Recursive Adaptable Grammars

by

John N. Shutt

A Thesis
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Master of Science
in
Computer Science
by
John N. Shutt
August 10, 1993

Approved:

Dr. Roy S. Rubinstein, Major Advisor

Dr. Robert E. Kinicki, Head of Department

Emended December 16, 2003

Abstract

Context-Free Grammars (CFGs) are a simple and intuitively appealing formalism for the description of programming languages, but lack the computational power to describe many common language features. Over the past three decades, numerous extensions of the CFG model have been developed. Most of these extensions retain a CFG kernel, and augment it with a distinct facility with greater computational power. However, in all the most powerful CFG extensions, the clarity of the CFG kernel is undermined by the opacity of the more powerful extending facility.

An intuitively appealing strategy for CFG extension is grammar adaptability, the principle that declarations in a program effectively modify the context-free grammar of the programming language. An adaptable grammar is equipped with some formal means for modifying its own CFG kernel. Most previous adaptable grammar formalisms have, unfortunately, failed to realize the potential clarity of this concept. In this thesis, a representative sample of previous grammar formalisms is surveyed. Based on insights gleaned from the survey, a novel adaptable grammar formalism (Recursive Adaptable Grammars, RAGs) is proposed that attempts to combine Turing-power with the conceptual elegance and simplicity of CFGs.

Note: Page numbering

I recommend using section numbers, chapter numbers, etc. in references to this thesis, rather than page numbers. All cross-references in the thesis follow this convention (excepting the table of contents).

Page numbers differ between one-sided and two-sided versions of the thesis, becuase LaTeX puts part and chapter headings on right-hand pages in two-sided copy. Also, pagination sometimes changes slightly as a result of emendations (see the following note).

Note: Emendations

This is an emended version of my Master's Thesis. Differences from the originally submitted thesis have been limited to the correction of flaws in the document. (To *emend* means to free from faults — as distinct from the more general verb *amend*, which means to modify. What qualifies as "freeing from faults" is of course subjective.)

A complete list of emendations by date is available from the author upon request.

Note: Anachronisms.

Since writing the thesis, I've made two significant adjustments to my treatment of RAGs.

1. In the thesis, a terminal or other predicate synthesizes λ; but in my recent work, a terminal or other predicate synthesizes the syntax that it generates (as *echo* does in the thesis).

2. All of my recent writings use the term *adaptive grammar* for what is here called an *adaptable grammar*. The adjective *adaptable* is more usefully reserved for entities that are adapted by an external agency, while entities that adapt themselves are better termed *adaptive*; eventually I had my nose rubbed in this distinction enough times that I reluctantly let general usage override the specific usage I had hoped to retain (fostering much-needed consistency in the subject) from Christiansen's survey article, [Chri 90].

The current document does not reflect these adjustments, because it is limited to emendations, whereas they would be amendments.

John N. Shutt
December 2003

jshutt@cs.wpi.edu
http://www.cs.wpi.edu/~jshutt/

Contents

Chapter 0

Introduction

0.1 The origins of grammar

In the second century B.C., the Greek philosopher Dionysius Thrax wrote a book about the structure of the Greek Language. His book —the *Techne*— classified words into eight parts of speech. This is an example of a *descriptive grammar*, in which correct sentences are described by making statements about them. For the next two thousand years, virtually all grammars were descriptive and traced their lineage directly back to the *Techne*.

Traditional descriptive grammars lacked mathematical rigor. In the 1950s, Noam Chomsky proposed a *generative* approach to grammar, in which correct sentences are generated by an abstract engine. Chomsky proposed a single abstract engine for all languages; each individual language is then defined by a set of rules that tell the engine how to generate all sentences of that language, and no others. This model is much more amenable than its predecessors to formal treatment. Since then, nearly all formal work in syntax —and particularly programming language syntax, with which the current thesis is concerned— has traced its lineage directly back to Chomsky.

0.2 Motivation for the thesis

Chomsky distinguished several classes of grammars, depending on the generality of form of the grammar rules. The most general forms of Chomsky grammars (types 0 and 1) are powerful, but opaque and difficult to work with. The more restricted (types 2 and 3), especially the context-free grammars or CFGs (type 2), are more lucid and easier to use, but not powerful enough to describe most programming languages.

Therefore, numerous extensions of the CFG model have been developed. Most of these extensions retain a CFG kernel, and augment it with a distinct facility that handles context-dependence. The most popular of these extensions are variations on a model introduced by Donald Knuth in the late 1960s, called attribute gram-

mars. Unfortunately, in all the most powerful CFG extensions, including attribute grammars, the clarity of the CFG kernel is undermined by the power of the distinct extending facility.

A concept with some intuitive appeal is that of *grammar adaptability*. The basic principle is that declarations (and information hiding) effectively modify the grammar of the language. For example, when an integer variable **k** is declared, this adds a grammar rule to the effect that **k** is a valid integer expression. An adaptable grammar model implements this principle by providing a formal means for modifying the set of rules.

Since the adaptability principle is a natural way to understand programming languages —including their context-dependent features— intuitively, it is not unreasonable to hope that adaptable grammars could provide a particularly natural way to define the same programming languages formally. A number of adaptable grammar models have been proposed over the past twenty-odd years, and some of them are Turing-powerful (i.e., as powerful as general Chomsky grammars); but most of them lack the descriptive clarity of attribute grammars, let alone CFGs. For descriptive purposes, the most successful to date appears to be a model recently proposed by Henning Christiansen.

Even under Christiansen's model, there are still unresolved difficulties with the adaptable grammar approach. Some modern language features have not yet submitted to elegant description. Also, some of the drawbacks of attribute grammars have been inherited by Christiansen's model, which strongly resembles attribute grammars.

The purpose of this thesis is to develop a Turing-powerful adaptable grammar model that preserves, as much as possible, the elegance and simplicity of CFGs.

0.3 Organization of the thesis

This document has two parts. Part I surveys existing grammar models. Part II proposes a new adaptable grammar model, Recursive Adaptable Grammars (RAGs).

The survey considers both adaptable and nonadaptable models. Descriptive merits and demerits of existing models are analyzed and compared. The survey is extensive, but by no means comprehensive. It is intended to identify and study important concepts and techniques that will be used in the development of the proposal in Part II. Therefore, emphasis is skewed in favor of those models that introduce important concepts or offer useful insights. Chapter 1 is devoted to Chomsky grammars; Chapter 2 surveys nonadaptable extensions of CFGs; Chapter 3 surveys adaptable grammar models.

In Part II, Chapter 4 details the design goals of the proposal and explains how the proposal addresses them. Chapter 5 formally defines the RAG model in its most general form, and Chapter 6 defines some important subclasses of RAGs. Chapter 7 compares RAGs to some of the models surveyed in Part I, and discusses areas for future research.

0.4 Putting subjectivity in its place

All science contains an element of subjectivity, implicit in what questions are asked and what kinds of answers are sought. However, although it cannot be eliminated, subjectivity can be minimized. In writing this thesis, the author has made a deliberate effort to minimize the subjective element. This wasn't easy, since the thesis as a whole, and especially Part I, wrestle with such intangibles as "clarity" and "simplicity". The author is bound to have left some residual bias in the thesis; moreover, the reader may also bring in some bias of his own. To help the reader circumvent these hazards, a few words are in order concerning common sources of trouble.

A major trap in any comparison is that with enough practice, anyone can get used to anything. A researcher who uses just one grammar model intensively for any length of time will probably find it quite easy to use.

One way to combat this effect is to concentrate on the steepness of the learning curve; that is, how much effort would be required for a novice to become facile with the model. This can be difficult to judge (even *with* statistical evidence to support one's claims). The author has pursued an alternative strategy: Rather than considering a student unfamiliar with all of the models surveyed, consider a student fluent with all of the models surveyed. This the author could at least approximate —although, admittedly, with some of the more elaborate models he never got beyond passing familiarity.

Another pitfall to be avoided is the discounting of models based on formal similarity. Two models can be nearly identical in a formal sense, and yet be significantly different in both concept and utility. Most of the nonadaptable models surveyed in Chapter 2 can be thought of as minor variations on each other (see for example §2.3.2 and §2.5.2); yet each has a unique character that sets it apart conceptually from the others.

0.5 Mathematical preliminaries

This section enumerates some basic mathematical concepts and notations that are used throughout both parts of the thesis. It is intended merely to clarify usage that often differs from author to author.

Most of this material is not required for the thesis summary, §0.6 below. It is assumed by Part I. The formal definition of RAGs in Part II additionally requires one-sorted algebra, which is developed rigorously in §5.2.

The set of all nonnegative integers is denoted $\mathbb{N}$. The set of all positive integers is denoted $\mathbb{N}_+$.

The set of all elements of set A with property p is denoted $\{a \in A \mid p(a)\}$, or just $\{a \mid p(a)\}$ when no ambiguity will result (such as when A is understood from context).

Set union is denoted $A \cup B$. Set difference is denoted $A - B = \{a \in A \mid a \notin B\}$. Set intersection is denoted $A \cap B$. The empty set is denoted $\emptyset$. The cardinality of a set is denoted $|A|$.

Ordered tuples are denoted $\langle x_1, \cdots x_n \rangle$ (where $n \in \mathbb{N}$). Cartesian product of two sets is denoted $A \times B = \{\langle a, b \rangle \mid a \in A \text{ and } b \in B\}$. Cartesian product of n sets, where $n \geq 2$, is denoted $A_1 \times A_2 \times \cdots \times A_n = \{\langle a_1, a_2, \cdots a_n \rangle \mid \forall 1 \leq k \leq n,\ a_k \in A_k\}$. The Cartesian product of a set A with itself n times is denoted A^n; thus, $A^2 = A \times A$. Also, $A^0 = \{\langle\rangle\}$, the set containing the 0-tuple.

Notation $A \subseteq B$ means that every element of set A also belongs to set B; that is, $B - A = \emptyset$. Further, $A \subset B$ means that $A \subseteq B$ and $A \neq B$.

The set of all subsets of a given set A is called the *power set* of A and denoted $\mathcal{P}(A)$. For every set A, $\emptyset \in \mathcal{P}(A)$ and $A \in \mathcal{P}(A)$. The set of all finite subsets of a given set A is denoted $\mathcal{P}_\omega(A)$.[1]

Function composition is denoted $f \circ g$ (read "f of g"); here, $(f \circ g)(x) = f(g(x))$.

An *alphabet* is a set of atomic symbols; all alphabets are assumed to be finite unless explicitly stated otherwise. The set of all strings (of finite length) over an alphabet A is denoted A^*. The empty string is denoted λ, and the set of all non-empty strings over A is denoted A^+. Hence, $A^* - \{\lambda\} = A^+$.

String concatenation is denoted $x \cdot y$, or sometimes xy when there is no danger of confusion. The length of a string is denoted $|x|$. A string w concatenated with itself n times, where $n \geq 0$, is denoted w^n. A string w reversed, i.e., "spelled backwards", is denoted w^R. Hence, $w^0 = \lambda$ and $(w^R)^R = w$.

A set of strings is called a *language*. The concatenation of languages X and Y is denoted $X \cdot Y$ or XY, and defined as $XY = \{xy \mid x \in X \text{ and } y \in Y\}$. Given language L over alphabet Z, the language $Z^* - L$ of strings over Z not in L is called the *complement* of L (with respect to Z).

The root node of a tree t is denoted $root(t)$. For each node n of tree t, the number of children of n is its *arity*, denoted $ar(n) \in \mathbb{N}$. The parent of node $n \neq root(t)$ is denoted $parent(n)$.

A Z-labeled tree is a tree with an associated function from the tree nodes into set Z. For each node n, the label of n is denoted $label(n) \in Z$.

