parse_C();

  else error();
```

In general, more than one token of lookahead may be necessary to choose the right rule. Sometimes, this can be solved by left factoring as described further below.

Predictive parsers. Predictive parsers are a special kind of top-down parsers that utilize lookahead and two special sets of tokens for each non-terminal, called the *FIRST* and *FOLLOW* sets, to choose the appropriate rule for a production. Predictive parsers can be implemented both as recursive or as non-recursive parsers. In the latter case, the parser uses an explicit stack and a parser table.

A recursive parser is assumed here.

The *FIRST* sets are used to generate the `if` statements that analyze the lookahead in functions that map non-terminals. The necessity for the *FIRST* sets stems from the fact that grammars can have ϵ-productions, as in productions of the form $B \rightarrow \epsilon$, with ϵ denoting an empty string. As an example, for the grammar:

$$\begin{aligned} A &\rightarrow B\,\underline{\mathtt{x}} \mid C\,\underline{\mathtt{z}} \\ B &\rightarrow \underline{\mathtt{y}} \mid \epsilon \\ C &\rightarrow \underline{\mathtt{z}} \end{aligned} \tag{A.3}$$

[1] $A \rightarrow \alpha \mid \beta$ is equivalent to the two separate productions $A \rightarrow \alpha$ and $A \rightarrow \beta$.

the function $parse_A()$ will have to be aware of the token $\underline{\texttt{x}}$ because of the possible derivation $A \Rightarrow B\,\underline{\texttt{x}} \Rightarrow \underline{\texttt{x}}$, where this token is the first to be read from input and as such a possible value of the lookahead.

The *FIRST* set for a symbol X is thus the set of tokens with which a derivation of X may begin and is determined as follows [ASU86, p. 189]:

```
if X is a terminal:
  FIRST(X) := {X}

if production X → ϵ exists:
  FIRST(X) += {ϵ}

if X is a non-terminal:
  for all productions X → Y1Y2...Yk of X:
    for i := 1 to k:
      if FIRST(Y1) ... FIRST(Yi−1) all contain ϵ:
        FIRST(X) += FIRST(Yi)

    if FIRST(Y1) ... FIRST(Yk) all contain ϵ:
      FIRST(X) += {ϵ}
```

The algorithm begins with all *FIRST* sets being empty and must be run repetitively for each symbol until no new items are added to the *FIRST* sets. The *FIRST* set for a string of symbols can be computed analogously, by starting with the *FIRST* set of the first symbol and successively adding the *FIRST* sets of the next symbols if the *FIRST* set of the last symbol included ϵ. If ϵ is contained in all the *FIRST* sets of the symbols, it is also added to the computed *FIRST* set.

FOLLOW sets are used to track (descend into) non-terminals that derive ϵ. The parser triggers the descent on encountering the first token following ϵ in the current production [ASU86, p. 189]:

```
FOLLOW(S) := {$}

do until FOLLOW sets no longer change:
  for all non-terminals A and B matching A → αBβ:
    FOLLOW(B) += FIRST(β) - {ϵ}
    if ϵ is in FIRST(β):
      FOLLOW(B) += FOLLOW(A)

  for each production A matching A → αB:
    FOLLOW(B) += FOLLOW(A)
```

The special symbol \$ is used to denote the end of input.

A recursive predictive parser for the grammar:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow \underline{+}TE' \mid \epsilon \\ T &\rightarrow \underline{0} \mid \underline{1} \mid \underline{(}E\underline{)} \end{aligned} \tag{A.4}$$

is given as an example. To begin with, the *FIRST* and *FOLLOW* sets for each symbol are determined:

$$FIRST(E) = \{\underline{0}, \underline{1}, \underline{(}\} \quad FIRST(E') = \{\underline{+}, \epsilon\} \quad FIRST(T) = \{\underline{0}, \underline{1}, \underline{(}\}$$
$$FIRST(\underline{0}) = \{\underline{0}\} \quad FIRST(\underline{1}) = \{\underline{1}\} \quad FIRST(\underline{)}) = \{\underline{)}\}$$
$$FIRST(\underline{(}) = \{\underline{(}\} \quad FIRST(\underline{+}) = \{\underline{+}\}$$

$$FOLLOW(E) = \{\underline{)}, \$\} \quad FOLLOW(E') = \{\underline{)}, \$\} \quad FOLLOW(T) = \{\underline{+}, \underline{)}, \$\}$$

The *FIRST* sets for each of the grammar rules' right hand sides are also calculated:

$$FIRST(TE') = \{\underline{0}, \underline{1}, \underline{(}\} \quad FIRST(\underline{+}TE') = \{\underline{+}\} \quad FIRST(\epsilon) = \{\epsilon\}$$
$$FIRST(\underline{(}E\underline{)}) = \{\underline{(}\}$$

The parse functions first analyze the lookahead to see which rule has to be applied in the context of the left hand side symbol. If the lookahead matches one of the tokens from the *FIRST* set of the right hand side string of a specific rule, then that rule is chosen. Additionally, a rule is chosen if it can derive ϵ and the lookahead is contained in the *FOLLOW* set of the left hand side. After a rule is selected by the parser, the corresponding *parse*() or *consume*() functions for each of the symbols on the right hand side are called. If the lookahead did not match any of the tokens, an error is thrown.

```
function parse_E():
  if lookahead is in {0, 1, (}:
    /* rule E → TE' */
    parse_T(); parse'_E();
  else:
    error();

function parse_E':
  if lookahead is in {+}:
    /* rule E' → +TE' */
    consume(+); parse_T(); parse_E';
  else if lookahead is in {), $}:
    /* rule E' → ϵ */
  else:
    error();

function parse_T():
  if lookahead is in {0}:
    consume(0);
  else if lookahead is in {1}:
    consume(1);
  else if lookahead is in {(}:
    consume(();
  else:
    error();
```

A table-driven predictive parser

The described recursive predictive parser can be modified to use a stack instead of function calls and a table derived from the *FIRST* and *FOLLOW* sets of the grammar instead of concrete application logic where the two sets are embedded.

The predictive table. To each combination of a non-terminal (represented by the current parse function in the recursive parser) and input token (lookahead), the table associates a production that will be "descended into" by the algorithm by placing the symbols from the right hand side of production onto the stack. Cells that remain empty denote unreachable parser states, they are treated as error entries. The table is built as follows [ASU86, p. 190]:

```
for each production A → α:
  for each terminal a in FIRST(α):
    M[A, a] := A → α
  if ϵ is in FIRST(α):
    for each terminal b in FOLLOW(A):
      M[A, b] := A → α
    if $ is in FOLLOW(A):
      M[A, $] := A → α
```

The table for the example grammar is:

	0	1	+	(	)	$
E	$E \to TE'$	$E \to TE'$		$E \to TE'$		
E'			$E' \to +TE'$		$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to 0$	$T \to 1$		$T \to ($		

The parsing algorithm. Given an input token (the lookahead) and the symbol at the top of the stack, the parser proceeds by either consuming the symbol, if it is a terminal, or by descending into the rule specified by the corresponding table entry. A formulation of the algorithm in pseudocode is given below [ASU86, p. 187]:

```
push the start symbol onto the stack;

while input exists:
  l := lookahead;
  S := symbol on top of stack;
  remove S from stack;
  if S is a terminal:
    consume(S);
  else if M[S, l] is empty:
    error();
  else:
    a_1 ... a_k := right hand side of M[S, l];
    push a_k ... a_1 onto the stack;
```

Limitations and pitfalls

Top-down parsers, although conceptually simple, have a series of limitations and pitfalls, which are described below.

Handling left recursivity. Top-down parsers cannot handle left recursive grammars[2], where for a non-terminal A, a derivation $A\alpha$ is possible, such as in the following example:

$$A \to A\,\underline{\texttt{a}} \mid \underline{\texttt{a}}$$

In this case, A can be derived either to $\underline{\texttt{a}}$ or to $A\,\underline{\texttt{a}}$, the latter being the reason for the grammar's left recursivity. The reason why the top-down parser fails is that it would first have to "descend" into A, however, since it is unknown how many $\underline{\texttt{a}}$s will be in the input string, the number of descents (calls to the `A()` function) cannot be determined. A predictive parser implemented as outlined in this section would loop forever for the given example.

Therefore, any left-recursive grammar that is to be parsed using this method has to be transformed into an equivalent grammar not exhibiting the property. An algorithm for this purpose exists [ASU86, p. 178] and is specified further below.

The basic idea of the algorithm is to gradually replace immediate left recursion, meaning productions of the form:

$$A \quad \to \quad A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \beta_n$$

by an equivalent pair of productions not exhibiting left recursion:

$$\begin{aligned} A &\to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A' \\ A' &\to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

This is done incrementally for subsets of the grammar's rules, so that non-immediate left recursivity is also covered:

```
Order the non-terminals in the grammar from A_1 to A_n.
for i := 1 to n:
  for j := 1 to i − 1:
    with A_j → δ_1 | δ_2 | ... | δ_k as all productions for A_j:
      Replace each production of the form A_i → A_j γ
      by A_i → δ_1 γ | δ_2 γ | ... | δ_k γ.