0.6 Summary

0.6.1 Survey of grammar models

Chomsky grammars

A Chomsky grammar consists of a vocabulary of symbols and a set of rewrite rules. The vocabulary is partitioned into terminal symbols and nonterminal symbols, and

[1] The notation $\mathcal{P}_\omega(A)$ is borrowed from [Wech 92, p. 47].

one of the nonterminals is designated as the *start symbol*. A rewrite rule has the form $\alpha \rightarrow \beta$, where α and β are strings of symbols, and α contains at least one nonterminal.

Whenever the left side of a rule occurs in a symbol string, the string may be rewritten by replacing that occurrence with the right side of the rule. The notation $x \overset{*}{\underset{G}{\Rightarrow}} y$ signifies that string x can be transformed into string y by repeatedly rewriting it via the rules of grammar G; y is said to be derived from x under G, and binary relation $\overset{*}{\underset{G}{\Rightarrow}}$ is called the derivation relation for G. The set of all terminal strings that can be derived from the start symbol of G is called the language generated by G.

Several interesting classes of Chomsky grammars are formed by placing restrictions on the form of rules. For purposes of this thesis, the most important such class is that of *context-free grammars*, or CFGs. A CFG is a Chomsky grammar such that the left side of every rule has length one. Thus, each context-free rewrite action expands a nonterminal symbol into a string of symbols. This uncomplicated expansion of nonterminals is easy to grasp intuitively, and lends itself readily to depiction via a parse tree. Because of these conceptual advantages, CFGs are very commonly used in the description of programming languages, even though the class of context-free languages (i.e., languages generated by CFGs) excludes some important programming language features, such as lexical scope and static typing.

There is a remarkably close analogy between Chomsky grammars and Turing machines, if the latter are considered, not as a computational model per se, but as a *grammar model*. Developing this analogy provides broad insights into the concept of grammatical derivation, and serves as good preparation for some of the more elaborate grammar models discussed later in the survey.

A Turing machine consists of a finite-state controller, and an infinite storage tape accessed through a read/write head. The controller determines what machine actions are possible from a given machine configuration (combination of state, tape contents, and head position), much as the rule set of a Chomsky grammar determines what rewrites are possible from a given symbol string. The overall behavior of the machine is described by a computation relation that transforms machine configurations by multiple actions, just as a Chomsky derivation relation transforms strings by multiple rewrites. There is a designated internal halt state from which no further action is possible, just as a Chomsky grammar cannot further rewrite a terminal string.

The analogy extends to subclasses of Turing machines as well. To each of the commonly used subclasses of Chomsky grammars, there is a standard automaton model of equal power; and although it is not usually done, each of these automaton models can be formulated as a subclass of Turing machines, by restricting the kinds of actions that can be permitted by the controller —just as the corresponding classes of grammars restrict the kinds of rewrites that can be permitted by the rule set.

Attribute grammars

In order to reconcile the desire to use CFGs with the need to rigorously define context-dependent programming language features, various extensions of the CFG model have

been developed. Most of these extensions retain a CFG kernel, and augment it with a distinct facility that handles context-dependence. The most popular such extension is Knuth's *attribute grammars*.

As originally envisioned by Knuth, an attribute grammar consists of a CFG and an attribute system. Terminal strings are processed in two stages. First, the CFG is used to parse the terminal string. Then, the nodes in the parse tree are decorated with attribute name/value pairs. The attribute system associates with each CFG rule a group of formulae called *semantic rules*, which specify the dependencies between the attributes of a parent node and the attributes of its children when the parent's nonterminal is expanded via that CFG rule.

Although more complicated than CFG derivation, this arrangement is still moderately intuitive, because context-dependent information is distributed only by propagation through the links of the context-free parse tree, which can be visualized.

A distinction is made between *inherited* attributes, whose values propagate down the parse tree, and *synthesized* attributes, whose values propagate up the parse tree. Knuth identified this distinction as key to his development of the model. However, a number of similar grammar models have been developed in which the distinction is downplayed or eliminated altogether. A feature common to such models is that they integrate attribute evaluation into the derivation relation, rather than treating it as a separate process as Knuth did.

One such model is *extended attribute grammars*. Here, each nonterminal occurrence is accompanied by its attribute values, listed in some predetermined order, with arrows up or down ($\uparrow$ or $\downarrow$) indicating which attributes are synthesized and which inherited. The set of rules of a CFG are replaced by a set of *rule forms*, in which polynomial expressions fill the attribute positions. The rewrite rules available for derivation are constructed by making every possible substitution of values for variables in the rule forms.

The distinction between inherited and synthesized attributes in an extended attribute grammar has no technical significance. There is no longer any definite direction of computation; derivations that use the wrong attribute values will simply reach an impasse without deriving a terminal string. This trend is carried further by *definite clause grammars*, which are a closely related grammar model based on principles from logic programming. In a definite clause grammar, context-dependent language features are described by the explicit imposition of logical constraints on a CFG; thus, definite clause grammars reject the inheritance/synthesis distinction even in concept.

A recent addition to the family of attribute grammar variant models is *higher-order attribute grammars*. This model introduces special *nonterminal attributes*, which appear as nonterminals in the context-free rules, but are not expanded during parsing; instead, they are evaluated as attributes whose values are new parse-tree branches. These branches are then attached to the existing parse tree, and the attribute evaluation process is expanded to include the attributes of the nodes in the new branches.

This evaluation of attributes created by evaluation of attributes may be viewed as a form of recursion.

Other nonadaptable grammars

W-grammars, sometimes called two-level grammars, were developed by Aad van Wijngaarden a few years before Knuth's work. The first level of a W-grammar is a context-free meta-grammar, whose "nonterminals" are called meta-variables, and whose "terminals" are symbols of the second level. The second level is defined by a set of *rule schemata*, which are context-free rules that may contain embedded meta-variables. The rewrite rules available for derivation are constructed by making every possible substitution of symbols for meta-variables in the rule schemata.

The meta-variables and rule schemata of a W-grammar are analogous to the variables and rule forms of an extended attribute grammar. However, W-grammars are more complicated in concept because they entail three fundamentally distinct kinds of derivation: meta-variables to symbols, rule schemata to rules, and nonterminals to terminals.

There are also many other grammar models that are less commonly used for programming language definition. One large family of grammar models increase the power of CFGs by placing restrictions on the order in which the context-free rules can be applied during derivation; a drawback of this approach is that the context-dependent factors are invisible in the parse tree. Indexed grammars are another less commonly used model, in which the application of otherwise context-free rules can be regulated by means of index lists, which are attached to the nonterminals much like inherited attributes in an extended attribute grammar. The use of index lists is reminiscent of, though less versatile than, the language attribute of Christiansen grammars (see below).

Adaptable grammars

One way of understanding declarations in a program is to view them as adding new grammatical constructs to the language. Based on this observation, some researchers have been led to the notion of an *adaptable grammar model*, i.e., a model in which the rule set of a grammar can vary during parsing. More precisely, in this survey a grammar model is considered adaptable iff it provides for the explicit manipulation of rule sets.

There are two major classes of adaptable grammar models, depending on whether rule sets are "manipulated" by imperative or declarative means. Most adaptable grammar models use the imperative approach. Three such imperative models are surveyed in this thesis. However, imperative grammar adaptation has an inherent weakness. The only purpose served by the rule set at any given moment is to determine the outcome of the next decision to be made by the parser. Therefore, the grammar designer must know exactly what decisions will be made by the parser,

and in what order; otherwise, there is no way of knowing what adaptations to make. But this means that the model is dependent on the parsing algorithm, which greatly increases the conceptual complexity of the model.

Although several apparently declarative adaptable models were encountered while researching the survey, the only one with a fully developed formalism is that first proposed by Henning Christiansen in the mid-1980s, called here Christiansen grammars. As presented in his earlier works, Christiansen's model is a modified form of extended attribute grammars. In his later works, he uses an equivalent formulation based on definite clause grammars. The earlier form is used here, because it more nearly resembles the notation developed for recursive adaptable grammars in Part II of this thesis, and because the inheritance/synthesis distinction provides useful insights into both models.

Structurally, a Christiansen grammar is just an extended attribute grammar such that the first attribute of every nonterminal is inherited and has as its value a Christiansen grammar; this special attribute is called the *language attribute*. Whenever a nonterminal is expanded during derivation, the rewrite rule is drawn from the language attribute of the nonterminal. The only significance of the initial Christiansen grammar is that it becomes the language attribute of the root node of the parse tree.

Observe that each language attribute value, and therefore each grammar adaptation, in a Christiansen grammar is localized to a particular branch of the parse tree. A fringe benefit of this approach is that block-structured scope is supported gracefully, which is generally not true in imperative adaptable grammars.

The decision to make the language attribute inherited rather than synthesized is not at all arbitrary. In any parse tree, the language attribute of a nonterminal says nothing —at least, nothing remotely finite— about the ancestors or siblings of the node; on the contrary, it is entirely concerned in a clearly delineated way with constraining the relation between the node and its *children*.

0.6.2 Recursive adaptable grammars

Design principles

The broad purpose of the design for the RAG model was to construct, if possible, a grammar formalism incorporating adaptability without compromising the formal and conceptual elegance of CFGs. The survey of existing models was undertaken primarily in order to isolate factors that contribute to the relative clarity or opacity of a grammar model. By combining the broad purpose of the design with insights gleaned from the survey, a number of more specific design principles were identified.

- Rules should have the one-to-many structure characteristic of CFG rules. This one-to-many structure was identified as essential to the conceptual clarity of the CFG model.

- Context-dependent information should be distributed locally within the rules, as in attribute grammars and their variants. This appeared to be the most intuitive approach to context-dependence in the survey.

- There should be only one computational mechanism in the model. Models, such as attribute grammars, that combine a CFG kernel with a more powerful augmenting facility tend to rely heavily on the augmenting facility, and lose the conceptual benefits of the CFG.

- The one computational mechanism should be an elementary derivation step relation, preferably based on simple rewriting. This is a characteristic of CFGs, and its violation coincides with many of the conceptual problems experienced by other models surveyed. The need to bring this elementary derivation step to bear on nontrivial expression evaluation led directly to the characteristic "recursive" nature of recursive adaptable grammars. (See below.)

- The expansion of each branch of a parse tree should depend entirely on a *single* value occurring at the parent node of that branch. This is intended to simplify the model. Christiansen grammars do not meet this criterion, because the nonterminal symbol and language attribute of the parent node are both required for expansion of a branch.

- The domains of syntactic, meta-syntactic, and semantic values should all be identical, so that any value in the combined domain is equally capable of playing any of these three roles. In Christiansen grammars, the three roles are technically separate: terminals are syntax, nonterminals and language attributes are meta-syntax, and all other attributes are semantics. Nevertheless, the concept of unifying the three roles is foreshadowed by some of his sample grammars.

The RAG model

The combined syntactic/meta-syntactic/semantic domain of a RAG G is called the *answer algebra* of G, and denoted A_G; elements of A_G are called *answers*. Formally, A_G is a *one-sorted algebra*, defined by a set of operators and a set of identity laws. Each RAG also has a designated subalgebra of A_G called the *terminal algebra*, denoted T_G, whose elements are called *terminals*. Elements of the difference set $A_G - T_G$ are called *nonterminals*.

Typically, the terminal algebra is just the domain of strings over an alphabet, together with the binary concatenation operator and the usual identity laws for strings: concatenation is associative, λ is a left and right identity.

An *unbound rule* is a construction of the form

$$\langle v_0, e_0 \rangle \quad \rightarrow \quad t_0 \langle e_1, v_1 \rangle t_1 \cdots \langle e_{n-1}, v_{n-1} \rangle t_{n-1} \langle e_n, v_n \rangle t_n$$

where $n \geq 0$, the v_k are distinct variables, the t_k are answers (usually, but not necessarily, terminal), and the e_k are polynomials in variables $v_0, \cdots v_n$.

The ordered pairs in an unbound rule, $\langle v_0, e_0 \rangle$ and all $\langle e_k, v_k \rangle$, are called *unbound pairs*. The left-hand component of each pair plays the same meta-syntactic role as a Christiansen language attribute. The t_k on the right side of the rule act as syntax. The right-hand component of each pair is usually interpreted as capturing the semantics of the terminal string generated by that pair.

The domain $\mathcal{R}_G$ of unbound rules is related to the domain of answers by means of a *rule function*, $\rho_G : A_G \to \mathcal{P}_\omega(\mathcal{R}_G)$. When binding the variables of an unbound rule r as above, the leftmost variable v_0 must be bound to some answer $a \in A_G$ such that $r \in \rho_G(a)$. This is equivalent to Christiansen's requirement that the rule used to expand each parent node of the parse tree must be drawn from the language attribute of that node.

The rule function determines the meta-syntactic sense of each answer. Nonterminal constants, i.e., nonterminal operators of arity zero, have a fixed meta-syntactic sense, just as nonterminals in CFGs. The unique nature of the rule function comes from the way it handles terminals, and non-constant nonterminal operators.

For all terminals t in a RAG G, $\rho_G(t) = \{\langle v_0, \lambda \rangle \to t\}$. In other words, when a terminal is used as meta-syntax, it generates the singleton language containing itself, and "synthesizes" semantic value λ.

For each nonterminal non-constant operator σ of arity $n \geq 1$ in a RAG G, there is a *rule equation* that defines $\rho_G(\sigma(\cdots))$ as a function of the arguments to σ. Here is a canonical example.

$$\rho(a_1 \sqcup a_2) \quad = \quad \left\{ \begin{array}{lcl} \langle v_0, v_1 \rangle & \to & \langle a_1, v_1 \rangle \\ \langle v_0, v_1 \rangle & \to & \langle a_2, v_1 \rangle \end{array} \right\}$$

This rule equation defines the meta-syntactic behavior of the binary infix $\sqcup$ operator. It says, in essence, that for all answers a_1 and a_2, the answer $a_1 \sqcup a_2$ generates the union of the languages generated by the two arguments. This particular operator is called the *union operator*.

The design goal of nontrivial expression evaluation is addressed by the introduction of a special binary operator called the *query operator*, written $a\!:\!b$, that denotes a semantic value associated by meta-syntactic value a with syntactic value b. That is,

Proposition 0.1 For all $a, b, c \in A_G$, $a\!:\!b \overset{*}{\underset{G'}{\Rightarrow}} c$ iff $\langle a, c \rangle \overset{*}{\underset{G'}{\Rightarrow}} b$. $\square$

Technically, the query operator does not belong to the answer algebra. It belongs to a larger algebra Q_G, called the *query algebra*, of which A_G is a proper subalgebra. The polynomials in an unbound rule are expressions over the query algebra, hence they can use the query operator.

In Proposition 0.1, the two derivations are, so to speak, working in opposite directions. The first derivation may be thought of as constructing a parse tree from the bottom up, while the second derivation constructs the same tree from the top down.

Since the derivation relation $\overset{*}{\underset{G}{\Rightarrow}}$ is merely the transitive closure of a step relation $\underset{G}{\Rightarrow}$, the step relation must somehow implement this kind of bi-directional behavior.

This is accomplished in the RAG model by introducing yet another special operator, this time a unary operator called the *inverse operator* and written $\overline{a}$. The inverse operator does not belong to the query algebra; it belongs to the *configuration algebra*, C_G, of which Q_G is a proper subalgebra. The configuration algebra also includes the binary *pairing operator*, $\langle , \rangle$, which is used to form the ordered pairs in unbound rules. Formally, the derivation step is a relation between configurations.

Algebraically, the inverse of an answer is that answer, i.e., $a = \overline{a}$, while the inverse of the inverse of any configuration is that configuration, i.e., $c = \overline{\overline{c}}$. The inverse operator figures prominently —and often subtly— in most of the axioms that define the derivation step relation; the most basic example is:

Axiom 0.1 For all $c, d \in C_G$, $c \underset{G}{\Rightarrow} d$ iff $\overline{d} \underset{G}{\Rightarrow} \overline{c}$. $\square$

Thus, for example, for all $a \in A_G$, $c \overset{*}{\underset{G}{\Rightarrow}} a$ implies $\overline{a} = a \overset{*}{\underset{G}{\Rightarrow}} \overline{c}$.

Proposition 0.1 falls out as a relatively straightforward consequence of these derivation step axioms. Another consequence of the same axioms is that, for all answers $a, b \in A_G$, $a \overset{*}{\underset{G}{\Rightarrow}} b$ implies $a = b$. Despite the simplicity of the latter result, its proof is highly nontrivial.

Subclasses of RAGs

The formal definition of the RAG model is more general than is ordinarily necessary or desirable. A number of simplifying assumptions are normally made. The terminal algebra is assumed to be an algebra of strings over an alphabet. A number of standard nonterminal operators are assumed, such as the union operator mentioned earlier. These measures are largely cosmetic. But in addition, two formal well-behavedness properties are required; these are: *strong stepwise decidability*, and *strong answer-encapsulation*.

A RAG is strongly stepwise decidable iff (1) the rule function can be computed by a Turing machine, and (2) given any two polynomials over the configuration algebra, a Turing machine can decide whether they are identically equal. Neither of these conditions is guaranteed by the general definition of RAGs. If both conditions hold for a RAG G, then it is possible to construct a Turing machine that decides, for any two configurations c and d, whether or not $c \underset{G}{\Rightarrow} d$.

Intuitively, weak answer-encapsulation means that when two answers have the same "interface", they cannot be told apart by formal means. The notion of interface is expressed, for a RAG G, by an equivalence relation $\equiv_G$ over A_G, such that $a \equiv_G b$ iff a and b generate the same syntax, and assign equivalent semantics to that syntax. More precisely, for all $x, x', y, z \in A_G$, if $x \equiv_G x'$ and $x : y \overset{*}{\underset{G}{\Rightarrow}} z$ then there exists $z' \in A_G$ such that $z \equiv_G z'$ and $x' : y \overset{*}{\underset{G}{\Rightarrow}} z'$. An operator on A_G is *G-safe* iff it preserves answer-equivalence; G is weakly answer-encapsulated iff all of the operators on A_G are G-safe (that is, formally, iff $\equiv_G$ is a congruence on A_G).

Strong answer-encapsulation extends the concept of operator safeness to classes of RAGs. Rule equations, such as the one given earlier for the union operator, are used to define the generic behavior of a set of operators, independent of any specific RAG that satisfies those equations. A collection of rule equations $\mathcal{F}$, called a *framework*, is *safe* iff, for all possible RAGs G that satisfy $\mathcal{F}$, and all operators σ defined by $\mathcal{F}$, σ is guaranteed to be G-safe. A RAG G is strongly answer-encapsulated iff there exists some safe framework that is satisfied by G and defines all of the operators on A_G.

All of the RAG frameworks used in this thesis are safe. However, the general problem of determining whether a framework is safe seems to be rather complicated, with large numbers of exceptions and special cases. In practice, it is convenient to use a combination of general and specific proof techniques. Most of the frameworks in the thesis are covered by a small group of moderately general framework safety theorems; only five operators are not covered, and these are easily handled on an individual basis.

It may be noted, in passing, that there exist rule equations that are inherently *un*safe, in the sense that no framework containing such an equation can possibly be safe.

A RAG is *well-behaved* iff it is both strongly stepwise decidable and strongly answer-encapsulated. The well-behaved RAG model is Turing-powerful. Formally, to every Turing machine M that accepts language L and computes partial function f_M, there exists a well-behaved RAG G that generates language L, and assigns to each string $w \in L$ the semantic value $f_M(w)$.

0.6.3 Comments

In his survey article on the subject [Chri 90], Christiansen identified six potential problems with grammar adaptability.

- Block-oriented lexical scope is not handled gracefully by any of the imperative adaptable grammar models surveyed. In Christiansen grammars, and in RAGs, block scope is a natural consequence of the way meta-syntactic information is locally distributed through the parse tree.

- In addition to immediately adding a new rule to the grammar, a declaration may entail the addition of new rules at some later point in the program. Christiansen's handling of this situation introduces an explicit distinction between variables, meta-variables, meta-meta-variables, and so on. The RAG model does not require such a distinction.

- For two of the problems (visibility and recursive declarations), Christiansen offered no solution that he himself considered satisfactory; his implemented parser addresses both problems through the introduction of logical operators that are alien to the conceptual framework of his grammar model. The RAG

model can handle both problems within its normal framework; whether it admits elegant solutions has yet to be determined.

- All of the adaptable grammar models surveyed (including Christiansen's) have some difficulty expressing the constraint that a variable cannot be declared more than once in the same block. A RAG solution to this problem is given in §4.3.

- Christiansen suggests that the use of an adaptable grammar may prohibit a language parser from constructing meaningful diagnostic messages for syntax errors. Based on previous experience with a parser for a grammar with limited adaptability, the current author is more optimistic than Christiansen on this point.

For the immediate future, implementation of a RAG-based parser generator should be a research priority, as the availability of such a system would facilitate practical, and to some extent theoretical, investigation of the RAG model. There are also a variety of purely theoretical conjectures regarding the RAG model that could be explored, although at the moment there seems no specific reason to assign high priority to them.

In the longer term, the author hopes to use the RAG model to develop a solid formal basis on which to analyze the abstraction mechanisms of arbitrary programming languages. Here, *abstraction* is meant in its most general and comprehensive sense applicable to programming, encompassing the entire evolutionary process by which new program entities are created and old ones are hidden. The criterion of strong answer-encapsulation was developed with this long-term goal specifically in mind.

Part I

Survey of grammar models

Chapter 1

Chomsky grammars

Chomsky grammars were proposed by Noam Chomsky in the 1950s. Chomsky presented them as a model of the way human beings comprehend natural languages. Their application to programming languages began with the development of ALGOL 60. (Actually, the ALGOL 60 Reports [Naur 60, Naur 63] used a variant notation, BNF, which was developed independently by John Backus. However, Chomsky's formulation is generally preferred for theoretical purposes.)

There is, in fact, a "Chomsky hierarchy" of four nested classes of Chomsky grammars, each defined by a constraint on the form of its grammar rules. They are sometimes numbered from *type 0* (the most general class) to *type 3* (the most limited class). A good treatment of the four classes of the Chomsky hierarchy occurs in [Hopc 79]; a shorter treatment occurs in [Clea 76]. Primary sources for the Chomsky hierarchy are [Chom 56, Chom 59].

1.1 Introduction

Formally, a Chomsky grammar has four parts: (1) a set of symbols called *terminals*, (2) a set of symbols called *nonterminals*, (3) a *start symbol*, which is one of the nonterminals, and (4) a set of *rewrite rules*.

Terminals are the vocabulary of the language defined by the grammar. For example, a grammar of the English language might have terminals such as **tree**, **trees**, **book**, **eat**, **sleep**, **slept**, **white**, and so on.

Nonterminals may be thought of (in the simpler cases, at least) as representing classes of terminals and classes of phrases made up of terminals. For example, a grammar of English might have nonterminals such as *noun*, *verb*, *sentence*, *subject*, *predicate*, and so on.