    Eliminate immediate left recursion in all A_i.
```

The algorithm requires the input grammar to be both:

- Cycle free: No non-terminal A may be derivable to A.
- ϵ-free: No rules of the form $A \to \epsilon$ may appear in the grammar, except if the symbol on the left side is the start symbol S and S does not appear on the right hand side of any other rule in the grammar.

Left factoring. Depending on the top-down algorithm and how many tokens of lookahead are used, the parser may encounter problems with rules having a common prefix, where it is not immediately clear which one to choose:

[2]This refers to left-to-right parsers, which parse the input from left to right. Right-to-left parsers, rarely seen in practice, can handle left recursive grammars, but cannot handle right recursive grammars.

$$\begin{aligned} A &\rightarrow \alpha\beta_1 \\ A &\rightarrow \alpha\beta_2 \end{aligned}$$

The choice of rule is deferred until it is possible by introducing a common rule:

$$A \rightarrow \alpha A'$$

and the rules of the non-terminal A' differentiating between β_1 and β_2 at the beginning:

$$\begin{aligned} A' &\rightarrow \beta_1 \\ A' &\rightarrow \beta_2 \end{aligned}$$

In order to get a left factored grammar, such transformations must be applied systematically to all non-terminals.

Summary of limitations. The limitations of top-down parsers explained above are circumvented by using equivalent, adequately formed grammars. In the context of attribute grammars and parsing in general, this is an undesired side effect, since the symbols (non-terminals in particular) no longer represent a sensible, readable syntactic specification of the given language.

A.1.2 Bottom-up parsers

Unlike top-down parsers, bottom-up parsers start by detecting the leaves of a parse tree, working their way up to the root. They use a successive series of "reverse derivations" that take part of the already parsed input, a so-called *handle*, match it to the right hand side of a rule and replace it with the left hand side. The process is called "handle pruning" and the operation itself is called a reduction, as a series of symbols is "reduced" to a non-terminal.

Shift-reduce parsers are a common type of bottom-up parsers. Given an input token, the parser chooses between four operations:

shift Assume the token is part of the right hand side of a rule, continue parsing accordingly.

reduce Perform a reduction; Replace some of the shifted tokens with a matching non-terminal.

accept Accept the input as part of the language specified by the grammar.

error Reject the input since the input string is not part of the language; notify that an error state has been reached.

The state of a shift-reduce parser is usually kept as a stack. When a shift operation is performed, the input token is pushed onto the stack. A reduce operation is only executed when the symbols at the top of the stack exactly match the right hand side of a rule, meaning that a handle has been found. In this case, the symbols corresponding to the handle are removed from the stack and replaced by the matching non-terminal.

For shift-reduce parsers, handles are always found on top of the stack [ALSU07, p. 237].

LR parsers

LR parsers are an implementation of shift-reduce parsers. They are composed of a "driver" algorithm which uses a stack and two tables called *action* and *goto* to perform the parsing (see Figure A.1). The driver algorithm is predefined, the *action* and *goto* tables must be computed for each grammar [ALSU07, p. 248]. Since this is complicated to do by hand, LR parser generators exist for this purpose.

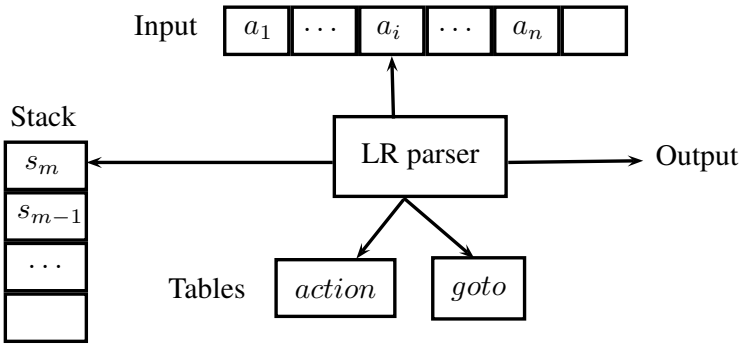


Figure A.1: LR parser schematic [ALSU07, p. 248]

The actual symbols need not be saved directly on the stack (as would be the case for shift-reduce parsers). Instead, LR parsers employ states which contain summarized information about the context. The state on top of the stack, together with the current input token (lookahead) determine the next action of the parser.

The tables. The *action* table encodes, for each combination of state and lookahead token, one of the four actions of a shift-reduce parser; the *goto* table is used only during reductions:

- A $shift$ action specifies a new state s' that will be pushed onto the stack and used for the next action of the parser.
- A *reduce* action specifies a rule to reduce with. After the number of states corresponding to the number of symbols on the right hand side of the rule have been removed from the stack, a new state s' is pushed. s' is determined by looking up the *goto* table entry for the state now on top of stack and the reduced non-terminal.
- *accept* and *error* actions have no additional parameters.

If a table cell has more than one entry, a *shift/reduce* or *reduce/reduce conflict* is said to exist, depending on whether a $shift$ action and one or more *reduce* actions or more *reduce* actions are listed in the cell. Such conflicts can arise if the grammar is ambiguous, not LR, or does not belong to the subclass of grammars the table generation algorithm adheres to (for example SLR or LALR, see below).

The algorithm. The LR driver algorithm simply follows the instructions laid out in the action table until the input is either accepted or an error occurs:

```
push the initial state onto the stack;
```

```
while true:
  s := symbol on top of stack;
  a := lookahead;

  if action[s, a] = shift s':
    push s' onto the stack;
    consume(a);

  else if action[s, a] = reduce A → β:
    pop |β| symbols from the stack;
    s' := symbol on top of stack;

    push goto[s', A] onto the stack;
    /* reduce A → β */

  else if action[s, a] = accept:
    /* done */
    break;

  else:
    error();
```

Calculating the tables is the most complex part of LR parsing. Several algorithms exist for this purpose, including simple LR (SLR), canonical LR and look-ahead LR (LALR). The subclasses of grammars that can be parsed with these methods are strictly contained in each other:

$$\text{SLR} \subset \text{LALR} \subset \text{Canonical LR}$$

Incidentally, the size of the parsers generated using these three algorithms increases together with its expressiveness.

SLR. The SLR algorithm uses so-called LR(0) sets to create a finite automaton that is later translated into *action* and *goto* tables. A LR(0) set is composed of LR(0) items, which encompass all productions of a grammar with a dot at any position on the right hand side. For example, for the rule

$$A \rightarrow BC$$

the following items are valid:

$$\begin{aligned} A &\rightarrow \cdot BC \\ A &\rightarrow B \cdot C \\ A &\rightarrow BC \cdot \end{aligned}$$

The dot symbolizes the progression of the parser within a rule; before parsing C in the above example, the parser will execute actions corresponding to $A \rightarrow B \cdot C$. After parsing C, actions for $A \rightarrow BC\cdot$ become relevant, such as reducing using $A \rightarrow BC$. As such, items are the starting point in calculating states; however, more items (a set) may be summed up by a single state.

Given a set of items I, the *closure* of I is defined as follows:

- All items in I belong to $closure(I)$.

- For each item matching $A \rightarrow \alpha \cdot B\beta$, all possible items matching $B \rightarrow \cdot\gamma$ also belong to $closure(I)$.

The definition is applied repetitively until $closure(I)$ no longer changes.
As an example, given the grammar:

$$\begin{array}{lcl} S & \rightarrow & E \\ E & \rightarrow & E \underline{+} N \\ E & \rightarrow & T \\ T & \rightarrow & \underline{(}\, N \,\underline{)} \\ N & \rightarrow & num \end{array} \tag{A.5}$$

$closure(\{S \rightarrow \cdot E\})$ is calculated by starting with $S \rightarrow \cdot E$. By matching it with $A \rightarrow \alpha \cdot B\beta$ ($A = S$ $\alpha = \epsilon$, $B = E$, $\beta = \epsilon$), it results that items belonging to all productions of E must be added:

$$\begin{array}{lcl} E & \rightarrow & \cdot E \underline{+} T \\ E & \rightarrow & \cdot T \end{array}$$

In the next step, items corresponding to all productions of E and T must be dealt with. Since items for E have already been added, only items for T remain:

$$T \rightarrow \cdot \underline{(}\, N \,\underline{)}$$

The entire set thus contains:

$$\begin{array}{lcl} S & \rightarrow & \cdot E \\ T & \rightarrow & \cdot E \underline{+} T \\ E & \rightarrow & \cdot T \\ T & \rightarrow & \cdot \underline{(}\, N \,\underline{)} \end{array}$$

The other operation needed for calculating SLR tables is called *goto* [ASU86, p. 224]. $goto(I, X)$ is the closure of the set of items which contain all matching $A \rightarrow \alpha X \cdot \beta$, provided that $A \rightarrow \alpha \cdot X\beta$ is in I.

For Grammar A.5, $goto(\{T \rightarrow E \cdot \underline{+} T\}, \underline{+})$ contains:

$$\begin{array}{lcl} T & \rightarrow & E \underline{+} \cdot T \\ T & \rightarrow & \cdot E \underline{+} T \\ T & \rightarrow & \cdot \underline{(}\, N \,\underline{)} \\ E & \rightarrow & \cdot T \end{array}$$

The first element is obtained by matching the given rule (moving the dot over the $\underline{+}$), the rest of the elements belong to the closure of the new set.

The canonical collection of LR(0) sets together with the *goto* operation can be viewed as describing a finite automaton that accepts all *viable prefixes* of a grammar [ALSU07, p. 245-247,256-257]. Items beginning with a dot ($B \rightarrow \cdot\gamma$) can be reached from an item with the dot right before the original item's left hand side ($A \rightarrow \alpha \cdot B\beta$) without consuming any symbol. This is equivalent to an ϵ transition in the automaton; the *closure* operation is used to remove these edges and merge states, turning the non-deterministic automaton into a deterministic one.

The collection is initialized with the closure of the start rule and is built by iteratively adding all sets that can be computed by application of *goto* until a fixed point is reached. In order to simplify the generation of *accept* actions, the input grammar is *augmented* by adding a new symbol S' and a rule $S' \rightarrow S$ to it, where S is the start symbol of the input grammar. The augmented grammar thus has a new start symbol S' and a unique start rule that, when reduced, triggers the *accept* action.

```
C := {closure({S' → · S})};
do until C no longer changes:
  for each I in C:
    for each grammar symbol X:
      if goto(I, X) != {} and goto(I, X) not in C:
        C += goto(I, X);
```

The LR tables (*action* and *goto*) can be derived from the canonical collection of LR(0) sets:

- Each set I_i designates a state i.
- Items from I_i matching $A \rightarrow \alpha \cdot a\beta$, whereas a is a terminal, generate the entry $action[i, a] :=$ shift$j$, whereas $I_j = goto(I_i, a)$.
- Items from I_i matching $A \rightarrow \alpha\cdot$, whereas A is not the start symbol, generate $action[i, a] :=$ reduce$A \rightarrow \alpha$ for all a in $FOLLOW(\alpha)$.
- Items from I_i matching $S \rightarrow \alpha\cdot$, whereas S is the start symbol, generate $action[i, \$] :=$ accept.
- For all items I_i and non-terminals X, generate $goto[i, X] := j$, whereas $I_j = goto(I_i, X)$.

SLR cannot parse all LR grammars. A notable exception is Grammar A.6 describing simple assignments, where the symbol $*$ indicates "content of", akin to C pointers [ASU86, p. 229]:

$$\begin{aligned} S &\rightarrow L \underline{=} R \\ S &\rightarrow R \\ L &\rightarrow \underline{*} R \\ L &\rightarrow id \\ R &\rightarrow L \end{aligned} \tag{A.6}$$

The SLR algorithm produces a table that has multiple actions defined for the state $[\{S \rightarrow L\cdot \underline{=} R, R \rightarrow L\cdot\}, \underline{=}]$ – either a $shift$ or a *reduce* can be executed. While the grammar is not ambiguous, not involving the lookahead in the calculation of separate states causes this result.

LALR

LALR can be viewed both as a more complex form of SLR or as a simplification of canonical LR. *Canonical LR* works by also embedding lookahead in items – the sets of items constructed in this manner are called LR(1) sets, and the algorithm is essentially the same as for SLR. The LR(1) sets are also the starting point of LALR [ALSU07, p. 260-264], so they are detailed below:

closure(I) is defined as a fixed point function, using the following rules:

- All items in I belong to *closure*(I).
- For each item matching $[A \rightarrow \alpha \cdot B\beta, a]$, all possible items matching $[B \rightarrow \cdot\, \gamma, b]$ also belong to *closure*(I), whereas b is a terminal from $FIRST(\beta a)$.

goto(I, X) is defined analogously to LR(0) and is the closure of the set of items which contain all matching $[A \rightarrow \alpha X \cdot \beta, a]$, provided that $[A \rightarrow \alpha \cdot X\beta, a]$ is in I.

The canonical set of LR(1) items is built similarly to the set of LR(0) items, starting with the closure of items $\{[S' \rightarrow \cdot\, S\}, \$])\}$.

Canonical sets of LR(1) items are very large, since they embed all possible lookaheads. It has been noticed, however, that multiple sets exist where the items are the same except for the lookahead. In this case, the *item cores* are said to be the same; such sets are merged to form an LALR set [ALSU07, p. 267].

The merging of sets is guaranteed not to produce any shift/reduce conflicts for a valid LR(1) grammar, since shift actions only depend on item cores. However, reduce/reduce conflicts may arise [ALSU07, p. 267-268]. On invalid input, an LALR parser may perform a reduce operation first instead of directly signalling an error, but the error will arise when the a token is shifted [ALSU07, p.266-267].

As to not generate the entire canonical sets of LR(1) items, a more direct approach is usually employed. In order to save space, only the so-called *kernel items* of the LR(0) set are calculated – items where the dot is not at the beginning, except for the "start item", $S' \rightarrow S$. Afterwards, the lookaheads are calculated for each item, resulting in the LALR(1) set.

Lookaheads can be obtained from two sources – they are either "spontaneously generated" for an item or "propagated" from one item to another. An algorithm that discovers them, based on the propagation of a dummy lookahead [ALSU07, p. 272], is given below. The algorithm takes two parameters, I as a set of LR(0) items and X as a non-terminal. It generates SP, the set of lookaheads spontaneously generated for an item in $goto(I, X)$ and LA, the set of items lookaheads are propagated from in $goto(I, X)$.