Rewrite rules take the form $\alpha \rightarrow \beta$, where α and β are strings of symbols. The basic idea is that you derive one string (of symbols) from another by replacing α with β somewhere in the string. For example, suppose there is a rewrite rule that says

prepositional-phrase $\rightarrow$ *preposition noun-phrase*

This means that anywhere the nonterminal on the left (*prepositional-phrase*) occurs, it can be replaced by the sequence of two nonterminals on the right (*preposition noun-phrase*). Thus, the four-symbol string

noun prepositional-phrase verb noun

may be rewritten as

noun preposition noun-phrase verb noun

A *derivation* is a series of rewrites that transform one string x into another string y. The notation for this relation is $x \overset{*}{\underset{G}{\Rightarrow}} y$, where G is the name of the grammar whose rewrite rules are being used. In particular, a terminal string w is said to be *generated* by a grammar if it can be derived from the start symbol of that grammar.

The following section formalizes the concepts outlined above. There will be more to say about the conceptual nature of Chomsky grammars once a rigorous framework has been established.

1.2 Type 0 Chomsky grammars

The most general class of Chomsky grammars are variously called type 0 grammars, unrestricted grammars, phrase-structure grammars, or semi-Thue systems. The name "phrase-structure grammar" should probably be avoided, as it seems to have been used by some authors to refer instead to the class of type 2 grammars. The name "unrestricted grammar" will be avoided because of possible confusion over the subject of §2.4. The name "type 0 grammar" is preferred here.

Every author seems to have a slightly different way to define Chomsky grammars. The treatment here was chosen for parallelism with subsequent treatments of other models.

Definition 1.1 A *vocabulary* is an ordered pair $V = \langle Z, T \rangle$, where $Z \supset T$ are finite sets of symbols. The elements of T are called *terminals* of V, those of $Z - T$ are called *nonterminals* of V, and those of Z are called simply *symbols* of V.

The set of all terminals of an arbitrary vocabulary V is denoted T_V; of all nonterminals of V is denoted N_V; and of all symbols of V, Z_V. □

Definition 1.2 A *rule* over a vocabulary V is a structure of the form

$$\alpha \rightarrow \beta$$

where $\alpha \in Z_V^* N_V Z_V^*$ and $\beta \in Z_V^*$. (That is, α and β are strings of symbols of V, and α contains at least one nonterminal.)

α is called the *left (hand) side*, or *lhs*, of the rule, and β the *right (hand) side*, or *rhs* of the rule. If $r = (\alpha \to \beta)$, one writes:

$$\begin{aligned} lhs(r) &= \alpha \\ rhs(r) &= \beta \end{aligned}$$

The length of the right side of r is called the *arity* of r, and denoted $ar(r) = |rhs(r)|$. r is said to be *terminal* iff $rhs(r) \in T_V^*$.

The domain of all rules over V is denoted $\mathcal{R}_V$. □

Some authors require only that the left side of every rule be nonempty, rather than that it contain a nonterminal. The same languages can be described either way. The definition above follows [Lewi 81] in requiring a nonterminal, because the weaker restriction would not imply Theorem 1.2 (below).

Definition 1.3 A *Chomsky grammar* is a four-tuple $G = \langle Z, T, R, s \rangle$, where $V = \langle Z, T \rangle$ is a vocabulary, $R \in \mathcal{P}_\omega(\mathcal{R}_V)$ is a finite set of rules, and $s \in N_V$.

R is called the *rule set* of G, and s the *start symbol* of G. Subscript G may be used in place of subscript V anywhere that subscript would have occurred; thus, set T_G of terminals of G, set N_G of nonterminals of G, and so on. Also, s_G denotes the start symbol of G. □

Definition 1.4 The *derivation step relation* for a Chomsky grammar G is the minimal binary relation $\underset{G}{\Rightarrow}$ over Z_G^* satisfying the following axioms.

Axiom 1.1 If $(\alpha \to \beta) \in R_G$ then $\alpha \underset{G}{\Rightarrow} \beta$. ◇

Axiom 1.2 If $\alpha \underset{G}{\Rightarrow} \beta$ and $\omega \in Z_G^*$ then $\alpha\omega \underset{G}{\Rightarrow} \beta\omega$ and $\omega\alpha \underset{G}{\Rightarrow} \omega\beta$. ◇

The *derivation relation* for a grammar G is the reflexive transitive closure of $\underset{G}{\Rightarrow}$, denoted $\underset{G}{\overset{*}{\Rightarrow}}$. The transitive closure of $\underset{G}{\Rightarrow}$ is denoted $\underset{G}{\overset{+}{\Rightarrow}}$.

A *derivation* over G is a sequence $\omega_0, \cdots \omega_n$, where $n \geq 0$, $\omega_k \in Z_G^*$ for all $0 \leq k \leq n$, and $\omega_{k-1} \underset{G}{\Rightarrow} \omega_k$ for all $1 \leq k \leq n$. Derivations are normally written as chains of derivation step relations; thus,

$$\omega_0 \underset{G}{\Rightarrow} \omega_1 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} \omega_n$$

n is called the *length* of the derivation. A derivation of length zero is *trivial*. □

The above definition of the derivation step relation is essentially inductive. This is not the usual treatment for Chomsky grammars. Most authors use a single axiom to define all derivation steps directly in terms of the rule set R_G; the usual axiom is proven here as the following theorem.

Theorem 1.1 Suppose G is a Chomsky grammar, and $\alpha, \beta \in Z_G^*$. Then $\alpha \underset{G}{\Rightarrow} \beta$ iff there exist $a, b, x, y \in Z_G^*$ such that $\alpha = xay$, $\beta = xby$, and $(a \to b) \in R_G$. □

Proof. Suppose $a, b, x, y \in Z_G^*$ and $(a \to b) \in R_G$. By Axiom 1.1, $a \underset{G}{\Rightarrow} b$. By Axiom 1.2, $ay \underset{G}{\Rightarrow} by$, and $xay \underset{G}{\Rightarrow} xby$.

On the other hand, suppose $\alpha \underset{G}{\Rightarrow} \beta$. Since the derivation step relation is minimal, $\alpha \underset{G}{\Rightarrow} \beta$ must follow from Axiom 1.1 by a finite (nonnegative) number of applications of Axiom 1.2. It is sufficient to observe that all derivation steps deduced in this way must have the requisite form. □

Definition 1.5 Given a Chomsky grammar G and symbol $z \in Z_G$, the *language generated* by z in G is $L_G(z) = \{w \in T_G^* \mid z \overset{*}{\underset{G}{\Rightarrow}} w\}$. The language accepted by G is $L(G) = L_G(s_G)$. □

There are not many well-behavedness properties for Chomsky type 0 grammars. However, the following theorem is one such. It says, in essence, that a derivation cannot continue further once it generates a terminal string. As noted earlier, the theorem would not have been valid if the weaker definition of type 0 grammar rules had been used.

Theorem 1.2 If G is a Chomsky grammar, $\tau \in T_G^*$, and $\omega \overset{+}{\underset{G}{\Rightarrow}} \tau$, then $\omega \notin T_G^*$. □

Proof. By Definition 1.2, the left side of every rule contains a nonterminal. Therefore, by Theorem 1.1, the left side of every nontrivial derivation contains a nonterminal. □

1.3 Subclasses of Chomsky grammars

Type 1 Chomsky grammars are more commonly known as context-sensitive grammars (CSGs); type 2 are called context-free grammars (CFGs); and type 3 are called regular grammars. Languages that can be generated by these classes of grammars are similarly called context-sensitive languages (CSLs), context-free languages (CFLs), and regular languages, respectively. Languages that can be generated by Chomsky type 0 grammars are called *recursively enumerable* languages.

Definition 1.6 Suppose G is a grammar, and $r \in R_G$. Then r is *context-sensitive* iff $|rhs(r)| \geq |lhs(r)|$ (that is, iff the right side of r is at least as long as the left).

Further, r is *normal context-sensitive* iff $r = (\alpha n \beta \to \alpha \nu \beta)$ for some $\alpha \in Z_G^*$, $n \in N_G$, $\beta \in Z_G^*$, and $\nu \in Z_G^+$.

A grammar G is context-sensitive iff all rules $r \in R_G$ are context-sensitive, and normal context-sensitive iff all $r \in R_G$ are normal context-sensitive. □

Some authors use the stronger (normal) condition as the definition of context-sensitivity. Any grammar satisfying the weaker condition can be rewritten to satisfy the stronger; and any grammar satisfying the stronger condition already satisfies the weaker. The above definition follows [Hopc 79] by using the weaker condition but introducing the stronger as a normal form. [Clea 76] uses the stronger condition, and introduces the weaker condition under the name "monotone grammars".

Note that, by either definition, the right side of a context-sensitive rule cannot be the empty string (λ). Consequently, λ cannot belong to any CSL.

Definition 1.7 Suppose G is a grammar, and $r \in R_G$. Then r is *context-free* iff $lhs(r) \in N_G$.

A grammar G is context-free iff all $r \in R_G$ are context-free.

A derivation over a context-free grammar is *leftmost* iff every derivation step rewrites the leftmost nonterminal in the string. □

Note that CFLs, unlike the "superclass" of CSLs, can contain the empty string. This detail complicates results dealing with the relationship between the two classes.

Definition 1.8 Suppose G is a grammar, and $r \in R_G$. Then r is *left-linear* iff $lhs(r) \in N_G$ and $rhs(r) \in (N_G \cup \{\lambda\})T_G^*$; that is, iff r is context-free and its right side is either a string of terminals, or a nonterminal followed by a string of terminals. r is *right-linear* iff $lhs(r) \in N_G$ and $rhs(r) \in T_G^*(N_G \cup \{\lambda\})$.

A grammar G is left-linear iff all $r \in R_G$ are left-linear, right-linear iff all $r \in R_G$ are right-linear, and *regular* iff G is either left-linear or right-linear. □

A language is generated by some left-linear grammar iff it is generated by some right-linear grammar. Note that a grammar with both left-linear and right-linear rules is neither left-linear nor right-linear (but does belong to the larger class of *linear* grammars, which generate a proper superset of the regular languages).

Some authors further require for left- and right-linearity that there be exactly one terminal on the right side of each rule. This stronger condition is not quite equivalent to the weaker, because it excludes the empty string (λ) from all regular languages. The definition here follows [Hopc 79].

1.4 Turing machines

The use of abstract automata, such as Turing machines, in this thesis is almost nil; those needs could be satisfied by one or two paragraphs in the section on mathematical preliminaries (§0.5). Rather, the purpose of the current section is to *survey* the Turing machine model, just as if it were a formal grammar model.

This is not as farfetched as it might sound. Computation by Turing machines is much the same as derivation by Chomsky grammars, except that the state of a

computation has more internal structure than that of a derivation. A comparison of the two models provides useful perspective on both, and also on some of the more elaborate models to be surveyed later (especially those of §3.2).

1.4.1 Introduction

In concept, a Turing machine is an abstract processor with a finite instruction set, a finite amount of internal memory (the machine state), and an infinite supply of external memory in the form of a tape accessed by a single read/write head. The tape is divided into cells, each of which contains one symbol from a special *tape alphabet*. The instructions are: (1) Move the head left one cell. (2) Move the head right one cell. (3) Write a specified symbol in the cell currently under the head.

At the beginning of computation, all of the tape cells contain the reserved symbol #, called the blank symbol, except for an input string written in the cells immediately to the right of the head. The behavior of the machine is defined by a transition function that specifies, for each possible combination of the symbol under the head and the internal state of the machine, what instruction to perform and what internal state to enter when doing so. The computation is complete when (and if) the machine enters a special state called the *halt state*; the output of the computation is the nonblank part of the tape contents when the machine halts.

In general, the transition function may be multivalued. When the transition function yields multiple values (instruction/state pairs), the machine chooses one at random. A machine is called *deterministic* if its transition function never has multiple values.

1.4.2 Definitions

As with Chomsky grammars, different authors define Turing machines differently. The definition here is fairly orthodox in its approach; technical details were chosen for simplicity, and for parallelism with the development of Chomsky grammars.