```
K := kernel of I;
for each item A → α·β in K:
  J := closure({[A → α·β, #]});
  for each [B → γ·Xδ, a] in J with a ≠ #:
    SP[B → γ·Xδ] += a;
  if [B → γ·Xδ, #] is in J:
    LA[B → γ·Xδ, I] += A → α·β;
```

To obtain the LALR(1) set, the algorithm must be executed iteratively for each kernel of the set of LR(0) items until no more lookaheads propagate. Parsing tables resulting from the LALR(1) set are then calculated same as for LR(1) sets.

A.1.3 Generalized LR

Generalized LR (GLR) parsers [Tom84; Tom85; Tom87] extend LR parsers by allowing non-cyclical ambiguous grammars to be processed. The algorithm is designed with natural language

processing in mind, a domain where grammar complexity is higher than for programming languages, but lower than that possible with general context-free languages (for which algorithms such as the Cocke Younger Kasami algorithm or Earley's algorithm have been devised).

The GLR algorithm is a so called *all-path parsing algorithm* based on a *graph-structured stack*. The standard LR algorithm employs a simple stack to keep record of the current state; the symbol on top of the stack, together with the input token, determine the next action of the parser and thus the new contents of the stack (see Appendix A.1.2). Having more than one entry in the *action* table (a shift/reduce or a reduce/reduce conflict) means that the stack would need to contain different values at once – this introduces ambiguity and thus non-determinism in the parsing process. GLR parsers choose to "split" the stack in case of conflict and pursue all possibilities.

In his paper [Tom85], Tomita offers several refinements of the stack-like data structure that backs the multiple parses. He starts with a "stack list" – here, the parser can have one or more "processes" running in parallel. Processes perform shift actions synchronously (thus always reading the same input). When a process encounters a conflict, it spawns more processes to handle all alternatives; each process has its own copy of the stack. When a process encounters an error state, it is killed. This paradigm is seen as inefficient – processes cannot communicate with each other or share duplicated work. The number of stacks used grows exponentially with each ambiguity encountered.

The next refinement is that of a "tree-structured stack". Here, the author observes that processes with the same state on top of the stack will act the same until the state is popped from the stack, so the processes can be merged into one. The stacks are then represented as a tree, with the common top of stack as the root of the tree. This representation saves some space, but the stack up to the common symbol still has to be copied to all processes.

The last refinement is the "graph-structured stack", in which common stack prefixes are also merged to save space, in addition to the optimizations of the tree-structured stack.

Tomita also offers a compact representation of a parse forest (a collection of parse trees), which is the output of the GLR algorithm. He uses two optimizations [Tom85, p. 758-759] to avoid an exponential number of trees in the forest: "sub-tree sharing" and "local ambiguity packing". Sub-tree sharing involves using only one node pointer for identical sub-trees and is achieved by pushing pointers to nodes of the parse tree instead of symbols [3] onto the stack; a new node is only created if an identical node does not exist. Local ambiguity packing refers to trees which have identical leaf nodes and roots; these are detected by the algorithm as symbols/node pointers surrounded (left and right) by identical states and merged.

An example of operations on a graph-structured stack is given in Figure A.2. The figure shows a simple stack, the effect of a shift/reduce conflict with two states being removed and one added for the *reduce* operation, and one state being added for the $shift$ operation and lastly how the stack is merged back together by a common shift operation.

In the case of a non-ambiguous grammar, GLR has the advantage of being as efficient as the underlying LR parser (except for some minor overhead); the runtime complexity of GLR "for most natural language grammars" is $O(n^3)$ [Tom85, p. 761].

[3] The LR parser described by Tomita pushes symbols, as well as states, onto the stack. Since symbols can be left out of LR parser stacks [ASU86, p. 216], his representation is used mainly for explanatory purposes.

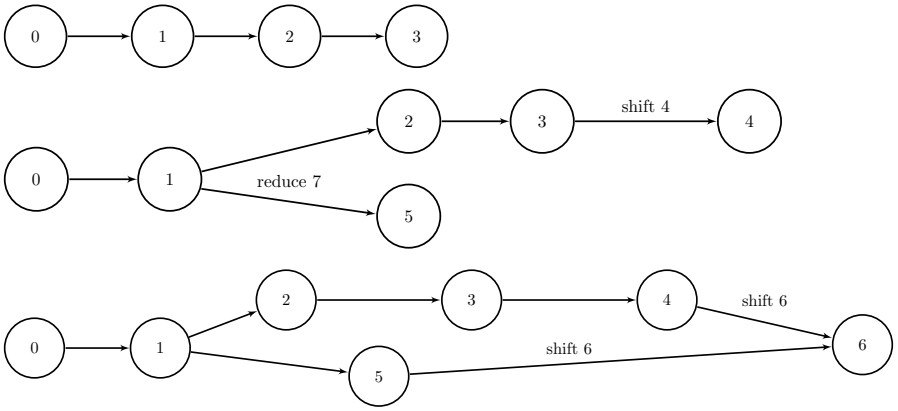


Figure A.2: Operations on a graph-structured stack: (a) simple stack, (b) shift/reduce conflict, (c) common shift

A.2 Techniques for static evaluation of attribute grammars

A.2.1 Subclasses of attribute grammars

Due to the exponential complexity involved in handling well-defined attribute grammars, several subclasses of AGs have been searched for and analyzed throughout literature. This subsection offers an overview of the more important subclasses.

Tree walk-based classification

A wide range of subclasses can be categorized by the operations executed on the parse tree during the evaluation stage [DJL88, p. 23ff][EF89, p. 69ff].

A *visit* to a parse tree node involves entering a node, performing a calculation on that node (such as evaluating an attribute) and finally exiting the node. A *sweep* is a traversal of the parse tree such that each subtree is visited exactly once. A *pass* involves a depth-first traversal of the parse tree, where nodes are visited in a fixed order – either left-to-right or right-to-left.

The classification also factors in the number of visits to a node or of sweeps or passes performed on the tree, respectively. Furthermore, a differentiation is made between *pure* and *simple* evaluators: pure evaluators are non-deterministic in that they can choose which attributes to evaluate at runtime; simple evaluators have a pre-defined set of attributes that they evaluate at a certain point.

For a *pure k-visit* AG, the evaluator thus visits each node k times, and chooses at runtime which attributes to evaluate. A *pure multi-visit* AG is a pure k-visit attribute grammar for which $k \geq 1$ [EF89, p. 846]. *k-sweep*, *k-pass*, *multi-sweep* and *multi-pass* AGs are defined analogously.

The hierarchy as well as the complexity of the evaluator and of the membership test are given in literature [EF89, p. 851] [DJL88, p. 26-27,29]. It can be noted that the *pure multi-visit* class is equivalent to that of well-defined attribute grammars and that the *simple multi-visit* class is equivalent to that of l-ordered attribute grammars (described later on). The membership test is polynomial for all *simple* evaluators except *simple multi-visit*, for which it is NP-complete.

One-visit attribute grammars

One of the classes with the lowest complexity is that of one-visit attribute grammars [AM91, p. 99ff]. Given an unattributed parse tree, all attributes can be calculated during one traversal (sweep) through the tree.

Examples of one-visit attribute grammars are **S-attributed grammars** and **L-attributed grammars**.

S-attributed grammars were first mentioned by Knuth [Knu68]. They only contain synthesized attributes. Both detecting and evaluating an S-attributed grammar is straightforward; due to the lack of inherited attributes, *IS-SET*s can contain at most one element, an empty graph. Scanning the grammar to ensure only synthesized attributes are used can be done in linear time in the size of the grammar. Evaluating an S-attributed grammar requires at most a postorder (bottom-up) pass through the parse tree.

Formally, any attribute grammar can be converted to an S-attributed grammar. An intuitive method of performing the transformation is to synthesize an attributed parse tree containing the values of all initial synthesized attributes with no dependencies, placeholders for the attributes that have not been calculated yet and links to the semantic rules and dependencies existing between attribute instances. The parse tree can then be subjected to dynamic evaluation as described in Section 2.3.3.

L-attributed grammars [ALSU07, p. 313ff][AM91, p. 100] are grammars that can be evaluated in one left-to-right pass of the parse tree. This is a superclass of S-attributed grammars, as it also allows grammars with inherited attributes. For a given production $X_{p_0} \to X_{p_1} \dots X_{p_n}$, all attribute occurrences associated with right hand side symbols (a, p, j), where $j \geq 1$ may only depend on occurrences (b, p, k) where $k < j$.

L-attributed grammars are an important subclass when considering left-to-right LL and LR parsers, as they can be evaluated efficiently in such cases. An L-attributed grammar backed by an LL context-free grammar is called an LL-attributed grammar and can be evaluated entirely during the parsing stage [AM91, p. 189].

Since the LR technique parses input in a bottom-up fashion, information necessary to compute some of the inherited attributes for L-attributed grammars may not be present. Thus, subclasses of L-attributed grammars and additional evaluation techniques for LR parsers have been devised which allow attribution during the parsing stage [AM91, p. 194-201][SIN85]. One example is the subclass of MLR-attributed grammars [AM91, p. 199]; here, synthesized and inherited attribute instances are both maintained on the LR stack together with parser states, with the limitation that inherited attributes must be grouped for all LR items corresponding to a state. Conflicts can arise if the offset between an attribute's symbol and the attribute's parent's symbol is not the same for all LR items in a state, or if inherited attributes can be calculated based on different expressions for different items in a state.

Furthermore, the general problem of grammars of the type:

$$\begin{array}{lcll} A & \to & Ax & \{ \quad A_1.i := f(A_0.i); \quad \} \\ A & \to & y & \{ \quad \dots \quad \} \end{array}$$

is not resolved in this case, since it cannot be known in advance how many applications of f are needed.

Checking that an attribute grammar is one-visit can be done in polynomial time[EF89, p. 852].

Absolutely non-circular grammars

Absolutely non-circular (ANC), also called strongly non-circular (SNC) [DJL88, p. 18ff.][AM91, p. 49ff.] attribute grammars, are a refinement of well-defined AGs that address the problem that the number of elements in *IS-SET*(X) can increase exponentially with the size of the grammar. They are computationally more expensive than one-visit grammars, but they cover more possible AGs.

The approach used is to replace the choice of random graphs from *IS-SET*(X) in Algorithm 2.1 with the single choice of $IS(X)$, a graph fulfilling the function of all graphs in *IS-SET*(X) – more precisely, $IS(X)$ contains an edge (a, b) if one of the graphs in *IS-SET*(X) contains (a, b).

$IS(X)$ can also be defined recursively as:

$$\bigcup_{p \in P} \{DG_p - 0^*[IS(X_{p1}), \ldots, IS(X_{pn_p})] \mid X_{p0} = X\}$$

It can be shown that the time needed to construct the $IS(X)$ graphs is polynomial with respect to the size of the grammar [AM91, p. 51].

$IS(X)$ can be seen as the graph of all possible "i-to-s" dependencies; however, due to the nature of its construction, $IS(X)$ may also contain edges where no corresponding dependencies exist in any of the sets (called "spurious dependencies").

The circularity test must check for each production p whether the graph $DG_p - 0[IS(X_{p1}), \ldots, IS(X_{pn_p})]$ contains a cycle. Circular grammars are always detected by the test, although some non-circular (well-defined) grammars are also falsely identified as circular.

As an example, consider the AG specified by the local dependency graphs in Figure A.3.

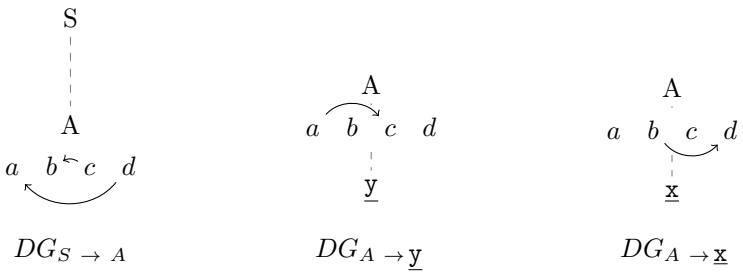


Figure A.3: Local dependency graph attribute specification

The analysis of *IS-SET*(A) reveals two graphs, one for each production, as outlined in Figure A.4 (a). In this case, the aggregated graph $IS(A)$ (Figure A.4 (b)) is equal to the union of both graphs from the *IS-SET*.

However, overlaying $IS(A)$ onto the dependency graph of production $S \to A$ reveals a falsely identified cycle between c, a, d and b (Figure A.4 (c)). The grammar is thus not *absolutely non-circular*.

A possible implementation of an ANC evaluator is using *partitions* [AM91, p. 55] [Nie83]. A *partition of the attributes of a symbol* X is defined as a sequence of disjunct subsets of $A(X)$, starting with a subset of inherited attributes and continuing with alternating subsets of synthesized and inherited attributes. A *partition of a production* p is defined as a sequence of (h_r, A_r), where h_r is the index of a symbol X_{ph_r} in the production ($0 \leq h_r \leq n_p$) and each A_r is a disjunct element of a partition of $A(X_{ph_r})$. An evaluator can make use of $DG_p - 0[IS(X_{p1}), \ldots, IS(X_{pn_p})]$ to construct partitions for each symbol and production. The evaluator then may, for example,

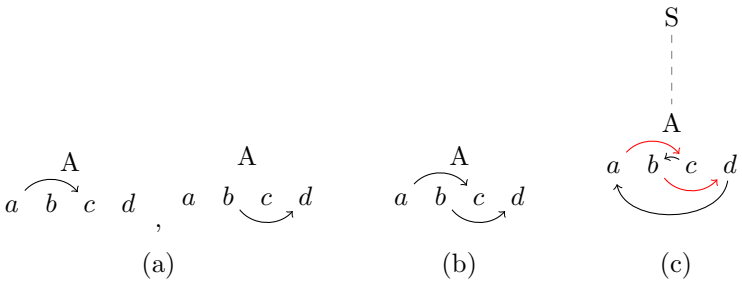


Figure A.4: (a) *IS-SET*(A), (b) $IS(A)$, (c) $DG_{S\to A} - 0[IS(A)]$

be implemented as a set of recursive procedures, one for each partition π of a symbol. Such a procedure would use an alternation of attribute calculations and calls to procedures that handle neighbouring nodes to evaluate all attributes in a valid order.

A similar partitioning system is used for both l-ordered attribute grammars and OAG, which are described below. The partitioning system is discussed in more detail for OAG grammars.

l-ordered attribute grammars

l-ordered attribute grammars are the largest class of attribute grammars that can be statically evaluated, meaning that for each possible node associated with a symbol, the order in which its attribute instances are evaluated is always the same. Therefore, all analyses of the grammar and its symbols can be performed at evaluator construction time.

Formally, this requires the construction of a total order $TO(X)$ for all attributes associated with symbol X. Attribute instances of nodes labelled X must then be evaluated in the order specified by it [AM91, p. 61][Nie83, p. 255]. Specifically, the order must be compatible with all possible dependencies given by $IS(X)$ and $SI(X)$.

Here, $SI(X)$ is defined analogously to $IS(X)$ as a graph containing all edges from graphs in *SI-SET*(X). More precisely, $SI(X)$ can be calculated using a fixed point algorithm from SI and IS graphs based on the following definition [AM91, p. 52]:

$$
\begin{aligned}
SI(Z) &= \text{Empty graph with vertices } A(Z) \\
SI(X) &= \cup_{p\in P}\{DG_p - k^*[SI(X_{p0}), IS(X_{p1}), \ldots, IS(X_{p(k-1)}), \\
&\qquad IS(X_{p(k+1)}), \ldots, IS(X_{pn_p})] \mid 1 \le k \le n_p, X_{pk} = X\} \\
&\quad \text{for all other symbols } X
\end{aligned}
$$

To be able to evaluate all possible trees, $TO(X)$ must exist for all productions p such that $DG_p[TO(X_{p0}), \ldots, TO(X_{pn_p})]$ is cycle free [AM91, p. 61].

In order to construct an evaluator, the set $A(X)$ is split into $m(X)$ disjoint, ordered subsets $\langle A_1(X), \ldots, A_{m(X)}(X)\rangle$ according to $TO(X)$. While performing a visit-based tree-walk, the n-th visit to a node labelled X allows the calculation of the attributes in $A_n(X)$. As the total orders $TO(X)$ for all $X \in V$ are required for determining membership, they can also be used to build the plan for the evaluator.

Whereas absolutely non-circular grammars may have a family of partitions per symbol, only a single partition per symbol exists in l-ordered grammars.

The membership test for l-ordered attribute grammars was shown to be NP-complete in the size of the grammar, so for practical reasons another formalism (OAG) was introduced.

OAG

The OAG formalism [AM91, p.62-64] is based on a graph $DS(X)$ derived from the union of $IS(X)$ and $SI(X)$, in which for each synthesized attribute a and each inherited attribute b of X there is either an edge from a to b or from b to a.

$DS(X)$ can be obtained with the aid of a partition:

$$\pi_X = \langle I_1(X), S_1(X), \ldots, I_{m(X)}(X), S_{m(X)}(X) \rangle$$

of the attributes of X, where $I_k(X) \subseteq I(X)$, $S_k(X) \subseteq S(X)$ and $m(X)$ is the number of elements in the sequence (or of visits to a node with symbol X). This is similar to the partitions described before, with the difference that inherited and synthesized attribute sets are split apart. The partition must fulfill several conditions – a dependency between two attributes a and b hereby denoting an edge (a, b) in $ISSI(X) = IS(X) \cup SI(X)$ in this context:

- Inherited attributes in $I_1(X)$ may not depend on any synthesized attributes of X.
- Inherited attributes in $I_k(X)$, $1 < k \leq m(X)$, may only depend on synthesized attributes from an earlier $S_i(X)$, $i < k$ and may only be dependencies of synthesized attributes from $S_j(X)$, $j \geq k$.
- Synthesized attributes in $S_k(X)$, $1 < k \leq m(X)$, may only depend on inherited attributes from $S_i(X)$, $i \leq k$ and may only be dependencies of inherited attributes from $S_j(X)$, $j > k$.
- Synthesized attributes in $S_{m(X)}$ may not be dependencies of any inherited attributes.

The partition can be calculated in reverse order by iteratively selecting from a set of non-assigned attributes either the synthesized or the inherited attributes with all dependencies met at the certain stage [AM91, p. 63f].

$DS(X)$ is built from the union of $ISSI(X)$ and the set of edges constructed pairwise from adjacent sets of the partition: (a, b) is an edge in this set if $a \in I_k(X)$ and $b \in S_k(X)$ or $a \in S_k(X)$ and $b \in I_{k+1}(X)$. With $DS(X)$ available, the membership test simply involves checking whether for every production p, $DG_p[DS(X_{p0}), \ldots, DS(X_{pn_p})]$ is cycle free.

Once the $DS(X)$ graphs are known, the construction of an evaluator is possible. A "visit graph" VG_p is created for each production, where vertices are either attribute occurrences in $DO(p)$ or vertices $v_{k,i}$ denoting visits to a neighbouring node X_{pk} for a given visit number i [AM91, p. 66]. The edges of VG_p stem from $DG_p[DS(X_{p0}), \ldots, DS(X_{pn_p})]$. Edges in $DG_p[\ldots]$ going from an attribute occurrence (a, p, j) in $DO(p)$ to (b, p, k) in $UO(p)$ or vice versa are mapped to $((a, p, j), v_{k,i})$ and $(v_{k,i}, (a, p, j))$ respectively in VG_p. The visit index i is determined by the set $I_i(X_{pk}) \cup S_i(X_{pk})$ from the partition of X_{pk} that contains the attribute b. Edges from attribute occurrences in $DO(p)$ to other attribute occurrences in $DO(p)$ are simply transferred to VG_p as-is.

The topological sort of VG_p yields a sequence VS_p, which is then split into $m(X_{p0})$ subsequences, each subset starting with a vertex $v_{0,i}$, $1 \leq i \leq m(X_{p0})$ denoting a parent visit. In order for the splitting procedure to work, the topological sort must ensure the first vertex is $v_{0,1}$, even if it other vertices have no dependencies.

Implementing an OAG evaluator using recursive procedures [AM91, p. 67] involves creating one such procedure per symbol and subsequence index $1 \leq i \leq m(X)$, where the operations in the i-th subsequence of the production matching a parse tree node are executed. $v_{k,j}$ vertices are translated into calls to the procedure matching X_{pk} and index j; attribute occurrence vertices are translated into executions of their evaluation rule.

Example. For AG 2.5, there are no "s-to-i" dependencies, resulting in an *SI-SET* containing just the empty graph for each symbol. In this particular case, the $ISSI$ graphs are the union of the respective *IS-SET* graphs and are given in Figure A.5.

OAG partitions can be calculated for each symbol in part:

$$\begin{aligned} \pi_N &= \langle \{\}, \{value\} \rangle \\ \pi_L &= \langle \{position\}, \{value\} \rangle \\ \pi_B &= \langle \{position\}, \{value\} \rangle \end{aligned}$$

From this, it becomes clear that *position* must be evaluated before *value* for all L and B. $ISSI$ equals DS, as the partitions add no additional edges to $ISSI$.

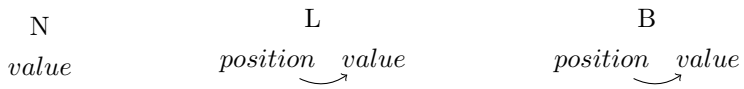


Figure A.5: $ISSI$ (and DS) graphs for AG 2.5

The next step involves creating the visit graphs VG_p and performing a topological sort on each of them to find the visit sequences, VS_p. The extended dependency graphs, $DG_p[DS(X_{p0}), \ldots, DS(X_{p_{n_p}})]$ and resulting visit graphs VG_p are given in Figure A.6.

For example, in graph (b), the attribute instance $B.value$, which belongs to $UO(p)$, is mapped to the corresponding visit node $v_{1,1}$ in VG_p. The edges of VG_p suggest the resulting sequence $\langle v_{0,1}, B.position, v_{1,1}, L_0.value \rangle$. This sequence is equivalent to the seeked evaluation order, as only one parent visit node $v_{0,1}$ exists.

All resulting sequences are given below:

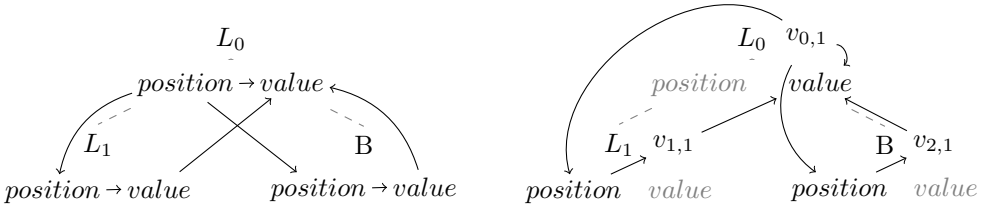
$$VS_{N \to L} = \langle L.position, v_{0,1}, v_{1,1}, N.value \rangle$$
$$VS_{L_0 \to B} = \langle v_{0,1}, B.position, v_{1,1}, L_0.value \rangle$$
$$VS_{L_0 \to L_1 B} = \langle v_{0,1}, B.position, L_1.position, v_{1,1}, v_{2,1}, L_0.value \rangle$$
$$VS_{B \to 0} = \langle v_{0,1}, B.value \rangle$$
$$VS_{B \to 1} = \langle v_{0,1}, B.value \rangle$$

The recursive evaluator can now be constructed. As there is a maximum of one visit per node, the number of resulting functions is also limited to one per node. In this example, each node *node* has properties $node.child_1, \ldots, node.child_{n_p}$ denoting the child nodes and further properties *position* or *value* referencing its attribute instances. The evaluator is started by calling $visit_{N.0}(root)$, where *root* is the root node:

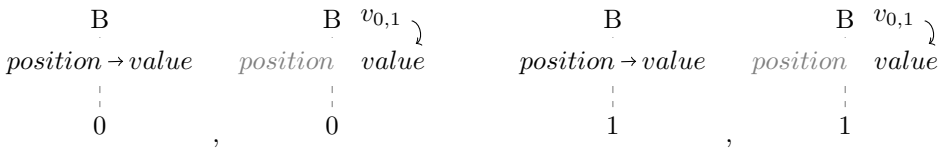
```
function visit_N.0(node):
    if production at node matches N → L:
        evaluate node.child_1.position
        visit_L.0(node.child_1)
        evaluate node.value
```

(a) $N \to L$

(d) $L \to B$



(c) $L \to LB$



(e) $B \to \underline{0}$ (f) $B \to \underline{1}$

Figure A.6: $DG_p[DS(X_{p0}), \ldots, DS(X_{p_{n_p}})]$ and VG_p graphs for AG 2.5