Note, however, that the treatment of less powerful automata is distinctly *un*orthodox. Although equivalent to the usual treatment, it is much more uniform across the whole family of automata. Again, it was chosen for simplicity and for parallelism with the development of the Chomsky hierarchy. (For conventional treatments, see [Hopc 79, Lewi 81].)

Definition 1.9 A *nondeterministic Turing machine* (NTM) is a five-tuple $M = \langle Q, Z, T, \delta, q_0 \rangle$, where

- Q is a finite set not including the reserved state h.
- Z is an alphabet including the reserved symbols # and \$ but not including the reserved symbols L and R.
- $T \subset Z$ is an alphabet not including symbols # and \$.

- $\delta : Q \times Z \to \mathcal{P}_\omega((Q \cup \{h\}) \times (Z \cup \{L, R\}))$.
- $q_0 \in Q$.

Further, M is a *deterministic* Turing machine (DTM) iff for all $\langle q, z\rangle \in Q \times Z$, $|\delta(q, z)| \leq 1$.

The five components are called the *state set*, *tape alphabet*, *input alphabet*, *transition function*, and *start state*, respectively. # is called the *blank symbol*, and \$ is called the *end marker*. □

Definition 1.10 Suppose $M = \langle Q, Z, T, \delta, q_0\rangle$ is an NTM. A *configuration* of M is a four-tuple $\langle q, \omega_L, z, \omega_R\rangle$ where $q \in Q \cup \{h\}$, $\omega_L \in Z^*$, $z \in Z$, and $\omega_R \in Z^*$. The *computation step relation* for M is the minimal binary relation $\vdash_M$ over the domain of configurations of M satisfying the following axioms. Throughout the axioms, assume that all q, x, z, and ω, with or without subscripts or primes, are universally quantified over Q, $Q \cup \{h\}$, Z, and Z^* respectively.

Axiom 1.3 If $\langle x, z'\rangle \in \delta(q, z)$, then $\langle q, \omega_L, z, \omega_R\rangle \vdash_M \langle x, \omega_L, z', \omega_R\rangle$. ⋄

Axiom 1.4 If $\langle x, L\rangle \in \delta(q, z)$, then $\langle q, \omega_L z_L, z, \omega_R\rangle \vdash_M \langle x, \omega_L, z_L, z\omega_R\rangle$, and $\langle q, \lambda, z, \omega\rangle \vdash_M \langle x, \lambda, \#, z\omega\rangle$. ⋄

Axiom 1.5 If $\langle x, R\rangle \in \delta(q, z)$, then $\langle q, \omega_L, z, z_R\omega_R\rangle \vdash_M \langle x, \omega_L z, z_R, \omega_R\rangle$, and $\langle q, \omega, z, \lambda\rangle \vdash_M \langle x, \omega z, \#, \lambda\rangle$. ⋄

The *computation relation* for an NTM M is the reflexive transitive closure of $\vdash_M$, denoted $\vdash_M^*$.

A *computation* by M is a sequence $c_0, \cdots c_n$ of configurations of M, where $n \geq 0$, and $c_{k-1} \vdash_M c_k$ for all $1 \leq k \leq n$. Computations are normally written as a chain of computation step relations; thus,

$$c_0 \vdash_M c_1 \vdash_M \cdots \vdash_M c_n$$

□

Definition 1.11 Suppose $M = \langle Q, Z, T, \delta, q_0\rangle$ is an NTM. An input string $w \in T^*$ is *accepted* by M iff $\langle q_0, \lambda, \$, w\$\rangle \vdash_M^* \langle h, \omega_L, z, \omega_R\rangle$ for some $\omega_L z \omega_R \in \{\#\}^*$. The *language accepted* by M, denoted $L(M)$, is the set of all strings accepted by M. □

A Turing machine intended to accept a language is sometimes called a *Turing acceptor*. A deterministic Turing machine can also be thought of as computing a function. In that case, the machine is called a *Turing transducer*. Usually, Turing machines are assumed to be acceptors unless otherwise stated.

Definition 1.12 Suppose $M = \langle Q, Z, T, \delta, q_0 \rangle$ is a DTM. Further suppose that M halts on input $w \in T^*$. Then the *output* of M on input w, denoted $f_M(w)$, is the unique shortest string $x \in Z^*$ such that

$$\langle q_0, \lambda, \$, w\$ \rangle \vdash_M^* \langle h, \omega_L, z, \omega_R \rangle$$

and $\omega_L z \omega_R = \#^i x \#^j$ for some $i, j \in \mathbb{N}$.

The partial function f_M is said to be *computed* by M. A function is *partial recursive* iff it is computed by some DTM. □

Note that the function computed by a Turing machine is partial. There is no guarantee, in general, that a Turing machine will ever halt on a given input. This leads to the following distinction.

Definition 1.13 A language L is *recursively enumerable* (or *Turing-acceptable*) iff there exists an NTM M such that $L = L(M)$. Further, L is *recursive* (or *Turing-decidable*) iff there exists a DTM M such that $L = L(M)$ and M halts on all input.[1] □

Just as the Chomsky hierarchy was constructed by placing various restrictions on the form of the rule sets of Chomsky grammars, some important classes of abstract automata may be defined by placing various restrictions on the transition functions of Turing machines. The particular classes of automata defined below correspond very closely to the Chomsky hierarchy; see §1.5.

Definition 1.14 A *linear bounded automaton* (LBA) is a Turing machine $M = \langle Q, Z, T, \delta, q_0 \rangle$ such that, for all $q \in Q$, $\delta(q, \#) \subseteq (Q \cup \{h\}) \times \{L, R\}$. □

In other words, an LBA is a Turing machine that never overwrites a blank. This limits the storage capacity of the tape to the length of the input string.

Definition 1.15 A *finite state automaton* (FSA) is a Turing machine $M = \langle Q, Z, T, \delta, q_0 \rangle$ such that, for all $\langle q, z \rangle \in Q \times Z$, $\delta(q, z) \subseteq (Q \cup \{h\}) \times (Z \cup \{R\})$. □

In other words, an FSA is a Turing machine that never moves its head left. This prevents the machine from effectively using any of the tape as temporary storage. In more conventional definitions, a finite state acceptor doesn't even *have* a tape.

Definition 1.16 A *pushdown automaton* (PDA) is a Turing machine $M = \langle Q, Z, T, \delta, q_0 \rangle$ such that, for all $\langle q, z \rangle \in Q \times (Z - \{\#\})$, $\delta(q, z) \subseteq (Q \cup \{h\}) \times (Z \cup \{R\})$. □

[1] For an explanation of this terminology, and a thorough formal treatment of the concepts, see [Roge 67].

In other words, a PDA is a Turing machine that cannot move its head left unless the symbol under the head is a blank. What this actually does to the machine is to turn the infinite tape into a stack (hence the word "pushdown" in the name). The reasoning here is just a bit tricky.

While performing its computation, the machine cannot make effective use of the part of the tape that lies to the right of the input string, because it would have to erase all of its working storage every time it needed to consult the original input. Therefore working storage is effectively limited to the left half of the tape. But access to this working storage is strictly on a last-in-first-out basis, because in order to access any given symbol stored there one must first erase all of the more recently stored symbols.

It should be remarked, since the question is relevant to §3.2, that pushdown transducers and finite state transducers are defined somewhat differently from the corresponding automata. Because the ability of the machine to write on the tape is so severely limited, it is not convenient to require that the output be written on the tape. Instead, the transition function is extended by associating an "output string" with each transition. The output of a computation is the concatenation of the output strings for all the computation steps.

1.4.3 Comments

The transition function of a Turing machine plays the same role as the rule set of a Chomsky grammar. In fact, the rule set of a Chomsky grammar could be thought of as a multi-valued function, mapping each string $\alpha \in (Z_G^* - T_G^*)$ into a set of strings $\rho_G(\alpha) = \{\beta \mid (\alpha \to \beta) \in R_G\}$. Each subclass of Chomsky grammars would then be defined by a restriction on ρ_G of very much the same form as Definitions 1.14–1.16. The use of a (somewhat different) rule function ρ_G is key to the RAG model proposed in Part II of this thesis.

Note that in any conventional definition of a pushdown automaton, the stack-oriented aspect is built explicitly into the machine. The infinite tape is replaced by an infinite stack, and the move-right/move-left instructions are replaced by push/pop. Such a treatment greatly clarifies the stack-oriented nature of the machine. On the other hand, Definition 1.16 offers insights that the conventional definition does not, regarding the relationship between PDAs and the other classes of automata defined above.

1.5 Computational power

This section recounts a number of results dealing with the relative power of the various classes of automata and grammars defined in the preceding sections. Proofs are omitted. Most of the material is fairly commonplace; for example, see [Hopc 79]. Citations are given in the text for less familiar results.

Theorem 1.3 Let L be a language.

- There exists an NFSA that accepts L iff there exists a regular grammar that generates L.
- There exists an NPDA that accepts L iff there exists a context-free grammar that generates L.
- There exists an NLBA that accepts L iff there exists a context-sensitive grammar that generates $L - \{\lambda\}$.
- There exists an NTM that accepts L iff there exists a type 0 grammar that generates L. □

The power of deterministic automata is another matter. The computational consequences of determinism have long been understood for finite automata, pushdown automata, and Turing machines; the consequences for LBAs are not known.

Definition 1.17 A regular language is *deterministic* iff it is accepted by some DFSA. Similarly, a CFL is *deterministic* iff it is accepted by some DPDA; a CSL, iff accepted by some DLBA; a recursively enumerable language, iff accepted by some DTM. □

Note that, since by definition every deterministic automaton is also a nondeterministic machine of the same class, it follows that every deterministic regular language is a regular language, and so on.

(Words prefixing the Chomsky classes of grammars and languages will always be written out, rather than being integrated into the standard acronyms. Thus, an NLBA *recognizes a* nondeterministic CSL*. Otherwise, confusion could arise from the acronym* NCSG*, with the* N *standing ambiguously for either "nondeterministic" or "normal".)*

Theorem 1.4 The following relations hold between deterministic and nondeterministic Chomsky language classes:

- Every regular language is deterministic.
- There exist context-free languages that are not deterministic.
- Every recursively enumerable language is deterministic. □

Theorem 1.5 The following containment relations hold between language classes:

- Every regular language is deterministic context-free.
- Every context-free language that does not include the empty string is deterministic context-sensitive.
- Every context-sensitive language is recursive.

- Every recursive language is recursively enumerable. □

Other than the details about determinism, the above class inclusions all follow trivially from the definitions of the classes. It is less obvious that the class inclusions are proper. For completeness, the determinism result for CFLs is repeated below.

Theorem 1.6 The following relations hold between language classes:

- There exist languages that are not recursively enumerable.
- There exist recursively enumerable languages that are not recursive.
- There exist recursive languages that are not context-sensitive.
- There exist deterministic context-sensitive languages that are not context-free.
- There exist context-free languages that are not deterministic context-free.
- There exist deterministic context-free languages that are not regular. □

Another property of language classes that is sometimes of interest is whether or not they are closed under complementation. That is, if $\mathcal{C}$ is a class of languages over an alphabet Z, does $L \in \mathcal{C}$ imply $(Z^* - L) \in \mathcal{C}$? The result for recursively enumerable languages in the following theorem is one of the most basic theorems in the theory of computation, sometimes called the *halting problem*. The result for nondeterministic CSLs is a recent development [Imme 88].

Theorem 1.7 The following language classes are *not* closed under complementation.

- The recursively enumerable languages.
- The context-free languages.

The following language classes *are* closed under complementation.