```
function visit_{L.0}(node):
    if production at node matches L → LB:
        evaluate node.child_2.position
        evaluate node.child_1.position
        visit_{L.0}(node.child_1)
        visit_{B.0}(node.child_2)
        evaluate node.value
    if production at node matches L → B:
        evaluate node.child_1.position
        visit_{B.0}(node.child_1)
        evaluate node.value

function visit_{B.0}(node):
    if production at node matches B → 0:
        evaluate node.value
    if production at node matches B → 1:
        evaluate node.value
```

Appendix B

Listings

B.1 AG2010: an attribute evaluator benchmark

parser.y

```
%{
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "helper.h"

void check_list(list_t *list, char *what, char *val) {
  if (!list_contains(list, val)) {
      fprintf(stderr, "%s '%s' undefined\n", what, val);
      exit(3);
  }
}

void check_field(list_t *fields, char *val) {
  check_list(fields, "field", val);
}

void check_variable(list_t *vars, char *val) {
  check_list(vars, "variable", val);
}


void check_var_or_field(list_t *fields, list_t *vars, char *val) {
  if (!list_contains(vars, val) && !list_contains(fields, val)) {
      fprintf(stderr, "variable (or field) '%s' undefined\n", val);
      exit(3);
  }
}

int yyerror() { exit(2); }
extern int yylex(void);

%}

%token _STRUCT _END _METHOD _VAR _IF _THEN _ELSE _WHILE _DO _RETURN
%token _NOT _OR _THIS _SEMICOLON _BRACE _BRACEEND _ASSIGN _DOT _MINUS
%token _STAR _LOWER _EQUALS _COMMA _ID _NUM

%autoinh visible_fields
%autoinh visible_vars

%attributes { long val; } _NUM
%attributes { char *val; } _ID
%attributes { list_t *fields; } Program Methoddef Structdef Structids
%attributes { list_t *vars; } Parids
%attributes {
```

```
47   list_t * %autoinh visible_fields;
48   list_t * %autoinh visible_vars;
49 } Statseq Statement Lexpr Expr Term MinusTerm StarTerm OrTerm Exprs ExprList
50 %attributes {
51 list_t * %autoinh visible_fields;
52 list_t * %autoinh visible_vars; char *val;
53 } Defstatement
54
55 %traversal check
56
57 %%
58
59 Program: Program Methoddef _SEMICOLON
60             @i  { $$.fields = $1.fields; }
61             @i  { $2.fields = $1.fields; /* warning: inherited here */ }
62          | Program Structdef _SEMICOLON
63             @m $$.fields {
64              list_t *intersection = list_intersects($1.fields, $2.fields);
65              if (intersection) {
66                fprintf(stderr, "field␣'%s'␣defined␣in␣two␣structs\n",
67                     intersection->val);
68                exit(3);
69              }
70              $$.fields = list_merge($1.fields, $2.fields);
71            }
72          |
73             @i  { $$.fields = 0; }
74 ;
75
76 Structdef: _STRUCT Structids _END
77             @i  { $$.fields = $2.fields; }
78 ;
79
80 Structids: /* Empty */
81             @i  { $$.fields = 0; }
82            | Structids _ID
83             @m $$.fields { if (list_contains($1.fields, $2.val)) {
84                  fprintf(stderr, "field␣'%s'␣redefined\n", $2.val);
85                  exit(3);
86                }
87                $$.fields = list_add($1.fields, $2.val,
88                                     list_length($1.fields));
89             }
90 ;
91
92 Methoddef: _METHOD _ID              /* Method definition */
93         _BRACE Parids _BRACEEND     /* Parameter definition */
94         Statseq
95         _END
96             @i  { $6.visible_fields = $$.fields; /* warning: inherited here */ }
97             @i  { $6.visible_vars = $4.vars; }
98 ;
99
100 Parids: /* Empty */
101             @i  { $$.vars = 0; }
102         | Parids _ID
103             @m $$.vars { if (list_contains($1.vars, $2.val)) {
104                  fprintf(stderr, "parameter␣'%s'␣redefined\n", $2.val);
105                  exit(3);
106                }
107                $$.vars = list_add($1.vars, $2.val, list_length($1.vars));
108             }
109 ;
110
111 Statseq: /* Empty */
112          | Statement _SEMICOLON Statseq /* autoinh visible_vars */
113          | Defstatement _SEMICOLON Statseq
```

```
114             @m $3.visible_vars { if (list_contains($$.visible_vars, $1.val)) {
115                  fprintf(stderr, "variable '%s' redefined\n", $1.val);
116                  exit(3);
117               }
118               $3.visible_vars = list_add($$.visible_vars, $1.val, 0);
119             }
120 ;
121
122 Defstatement: _VAR _ID _ASSIGN Expr /* Variable definition */
123             @i  { $$.val = $2.val; }
124 ;
125
126 Statement: Lexpr _ASSIGN Expr            /* Assignment */
127          | Expr                          /* Command expression */
128          | _IF Expr _THEN Statseq _END
129          | _IF Expr _THEN Statseq _ELSE Statseq _END
130          | _WHILE Expr _DO Statseq _END  /* WHILE command */
131          | _RETURN Expr
132          ;
133
134 Lexpr: _ID
135           @check {
136             check_var_or_field($$.visible_vars, $$.visible_fields, $1.val);
137           }
138 | Term _DOT _ID
139           @check {
140             check_field($$.visible_fields, $3.val);
141           }
142 ;
143
144 Expr: Term
145 | _NOT Term
146 | Term MinusTerm
147 | Term StarTerm
148 | Term OrTerm
149 | Term _LOWER Term
150 | Term _EQUALS Term
151 ;
152
153 MinusTerm: MinusTerm _MINUS Term | _MINUS Term;
154 StarTerm: StarTerm _STAR Term | _STAR Term;
155 OrTerm: OrTerm _OR Term | _OR Term;
156
157
158 Term: _BRACE Expr _BRACEEND
159     | _NUM
160     | _MINUS _NUM
161     | _THIS
162     | _ID                    /* Variable / Field read */
163           @check {
164             check_var_or_field($$.visible_fields, $$.visible_vars, $1.val);
165           }
166     | Term _DOT _ID          /* Field read */
167           @check {
168             check_field($$.visible_fields, $3.val);
169           }
170
171     | _ID _BRACE Exprs _BRACEEND /* Method call */
172     | Term _DOT _ID _BRACE Exprs _BRACEEND /* Method call */
173     ;
174
175 Exprs: ExprList Expr | ExprList ;
176
177 ExprList: | ExprList Expr _COMMA ;
178
179 %%
```

scanner.l

```
%{
#include <stdio.h>
#include "helper.h"
#include "parser.tab.h"
%}

%option noyywrap

%x comment

%option attributes
%%

struct return _STRUCT;
end return _END;
method return _METHOD;
var return _VAR;
if return _IF;
then return _THEN;
else return _ELSE;
while return _WHILE;
do return _DO;
return return _RETURN;
not return _NOT;
or return _OR;
this return _THIS;

";" return _SEMICOLON;
"(" return _BRACE;
")" return _BRACEEND;
":=" return _ASSIGN;
"." return _DOT;
"-" return _MINUS;
"*" return _STAR;
"<" return _LOWER;
"=" return _EQUALS;
"," return _COMMA;

[a-zA-Z_][a-zA-Z0-9_]* {  $_ID.val = strdup(yytext);  return _ID; }
0x[0-9a-fA-F]+ {  $_NUM.val = strtol(yytext, NULL, 16);  return _NUM; }
[0-9]+ {  $_NUM.val = strtol(yytext, NULL, 10);  return _NUM; }

"/*" BEGIN(comment);
<comment>"*/" BEGIN(0);
<comment><<EOF>> exit(1);
<comment>.

<*>[\n\r\t ]

. exit(1);

%%

int main() {
  int ret = yyparse();
  return ret;
}
```

helper.h

```
#ifndef HELPER_H
#define HELPER_H

typedef struct list {
  char *val;
```

```
6   int pos;
7   struct list *next;
8 } list_t;
9
10 list_t *list_add(list_t *, char *, int);
11 int list_length(list_t *);
12 list_t *list_contains(list_t *, char *);
13 list_t *list_merge(list_t *, list_t *);
14 list_t *list_intersects(list_t *, list_t *);
15
16 #endif
```

helper.c

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include "helper.h"
4
5  list_t *list_add(list_t *in, char *v, int p) {
6    list_t *l = malloc(sizeof(list_t));
7    l->val = v;
8    l->pos = p;
9    l->next = in;
10   return l;
11 }
12
13 int list_length(list_t *in) {
14   int i;
15   for (i = 0; in; in = in->next, i++);
16   return i;
17 }
18
19 list_t *list_contains(list_t *in, char *v) {
20   while (in) {
21     if (!strcmp(v,in->val))
22       return in;
23     in = in->next;
24   }
25   return in;
26 }
27
28 list_t *list_merge(list_t *one, list_t *two) {
29   /* a new list will be allocated */
30   list_t *l = 0;
31   for (; one; l = list_add(l, one->val, one->pos), one = one->next);
32   for (; two; l = list_add(l, two->val, two->pos), two = two->next);
33   return l;
34 }
35
36 list_t *list_intersects(list_t *one, list_t *two) {
37   while (one) {
38     list_t *two_i;
39     for (two_i = two; two_i; two_i = two_i->next) {
40       if (!strcmp(one->val, two_i->val)) {
41         return one;
42       }
43     }
44     one = one->next;
45   }
46   return 0;
47 }
```

Example input (352 lines, 3850 parse tree nodes)

```
method RMme1r()
end;
method i( Wj jK9 ngDNJ1 E7)
  return not 632;
  Wj or (not this);
  return not Wj;
end;
method KLx1()
  var L3 := this or 9;
  L3 := 162;
  L3 := 494 or 0xAa1809;
  not L3;
  var PYjF1 := L3 or 0x90Edb7 or
    -0x4b28 or L3;
  return L3 < (0x1b * L3 * this *
    97736);
  PYjF1 - 0xCbE;
  PYjF1 := -0xC=this;
  L3 := not 321;
  var V := 860129;
  var A0H2 := 0x4d9dBC < this;
  return this * this * 0xE0B * 17;
  var r10 := -29 * 99;
end;
method Nu_oSx()
  return 99 - this;
  while -0xB or this do
    not -0x14;
end;
  var dVHhP := this=0xBd8ca;
  var SW1b := 8357;
  dVHhP * -6675600;
  SW1b := -0xceE;
  SW1b := SW1b * this;
  SW1b := not this;
  var xBBF3R := 0 * this;
  var Myd := not xBBF3R;
end;
method DfNYh()
  return 0536196 < 0x53E;
end;
method xExk( kA esg dnE hN2pD8 Y6W)
  return this or esg or this;
  return -0x2ea * Y6W * 0x5BD;
  0xAD6 * Y6W;
  return (hN2pD8) - dnE - -0xBDDe
    - this;
  hN2pD8 := not kA;
  -51251 - this - -0xB97 - this;
  return 7632444=-8096;
  80=this;
  var lY := -0xE9E or 2545;
  esg := -0xcd0C < 7637472;
end;
method y0M()
  while -0xF8d or this do
    this;
    this < -0128;
    var RDf := -0x29EFc;
    while (not 0xA62)=0xdFCb do
      RDf := RDf * -009349 * RDf *
        074484 * 0xa * this;
      RDf or 1425084 or -0xe258 or
        this or RDf;
      907266 or 34;
      var Giuz := RDf or RDf;
      return this=this;
      not Giuz;
      Giuz := Giuz;
      return not RDf;
      this or -0x9f8E3e;
      RDf := -401;
      var WRWf := -0x3503a;
end;
    var tpD2 := this=-0x7B54E;
    return 0x07E1d;
    return 886;
end;
  var Clzqb := 4=this;
  var s80gDY := -66789 < -0x8;
  not this;
  return s80gDY=0xFEca1;
  return 0xDD3 * Clzqb;
  var g3rU0 := not this;
  while s80gDY - g3rU0 do
    return this * g3rU0;
    Clzqb * -0x45;
    while s80gDY * 52 do
      return this * Clzqb * -875;
      not -0xa58;
      var VeE := 0501902 or 0xe6 or
        this or this or this or
        Clzqb or -0xDa or Clzqb;
      g3rU0 < this;
      s80gDY := -0x3B0fe < s80gDY;
      VeE := not 31232;
      -060 < 0x27c;
      VeE;
      return -0x6e - this - this;
      return this - VeE - 0xf;
      var x := -63 * s80gDY * this;
      return -6699 or -0x4;
      this - this;
      var K4Hl3o := -4069344 *
        (-0x93=0x4fE5);
      s80gDY := this < this;
      if g3rU0=this then
        this=-0x599ba;
        var t := this - -213186 -
            this - this - -3521 -
            this - this;
        this * K4Hl3o;
        -0x0 or x or this;
        t := s80gDY;
        VeE := not s80gDY;
 else
        var W8OUVJ := g3rU0;
        W8OUVJ := s80gDY < this;
        var wsDg := 0xE49 < this;
        -751 * -88826 * VeE;
        0 - -893;
        var W1z3J2 := this or 0x1d
            or 0013913;
        var xR := g3rU0;
        return this - 41668 -
            -0xF82C11 - -0x4;
 end;
end;
    not this;
    while this - -0x7b do
      0xCa2 or 0x7ee or -118586 or 946;
      g3rU0 - 0x4 - this;
      s80gDY := (263=-0x0fBa1)=-0xAa;
      return Clzqb or -08222 or 4;
      Clzqb := -860 - 0x2a1D7
        - (Clzqb < this) - 2 - g3rU0
        - (not g3rU0);
end;
    return 7091935 or g3rU0 or -6;
    var HiL := Clzqb - 0637;
    HiL := 0x3F or -0x8;
    this - this - s80gDY;
    var t := this < s80gDY;
    return -0xcbe3;
    return this or 91240 or this or 7
        or this or -3 or -81708;
    while 0x7B054 or (this < 0xB) do
      t := -070 < -0xE8;
      var arj := HiL - -0x0cE88;
      var Pmv := t or this;
```

```
end;
    var RPF1E := this=g3rU0;
end;
  var R2nPD6 := Clzqb < Clzqb;
  var R_M1g_ := not R2nPD6;
end;
method hWA0mp()
  var U := 454=9889980;
  U := this or this or 948506;
  U := this * U * this * 0xB31;
  U := -0xBb or this or 0024;
  return 0x120E1c < 5;
  U := this;
  var RBOge := -2207468=this;
  var J_JFL := this < this;
  return this or -7652;
  J_JFL := J_JFL < -0xBE332;
  -0xC * this * this * -0x3D;
  return 7465 - 217383;
  var dWMVmP := -7421 - 4846 - J_JFL
    - U - RBOge;
  while -677001=dWMVmP do
    997 < -0xaeEC;
    return -0x6 * -0xf3D081;
    this=this;
    J_JFL := U - -0x1Cd7 - this -
        this - RBOge - this
        - this;
    -0x15 * 0xcF16fA * -372;
    return (this * this * this)=2;
    U := J_JFL * this;
    not 312;
    var H := dWMVmP or this;
    J_JFL := this - -73 - RBOge - this;
end;
  U := 747;
  var I := 0xD < 0x668a9;
  (this * this) or -43 or I or -8219;
  I := -0xB=0xdf294d;
  var s := -0xA * this * this * 87;
  dWMVmP := this or this or J_JFL;
  var q := I * -984996;
  return dWMVmP - 0x5c;
  U := this < -0x55;
  var vdC7p := RBOge;
  0x9 or (this=s) or this or -46021;
  if 0x9a or -0x4fCbA1 or this then
    var A6 := this;
    var B4 := s < -0xC;
    return not q;
    not this;
    dWMVmP := (this < this) * -1722
        * I * 0xa * this *
        -0xC * (-6208591 < -5295456);
    return 0xd2=6683541;
 else
    return this - J_JFL;
    0x6=34306;
    0xe < this;
    return this * vdC7p * this
        * -0xAcC * J_JFL;
    var lGYyn := this;
    var J := -9369 * this * -1293889;
    q < -1014178;
 end;
  -6884392 - this;
  if -6 < -0x4c then
    var jkP := -0xEb - 0xB;
    if -0xa36fc or U then
      this;
 end;
    return s * 106 * J_JFL;
    return this or 0xF;
    vdC7p := I;
    jkP := 872618 - 0x8 - this - (043);
 else
    var fy53G := not -0x846c1b;
    var Su := 0x49 - 9801 - this;
    return this - J_JFL;
    var HSY1 := not this;
 end;
  this or -4686;
  RBOge := this < 2346;
  vdC7p := not this;
end;
method ut( dSx0lA HJE)
  HJE := not HJE;
  var WrsQ := this;
  return (6555=-476626);
  dSx0lA := WrsQ - -0x1C;
  var P3 := WrsQ or dSx0lA;
  HJE := -0x0=91;
end;
method So( UZr MGskiq)
  var wJpkG9 := 0xB0bf99 - this;
  wJpkG9 := wJpkG9=0xa4;
  wJpkG9 := 0x38F74C=UZr;
  if -0xf25E - MGskiq then
    MGskiq := this or UZr;
    return this or MGskiq or this;
    MGskiq := (8533 or 0xc1 or UZr)
        < 0x4;
    this < UZr;
    return 0x6FCe * (0xa65579 *
        -0xa834F);
    var vTZERS := -0x1eBB3 < this;
    UZr := not -91909;
    var lWr := not 0xab5f;
    0xeac3E < this;
    MGskiq := this or -42680;
 end;
  if wJpkG9 then
    0x51333;
    -0x29=327;
    if not this then
      -21895 or 0xfEfc;
      this;
      MGskiq or this or wJpkG9 or
        -9746 or -477;
      return not MGskiq;
      return -748376 or this;
      return (this or this) or wJpkG9
        or -835770 or (-2263
        - wJpkG9) or 041122 or 98;
      this;
      if MGskiq or 9 then
        var t := -18812 * 030;
 end;
      wJpkG9 := this < (this * MGskiq);
      not wJpkG9;
      UZr := 0xfcBf1D;
      return this - this - -0xEe9
        - 0xBd;
      return wJpkG9 < 0x145;
      0x0F < this;
      -0888 - 50;
      var EsWitV := not MGskiq;
      return 49269 < this;
 else
      MGskiq := -846348 * this;
 end;
    wJpkG9 := this;
 else
    var omlL := 8 * UZr * 77;
    return 78602 - omlL - 56513 - this;
    var n := not UZr;
    return MGskiq or ((086 * 44) or
        this);
    omlL - this;
 end;
  wJpkG9 := MGskiq or 0xFD93DC;
  UZr := not 0xaCA;
  var aL := -0xCFeb0 or this;
  aL := UZr - aL;
end;
method S12P( RUfG5 wdg9T2 vU)
```

```
    var gw1 := this * 0xAe7;
    9263601=(6 * vU);
    return this=wdg9T2;
    not -0xF;
  end;
  method FUoP()
    var Iv := 4436428;
    return Iv < -65;
    this < this;
  end;
  method j()
    return -9625 - 156;
  end;
  method _NUy()
    return this=this;
    3925265 or 7847448;
    -0x2B * -0x9d;
    not (this=438603);
    var e := this - -0xbFfAf9;
    return e < 0x2f;
    not e;
    var Y := this=-498748;
    this * this;
    var wX1k := 0xcc=this;
    Y := 914653 or -847484;
    80755 - this - e;
    return not -7160197;
    return -019958 < wX1k;
    this - 0x7Cf - 0xA7cb3e;
    var UoQ := Y=20;
    var aoNRo := 0x1BC4b < wX1k;
    return 653118;
    UoQ := e * 0;
    Y := 83 < 0x8AaB8;
    return 71 or this or aoNRo or
      0x4B354;
    return not -432971;
  end;
  method D()
  end;
  method vRP( GPSEFb CVgd SKy1t)
    var uxXc9 := (CVgd or -0x0E5E22 or
      SKy1t or CVgd or this
      or SKy1t or this) < this;
    return not -4;
  end;
```