- The recursive languages.
- The context-sensitive languages.
- The deterministic context-sensitive languages.
- The deterministic context-free languages.
- The regular languages. □

When confronted with a class of languages (such as the class of all languages generated by some class of non-Chomsky grammars), it is useful to be able to determine how the class relates to the Chomsky language classes. To aid in this task, as well as illustrate some of the character of the various classes, here are some commonly used examples of languages that differentiate Chomsky classes.

Theorem 1.8 Suppose Z is an alphabet, and $\{a, b, c\} \subseteq Z$.

- The following language is deterministic context-free, but not regular.

  $$\{a^n b^n \mid n \in \mathbb{N}\}$$

  That is, the set of all strings consisting of some number of a's followed by the same number of b's.

- The following language is context-free, but not deterministic.

  $$\{ww^R \mid w \in Z^*\}$$

  That is, the set of all strings consisting of a string w followed by the reverse of w. This language may also be characterized as the set of all palindromes of even length. (The set of all palindromes of arbitrary length is also context-free and not deterministic.)

- The following languages are context-sensitive, but not context-free.

  $$\{a^n b^n c^n \mid n \in \mathbb{N}_+\}$$

  $$\{www \mid w \in Z^*\}$$

□

There are no trivial examples of languages that differentiate the larger classes; as a rule of thumb, any language that cannot be recognized by a linear bounded automaton is a mess. Typically, inclusion or non-inclusion of languages at the high end of the hierarchy is addressed in terms of the construction of automata. (For example, see [Wegb 70].)

1.6 Understanding CFGs

CFGs are the most commonly used class of Chomsky grammars, especially if one includes the many non-Chomsky grammar models that are essentially augmentations of CFGs. This section explores the conceptual properties of CFGs, in an attempt to identify the basis for their popularity. To begin with, two complete examples of Chomsky grammars are introduced and compared.

Example 1.1 Let $G_{1.1}$ be the following context-free grammar.

$$\begin{aligned} S &\rightarrow \lambda \\ S &\rightarrow aSb \end{aligned}$$

Here, by convention, upper case letters are nonterminals and lower case letters are terminals. Unless otherwise stated, S is the start symbol. The language generated by the above grammar is $L(G_{1.1}) = \{a^n b^n \mid n \in \mathbb{N}\}$, which is deterministic context-free, but not regular.

As an illustration of how the grammar works, here is a derivation (in fact, the only possible derivation) for the string *aaabbb*. Unless otherwise stated, derivations are assumed to begin with the start symbol of the grammar under consideration.

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$

□

Example 1.2 Let $G_{1.2}$ be the following context-sensitive grammar.

$$\begin{array}{rcl} S & \rightarrow & aBc \\ S & \rightarrow & aSBc \\ cB & \rightarrow & Bc \\ aB & \rightarrow & ab \\ bB & \rightarrow & bb \end{array}$$

The language generated by the above grammar is $L(G_{1.2}) = \{a^n b^n c^n \mid n \in \mathbb{N}_+\}$, which is not context-free.

The first two rules are substantially the same as those of Example 1.1; they generate strings of the form $a^n(Bc)^n$. The third rule (which is not, by the way, in normal form) allows the B's in these strings to migrate to the left, generating strings of the form $a^n B^n c^n$. The last two rules convert the nonterminal B's into terminal b's; but a given B can only be converted once it has migrated as far to the left as possible.

As an illustration, here are derivations for the strings *abc*, *aabbcc*, and *aaabbbccc*. Note that *abc* is the only string in the language whose derivation by this grammar is unique.

$$S \Rightarrow aBc \Rightarrow abc$$

$$S \Rightarrow aSBc \Rightarrow aaBcBc \Rightarrow aaBBcc \Rightarrow aabBcc \Rightarrow aabbcc$$

$$\begin{aligned} S &\Rightarrow aSBc \Rightarrow aaSBcBc \Rightarrow aaaBcBcBc \Rightarrow aaaBBccBc \Rightarrow aaaBBcBcc \\ &\Rightarrow aaaBBBccc \Rightarrow aaabBBccc \Rightarrow aaabbBccc \Rightarrow aaabbbccc \end{aligned}$$

□

Two questions will be considered: First, what conceptual role do nonterminals play in these examples? —and second, what conceptual role do grammar rules play in these examples? Recall from §1.1 the grammar rule

prepositional-phrase → *preposition* *noun-phrase*

Here, the answers to both questions are fairly clear. Each nonterminal represents a class of phrases; and the rule as a whole signifies that a prepositional phrase may consist of a preposition followed by a noun phrase.

This interpretation of nonterminals and rules applies with equal facility to any context-free grammar. In Example 1.1, S stands for a class of phrases (specifically, all phrases of the form $a^n b^n$ where $n \geq 0$). The first rule signifies that the empty string is such a phrase; the second signifies that an S phrase may consist of an a, followed by an S phrase, followed by b.

However, this scheme doesn't work for Example 1.2. Nonterminal B does apparently correspond to a class of phrases —namely, $\{b\}$. At first glance, one might be tempted to identify S with the class of all phrases $a^n b^n c^n$ with $n \geq 1$. But is this really a valid identification? Consider the second rule,

$$S \rightarrow aSBc$$

Certainly the S on the right side of this rule could be expanded to a string $a^n b^n c^n$, but that string would then have to be broken up in order to complete the derivation. For example, here is another derivation for $aabbcc$, with the right-hand S and its derivative symbols underlined.

$$S \Rightarrow a\underline{S}Bc \Rightarrow a\underline{aBc}Bc \Rightarrow a\underline{abc}Bc \Rightarrow a\underline{ab}B\underline{c}c \Rightarrow a\underline{ab}b\underline{c}c$$

In this case, the difficulty with interpretation of nonterminals is symptomatic of difficulty with interpretation of grammar rules. Following the context-free heuristic, a production $S \rightarrow aSBc$ ought to mean that an S phrase may consist of an a, followed by an S phrase, followed by a B phrase, followed by a c. But as just seen, this is not so. The right-hand S phrase isn't followed by the B phrase at all; rather, they intermingle.

Since the problem seems to arise from the leftward migration of the B's, one might suspect that the difficulties arise only because the CSG is not in normal form. However, this is not the case. Normal CSGs have much the same problem.

Example 1.3 Let $G_{1.3}$ be the following normal context-sensitive grammar.

$$\begin{aligned} S &\rightarrow aBC \\ S &\rightarrow aSBC \\ CB &\rightarrow CD \\ CD &\rightarrow BD \\ BD &\rightarrow BC \\ aB &\rightarrow ab \\ bB &\rightarrow bb \\ C &\rightarrow c \end{aligned}$$

The language generated by the above grammar is $L(G_{1.3}) = L(G_{1.2}) = \{a^n b^n c^n \mid n \in \mathbb{N}_+\}$.

Compare this grammar to the previous example. $G_{1.2}$ used two nonterminals and five rules; $G_{1.3}$ uses four nonterminals and eight rules. But aside from increased

verbosity, they are pretty much the same grammar. The only difference is that, technically, $G_{1.3}$ is normal and $G_{1.2}$ is not.

The technicality is achieved by replacing the non-normal third rule of Example 1.2,

$$cB \rightarrow Bc$$

with a suite of rules,

$$\begin{aligned} CB &\rightarrow CD \\ CD &\rightarrow BD \\ BD &\rightarrow BC \end{aligned}$$

which accomplishes the same thing without violating the definition of normal form.

Here are derivations for the strings *abc* and *aabbcc*. (Derivations for *aaabbbccc* have now gotten rather too long for useful illustration.)

$$S \Rightarrow aBC \Rightarrow aBc \Rightarrow abc$$

$$\begin{aligned} S \Rightarrow aSBC \Rightarrow aaBCBC \Rightarrow aabCBC \Rightarrow aabCBc \Rightarrow aabCDc \Rightarrow aabBDc \\ \Rightarrow aabBCc \Rightarrow aabbCc \Rightarrow aabbcc \end{aligned}$$

□

Introducing normal form has not solved the problem. If anything, the conceptual role of nonterminals is more obscure. If the suite of rules that replaced $cB \rightarrow Bc$ are viewed as a unit, then nothing has changed; but now, one might also try to associate each nonterminal with a particular group of positions in the string rather than with a particular class of phrase:

$$S \Rightarrow a\underline{S}BC \Rightarrow a\underline{aBC}BC \stackrel{+}{\Rightarrow} a\underline{abC}Bc \stackrel{+}{\Rightarrow} a\underline{abB}Cc \stackrel{+}{\Rightarrow} a\underline{abb}cc$$

Under this interpretation, the conceptual connection between the right-hand S and the terminal string *aabbcc* has become still more remote (or at least, less readily apparent).

To bring out the problem in even greater relief, consider the new nonterminal D. What is its conceptual role? Interpreted as a string position, it is a single element in a rather primitive distributed computation. As a phrase, it could expand to b or c; again, its significance in the overall scheme is lost. Perhaps the most informative explanation of its role is that it serves as a place-holder for the distribution of contextual information through the string; but in addition to introducing a whole new kind of conceptual role, this reemphasizes the conceptual problems of the other nonterminals.

1.7 Parse trees

For formal purposes, the process of deriving terminal strings from a grammar was described via string transformations (specifically, via the derivation step relation). However, in practice it is more common to describe the derivation process by means of a *parse tree*.

Definition 1.18 Suppose t is a $(Z_G \cup \{\lambda\})$-labeled tree for some grammar G. Then for each node n of t with $label(n) \in N_G$, $rule(n) \in \mathcal{R}_G$ denotes the grammar rule

$$label(n) \quad \rightarrow \quad label(n_1) \cdots label(n_{ar(n)})$$

where $n_1 \cdots n_{ar(n)}$ are the children of n in order from left to right.

A *parse tree* over a grammar G is a $(Z_G \cup \{\lambda\})$-labeled tree t such that all the following conditions hold.

- $label(root(t)) = s_G$.
- For all nodes n of t, $ar(n) > 0$ iff $label(n) \in N_G$.
- For all nodes n of t, $ar(n) > 0$ implies $rule(n) \in R_G$.
- For all nodes n of t, $label(n) = \lambda$ implies $ar(parent(n)) = 1$. □

For example, the derivation

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$

from Example 1.1 is represented as a parse tree in Figure 1.1. In this particular case, there is only one way to derive the string, hence the derivation and the parse tree contain exactly the same information, although the tree diagram might have some advantages as a communication medium. But in general, parse trees and formal derivations do not carry the same information content.

Example 1.4 Let $G_{1.4}$ be the following CFG.

$$\begin{array}{lcl} S & \rightarrow & E \\ E & \rightarrow & E \text{ “+” } E \\ E & \rightarrow & E \text{ “}*\text{” } E \\ E & \rightarrow & \text{“}\mathbf{x}\text{” } | \text{ “}\mathbf{y}\text{” } | \text{ “}\mathbf{z}\text{” } | \text{ “}\mathbf{t}\text{”} \end{array}$$

Here, the notation $\alpha \rightarrow \beta_1 \mid \beta_2 \mid \cdots$ is shorthand (borrowed from the BNF notation of the ALGOL 60 Reports) for a suite of grammar rules with the same left-hand side: $\alpha \rightarrow \beta_1$, $\alpha \rightarrow \beta_2$, $\cdots$. The boldface symbols delimited by double quotes are “string literals”, terminal symbols chosen for their textual appearance. String literals will always be typeset as boldface text, but the delimiting double quotes may be omitted when no confusion will result.

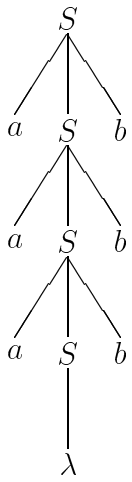

Figure 1.1: Parse tree for *aaabbb*

Let $w \in L(G_{1.4})$ be the following terminal string.

$$\mathbf{x} * \mathbf{y} + \mathbf{z} * \mathbf{t}$$

There are a great many different ways to derive w from the start symbol S, depending on the order in which the grammar rules are applied. To simplify this scenario, disregard the terminal rule applications. There are four nonterminal rule applications: first $S \to E$, and then one for each of the three binary operators in w. The latter three may be applied in any order (left-middle-right, left-right-middle, etc.), giving a total of six different ways to derive w. The corresponding derivations (omitting terminal rules) are:

$$\begin{array}{lccccccccc}
\text{(a) L-M-R:} & S & \Rightarrow & E & \Rightarrow & E*E & \Rightarrow & E*E+E & \Rightarrow & E*E+E*E \\
\text{(b) L-R-M:} & S & \Rightarrow & E & \Rightarrow & E*E & \Rightarrow & E*E*E & \Rightarrow & E*E+E*E \\
\text{(c) M-L-R:} & S & \Rightarrow & E & \Rightarrow & E+E & \Rightarrow & E*E+E & \Rightarrow & E*E+E*E \\
\text{(d) M-R-L:} & S & \Rightarrow & E & \Rightarrow & E+E & \Rightarrow & E+E*E & \Rightarrow & E*E+E*E \\
\text{(e) R-L-M:} & S & \Rightarrow & E & \Rightarrow & E*E & \Rightarrow & E*E*E & \Rightarrow & E*E+E*E \\
\text{(f) R-M-L:} & S & \Rightarrow & E & \Rightarrow & E*E & \Rightarrow & E+E*E & \Rightarrow & E*E+E*E
\end{array}$$

On the other hand, Figure 1.2 shows the corresponding parse trees.

To a first approximation, the parse trees are a more abstract representation of the derivation process. No simplifying assumptions were made in constructing the parse trees, because parse trees are already independent of the order of terminal rule application. Also, trees (c) and (d) are identical, while derivations (c) and (d) are distinct.

However, the relationship between derivations and parse trees is not strictly many-to-one. Trees (b) and (e) are distinct, while derivations (b) and (e) are identi-

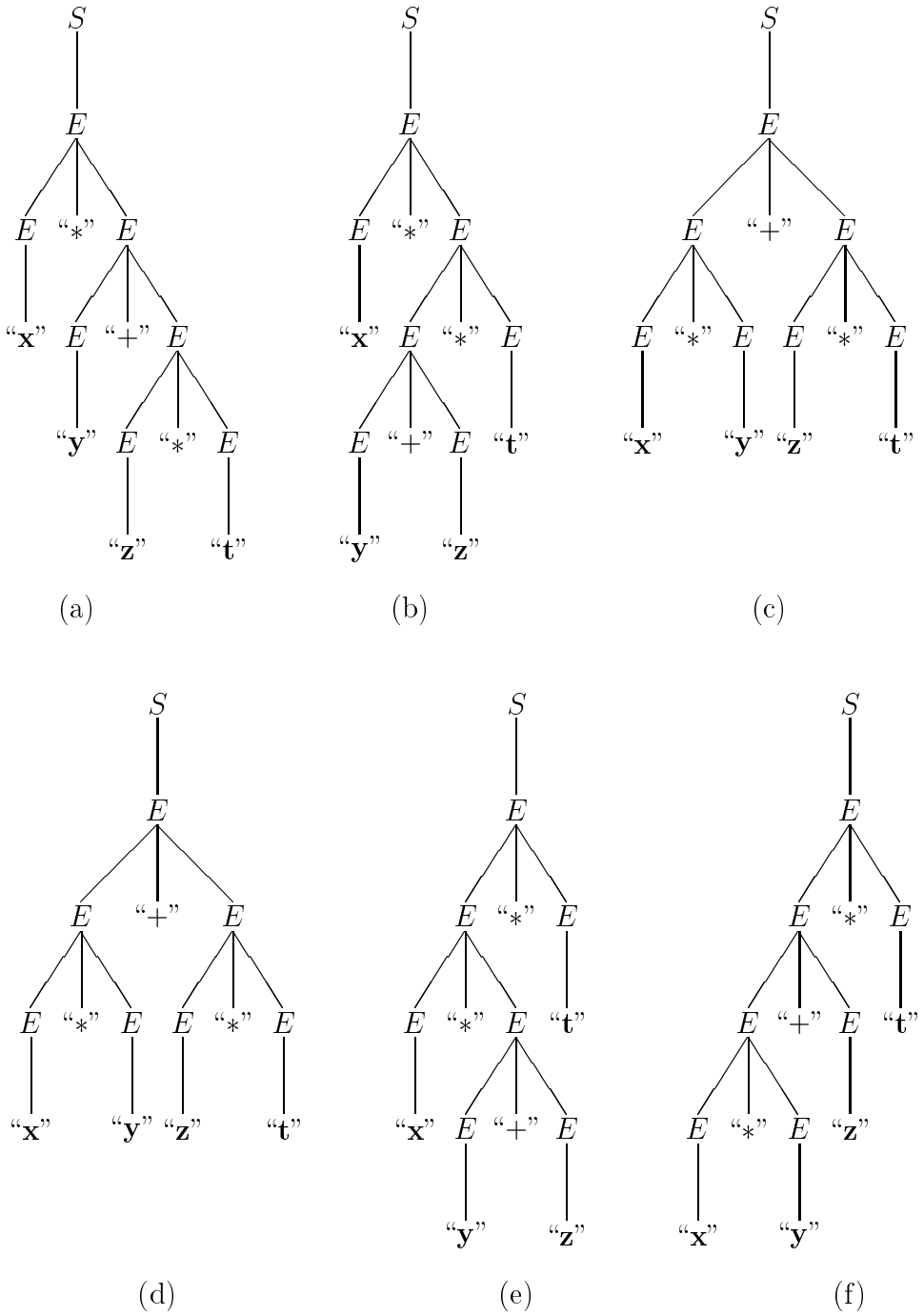

Figure 1.2: Parse trees for $\mathbf{x}*\mathbf{y}+\mathbf{z}*\mathbf{t}$

cal. Neither the derivations nor the parse trees capture the exact procedure followed. □

The widespread popularity of parse trees is not just a matter of notation, although the readability of tree diagrams should not be lightly dismissed. Formal derivations do not readily support the conceptual framework described in the previous section (1.6), while parse trees not only support it, but lead to a fairly natural extension of it (on which more will be said in §2.2). It is not accidental that the standard definition of ambiguity for Chomsky grammars uses parse trees rather than derivations.

Definition 1.19 A CFG is *ambiguous* iff there exists some terminal string for which the grammar admits more than one parse tree. □

There exist context-free languages that are inherently ambiguous, in that they cannot be generated by any unambiguous CFG. However, $L(G_{1.4})$ is not such a language.

Example 1.5 Let $G_{1.5}$ be the following CFG.

$$
\begin{array}{lcl}
S & \rightarrow & T \\
T & \rightarrow & T \text{ “+” } F \\
T & \rightarrow & F \\
F & \rightarrow & F \text{ “}*\text{” } I \\
F & \rightarrow & I \\
I & \rightarrow & \text{“}\mathbf{x}\text{” } | \text{ “}\mathbf{y}\text{” } | \text{ “}\mathbf{z}\text{” } | \text{ “}\mathbf{t}\text{”}
\end{array}
$$

This grammar generates exactly the same language as that of the previous example, $L(G_{1.5}) = L(G_{1.4})$. However, $G_{1.5}$ admits exactly one parse tree for each terminal string in the language. The parse tree for $\mathbf{x}*\mathbf{y}+\mathbf{z}*\mathbf{t}$ is shown in Figure 1.3. Note that there are still a large number of derivations corresponding to this one tree. □

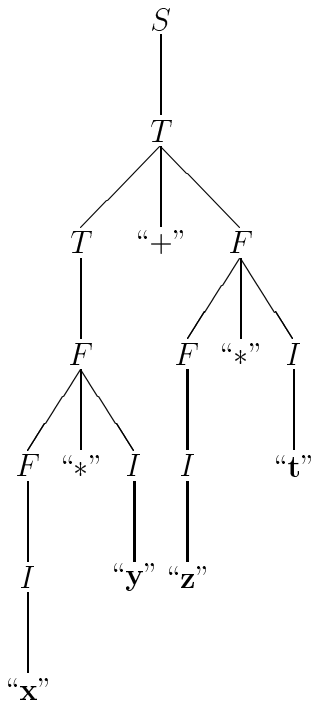

Figure 1.3: Parse tree for **x**∗**y**+**z**∗**t**

Chapter 2

Nonadaptable grammars

2.1 Introduction

Programming languages are not usually context-free, but language designers construct CFGs for their languages anyway. The resulting CFG of a programming language defines a superset language, from which some grammatically valid sentences are rejected during subsequent analysis, based on context-dependent criteria such as lexical scoping or static typing.

In order to reconcile the desire to use CFGs with the need to rigorously define context-dependent language features, various extensions of the CFG model have been developed. Most of these extensions retain a CFG kernel, and augment it with a distinct facility that handles context-dependence.

Of particular concern for this thesis is the family of CFG augmentations that feature *adaptability*; Chapter 3 will survey models in that family. The current chapter surveys *non*adaptable CFG augmentations.

2.2 Attribute grammars

Attribute grammars were introduced by Donald Knuth in the late 1960s; the original papers are [Knut 68, Knut 71]. Knuth traces the historical and intellectual development of the idea in [Knut 90]. A good standard reference on the formalism itself is [Dera 88].

Attribute grammars are the most commonly used augmentation of CFGs for programming language design. The concepts underlying them are also of particular interest for this thesis. Accordingly, they will be covered in greater depth than any other model in this chapter.

2.2.1 Introduction

As early as 1960, E. T. Irons [Iron 61] suggested that the meaning of the symbol on

the left side of a CFG rule could be synthesized from the meanings of the symbols on the right side. The idea is best understood in terms of parse trees (which do not occur in [Iron 61]): The meaning of each parent node in the parse tree is synthesized from the meanings of its children.

Example 2.1 Let $G_{2.1}$ be the following unambiguous CFG.

$$\begin{array}{lcl} S & \rightarrow & B \\ S & \rightarrow & SB \\ B & \rightarrow & \text{“}\mathbf{0}\text{”} \mid \text{“}\mathbf{1}\text{”} \end{array}$$

The language $L(G_{2.1})$ is simply the set of all nonempty strings of zeros and ones. If these strings are interpreted as nonnegative binary integers, one could assign semantic values to the parse tree nodes as follows.

For each grammar rule $r \in R_{G_{2.1}}$, define a function $f_r : \mathbb{N}^{ar_N(r)} \rightarrow \mathbb{N}$, where $ar_N(r)$ is the number of nonterminals on the right side of r, according to the following table.

grammar rule			function		
S	$\rightarrow$	B	$f(x)$	$=$	x
S	$\rightarrow$	SB	$f(x,y)$	$=$	$2x+y$
B	$\rightarrow$	“**0**”	$f()$	$=$	0
B	$\rightarrow$	“**1**”	$f()$	$=$	1

To each nonterminal node n of each parse tree over $G_{2.1}$, associate a nonnegative integer $d(n)$ such that

$$d(n) \;=\; f_{rule(n)}(d(n_1), \cdots d(n_{ar_N(r)}))$$

where $n_1, \cdots n_{ar_N(r)}$ are the nonterminal children of node n from left to right.