Bibliography

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools – Second Edition*. Boston, MA, USA: Pearson Education, Inc., 2007.

[AM91] H. Alblas and B. Melichar, eds. *Attribute Grammars, Applications and Systems*. Springer Verlag, June 1991.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[Bis93] Kurt M. Bischoff. *Ox: An Attribute-Grammar Compiling System based on Yacc, Lex, and C: User Reference Manual*. 1993.

[Cho56] Noam Chomsky. "Three models for the description of language". In: *IRE Transactions on Information Theory* 2 (1956), pp. 113–124.

[Cho59] Noam Chomsky. "On certain formal properties of grammars". In: *Information and Control* 2.2 (1959), pp. 137–167.

[DDPS09] Akim Demaille, Renaud Durlin, Nicolas Pierron, and Benoît Sigoure. "Semantics driven disambiguation: A comparison of different approaches". In: *Electronic Notes in Theoretical Computer Science* 238.5 (October 2009), pp. 101–116.

[DJL88] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars: Definitions, Systems, and Bibliography*. Vol. 323. Lecture Notes in Computer Science. Springer, 1988.

[EF89] Joost Engelfriet and Gilberto Filé. "Passes, sweeps, and visits in attribute grammars". In: *Journal of the ACM* 36 (4 October 1989), pp. 841–869.

[EH07] Torbjörn Ekman and Görel Hedin. "The JastAdd system – modular extensible compiler construction". In: *Science of Computer Programming* 69.1 (2007). Special issue on Experimental Software and Toolkits, pp. 14–26.

[FSF15a] Free Software Foundation. *GNU Bison - The Yacc-compatible Parser Generator*. 2015. URL: `https://www.gnu.org/software/bison/manual/bison.html` (Retrieved on November 13, 2017).

[FSF15b] Free Software Foundation. *GNU Bison*. 2015. URL: `http://ftp.gnu.org/gnu/bison/bison-3.0.4.tar.gz` (Retrieved on January 13, 2018).

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification (3rd Edition)*. Addison-Wesley Professional, 2005.

[Jon03] Simon Peyton Jones, ed. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge, England: Cambridge University Press, 2003.

[JOR75] M. Jazayeri, W. F. Ogden, and W. C. Rounds. “On the complexity of the circularity test for attribute grammars”. In: *POPL ’75: Proceedings of the 2nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. Palo Alto, California: ACM, 1975, pp. 119–129.

[Kah62] A. B. Kahn. “Topological sorting of large networks”. In: *Communications of the ACM* 5.11 (1962), pp. 558–562.

[Knu65] Donald E. Knuth. “On the translation of languages from left to right”. In: *Information and Control* 8.6 (1965), pp. 607–639.

[Knu68] Donald E. Knuth. “Semantics of context-free languages”. In: *Theory of Computing Systems* 2.2 (June 1968), pp. 127–145.

[Knu71] Donald E. Knuth. “Correction: Semantics of context-free languages”. In: *Mathematical Systems Theory* 5.1 (1971), pp. 95–96.

[Knu90] Donald E. Knuth. “The genesis of attribute grammars”. In: *WAGA: Proceedings of the International Conference on Attribute Grammars and their Applications*. Paris, France: Springer-Verlag New York, Inc., 1990, pp. 1–12.

[NGIHK99] Shin Natori, Katsuhiko Gondow, Takashi Imaizumi, Takeshi Hagiwara, and Takuya Katayama. “On eliminating Type 3 circularities of ordered attribute grammars”. In: *Second Workshop on Attribute Grammars and their Applications, WAGA’99*. Ed. by D. Parigot and M. Mernik. Amsterdam, The Netherlands: INRIA Rocquencourt, March 1999, pp. 93–112.

[Nie83] H. Riis Nielson. “Computation sequences: A way to characterize classes of attribute grammars”. In: *Acta Informatica* 19 (1983), pp. 255–268.

[POSIX09] “International Standard - Information technology Portable Operating System Interface (POSIX) Base Specifications, Issue 7”. In: *ISO/IEC/IEEE 9945:2009(E)* (September 2009), pp. 1–3880.

[SIN85] M. Sassa, Harushi Ishizuka, and Ikuo Nakata. “A contribution to LR attributed grammars”. In: *Journal of Information Processing* 8 (1985), pp. 196–206.

[Sip06] Michael Sipser. *Introduction to the Theory of Computation – Second Edition*. Boston, MA, USA: Thomson Course Technology, 2006.

[SKV10] Anthony M. Sloane, Lennart C.L. Kats, and Eelco Visser. “A pure object-oriented embedding of attribute grammars”. In: *Electronic Notes in Theoretical Computer Science* 253.7 (2010). Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009), pp. 205–219.

[Tar72] Robert Tarjan. “Depth-first search and linear graph algorithms”. In: *SIAM Journal on Computing* 1.2 (1972), pp. 146–160.

[Tay02] R. Gregory Taylor. “LL parsing, LR parsing, complexity, and automata”. In: *SIGCSE Bulletin* 34.4 (December 2002), pp. 71–75.

[Tom84] Masaru Tomita. “Parsers for natural language”. In: *COLING84: 10th International Conference on Computational Linguistics*. 1984.

[Tom85] Masaru Tomita. “An efficient context-free parsing algorithm for natural languages”. In: *IJCAI’85 Proceedings of the 9th International Joint Conference on Artificial Intelligence*. Vol. 2. 1985, pp. 756–764.

[Tom87] Masaru Tomita. “An efficient augmented context-free parsing algorithm”. In: *Computational Linguistics* 13.1-2 (1987).

[Wai90] W. M. Waite. “Use of attribute grammars in compiler construction”. In: *Attribute Grammars and their Applications*. Ed. by P. Deransart and M. Jourdan. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 255–265.

[Wu04] Pei-Chi Wu. “On exponential-time completeness of the circularity problem for attribute grammars”. In: *ACM Transactions on Programming Languages and Systems* 26.1 (January 2004), pp. 186–190.