This arrangement is illustrated by an *attributed parse tree* in Figure 2.1. The notation $z(d = k)$ at a node n signifies that $label(n) = z$ and $d(n) = k$. d is called a *synthesized attribute*, because it depends on the children of n. (That is, it is “synthesized” from them.) □

Recall, from the previous chapter, the parse tree of Figure 1.3. How should one assign meanings to the nodes of this tree? By analogy with Example 2.1, one might impose attribute functions something like this:

grammar rule			function		
S	$\rightarrow$	T	$f(x)$	$=$	x
T	$\rightarrow$	T “+” F	$f(x,y)$	$=$	$x+y$
T	$\rightarrow$	F	$f(x)$	$=$	x
F	$\rightarrow$	F “$*$” I	$f(x,y)$	$=$	$x*y$
F	$\rightarrow$	I	$f(x)$	$=$	x

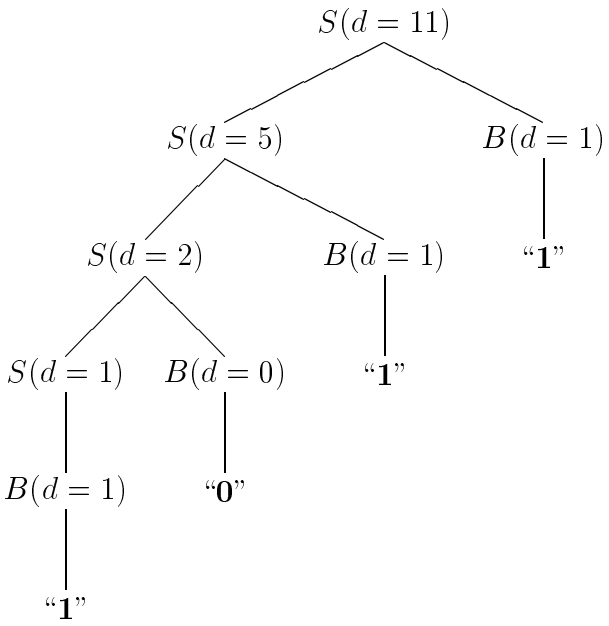

Figure 2.1: Parse tree for **1011**

But this only accounts for the nonterminal rules. How does one assign a value to I in the rule $I \rightarrow$ "**x**"? If the example is taken to be part of a typical programming language, **x** might be the name of, say, a local variable. The "meaning" of identifier **x** is then itself context-dependent. It might be undefined and therefore illegal in the local environment. Or worse, **x** and **y** might be of incompatible types, say a pointer and a real number, that cannot be multiplied together.

Knuth's solution to this problem of context-dependent meaning was to introduce a second kind of attribute, called an *inherited* attribute, that propagates downward in the parse tree from parent to child, just as synthesized attributes propagate upward from children to parent. (The terminology here is due to Knuth.) In general, there may be any number of inherited and/or synthesized attributes associated with each grammar symbol.

2.2.2 Definitions

The definitions and notation of attribute grammars vary from author to author, just as do those of Chomsky grammars. The material below is adapted chiefly from [Knut 68] and [Dera 88].

Definition 2.1 Let G be a context-free grammar. For each $r \in R_G$, let $r = (z_{r,0} \rightarrow z_{r,1} \cdots z_{r,ar(r)})$, where $z_{r,k} \in Z_G$ for all $0 \leq k \leq ar(r)$. An *attribute system* on G has the following parts.

• An alphabet $Attr$. Elements of $Attr$ are called *attributes*.

• Two functions, Syn and Inh, each mapping Z_G into $\mathcal{P}_\omega(Attr)$, subject to the conditions:

$$\begin{array}{rrcl} & Inh(s_G) & = & \emptyset \\ \forall t \in T_G, & Syn(t) & = & \emptyset \\ \forall z \in Z_G, & Syn(z) \cap Inh(z) & = & \emptyset \end{array}$$

For any $z \in Z_G$, elements of $Syn(z)$ or $Inh(z)$ are called *synthesized* or *inherited* attributes of z, respectively. The set of all attributes of z is denoted $Attr(z) = Syn(z) \cup Inh(z)$.

Given a rule $r \in R_G$, suppose $a \in Attr(z_{r,k})$. (That is, suppose a is an attribute of the symbol in position k of rule r.) Then $a(k)$ denotes the combination of attribute a with position k. $a(k)$ is called an *attribute occurrence* in rule r.

• For each attribute $a \in Attr$, a set V_a of *attribute values*.

• For each rule r, and attribute occurrence $a(k)$ in r, where either $k = 0$ and $a \in Syn(z_{r,k})$, or $k > 0$ and $a \in Inh(z_{r,k})$, an equation of the form

$$a(k) \quad = \quad f_{r,k,a}(a_1(k_1), a_2(k_2), \cdots a_i(k_i))$$

where $a_1(k_1) \cdots a_i(k_i)$ are attribute occurrences of r, and $f_{r,k,a}$ is a function

$$f_{r,k,a} : V_{a_1} \times \cdots \times V_{a_i} \to V_a$$

Such equations are called *semantic rules* for r.

An *attribute grammar* (AG[1]) is an ordered pair $\langle G, A \rangle$, where G is a context-free grammar and A is an attribute system on G. □

There is no special concept of derivation relation for attribute grammars as formulated by Knuth. Instead, he used the semantic rules to assign sets of attribute values to the nodes of context-free parse trees. This suggests that attribute evaluation should be thought of as a separate phase of language processing, to be performed *after* parse tree construction. Knuth considered language syntax to be just that which can be described by a CFG, and everything else to be semantics, whereas most of his colleagues at the time defined syntax more broadly [Knut 90].[2]

Note, however, that there are equivalent grammar formalisms with comprehensive derivation relations, encompassing both context-free parsing and attribute evaluation in a single transformation; see §2.2.4. Knuth's formulation has remained dominant, perhaps because his emphasis on parse trees appeals to the intuition.

Definition 2.2 Given an attribute grammar $\langle G, A \rangle$, an *attributed parse tree* over $\langle G, A \rangle$ is a parse tree over G together with, for each node n, a function f_n mapping

[1] The acronym AG for attribute grammars is ubiquitous in the literature. In this thesis, the term "adaptable grammar" is never abbreviated.

[2] Both views of semantics have survived to the present day, each flourishing within its own community of computer scientists who, presumably, find it useful. The conflict between the two views flares up from time to time. See for example [Meek 90].

each $a \in Attr(label(n))$ into an element of V_a, subject to the following condition.

Let n_0 be a nonterminal node, let $r = rule(n_0)$, and let $n_1 \cdots n_{ar(n)}$ be the children of n_0 from left to right. Also, for convenience of notation, let $f_k = f_{n_k}$ for all $0 \leq k \leq ar(n_0)$. If

$$a(k) \quad = \quad f_{r,k,a}(a_1(k_1), \cdots a_i(k_i))$$

is a semantic rule for r, then the following relation must hold.

$$f_k(a) \quad = \quad f_{r,k,a}(f_{k_1}(a_1), \cdots f_{k_i}(a_i))$$

$\diamond$

An attributed parse tree may be called simply a "parse tree" when it is explicitly stated to be over an attribute grammar. $\square$

Example 2.2 Using the AG notation just defined, the grammar and attribute system of Example 2.1 would be written:

$$\begin{array}{lcll} S & \rightarrow & B & d(0) = d(1) \\ S & \rightarrow & SB & d(0) = 2d(1) + d(2) \\ B & \rightarrow & \text{“}\mathbf{0}\text{”} & d(0) = 0 \\ B & \rightarrow & \text{“}\mathbf{1}\text{”} & d(0) = 1 \end{array}$$

Here, the attribute sets have been specified implicitly (in the same fashion as the terminal and nonterminal alphabets).

In illustrating an attributed parse tree, the notation $z(a_1 = v_1, \cdots a_i = v_i)$ at a node n signifies that $Attr(z) = \{a_1 \cdots a_i\}$ and, for all $1 \leq k \leq i,\ f_n(a_k) = v_i$. $\square$

The example of evaluating binary numbers is used in both [Knut 68] and [Dera 88] (with the minor complication that rational numbers are supported, rather than just integers). In order to illustrate inherited as well as synthesized attributes, a more elaborate evaluation strategy is used.

Example 2.3 In contemplating the parse tree of Figure 2.1, it could be argued that the value of each digit depends on where it occurs in the sentence. Thus, the value of the leading "**1**" is 8 rather than 1. This idea can be formalized in an AG by introducing an inherited attribute r (for *rank*), which is the power of two by which the value of each digit should be multiplied. The leading "**1**" has a rank of 3. (To make this example work out naturally, it is necessary to remove the start symbol from the right sides of grammar rules by introducing a new nonterminal T.)

$$\begin{array}{lcll} S & \rightarrow & T & r(1) = 0, \quad d(0) = d(1) \\ T & \rightarrow & B & r(1) = r(0), \quad d(0) = d(1) \\ T & \rightarrow & TB & r(2) = r(0), \quad r(1) = r(0) + 1, \quad d(0) = d(1) + d(2) \\ B & \rightarrow & \text{“}\mathbf{0}\text{”} & d(0) = 0 \\ B & \rightarrow & \text{“}\mathbf{1}\text{”} & d(0) = 2^{r(0)} \end{array}$$

The parse tree for **1011** under this AG is shown in Figure 2.2. $\square$

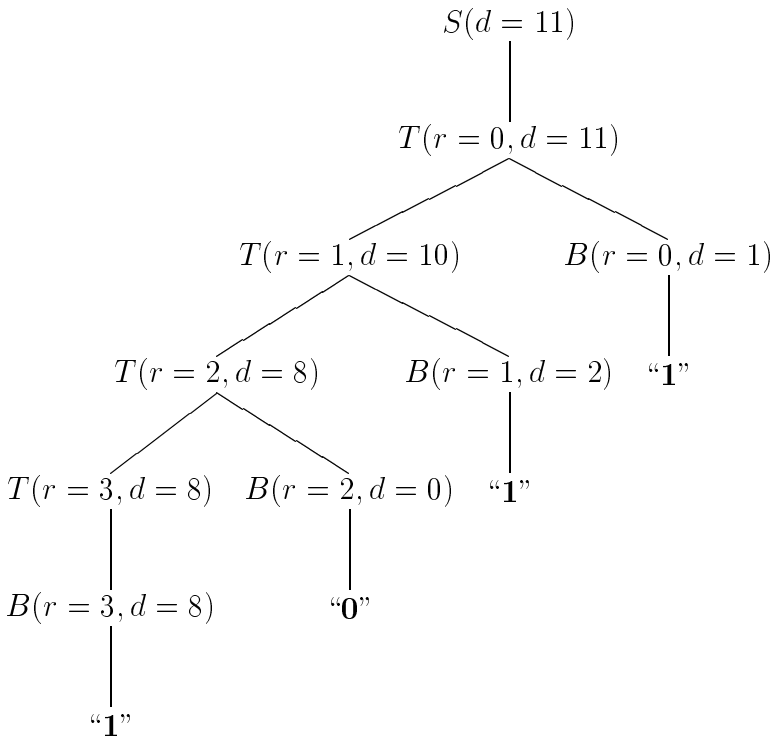

Figure 2.2: Parse tree for **1011**

Knuth claims that the second, more elaborate evaluation strategy "does seem to correspond better to our intuition than [the simpler strategy]" [Knut 68, p. 131]. (This may be so, but the first evaluation strategy led to a simpler attribute grammar.) At any rate, inherited attributes were also intended to address more substantial problems, such as the context-dependent semantics of identifiers in Figure 1.3.

Indeed AGs can and do address the more substantial problems. The technique usually employed is to pass the entire symbol table for the local environment about the tree in an environment attribute. Each node of the tree inherits a local symbol table. Identifier references synthesize their meanings by looking themselves up; declarations synthesize a new environment by adding to the inherited table. However, examples of such techniques are inappropriately large for this survey.

2.2.3 Properties

Definition 2.3 An attribute grammar $\langle G, A \rangle$ is in *normal form* iff, for every grammar rule $r = (z_{r,0} \to z_{r,1} \cdots z_{r,ar(r)}) \in R_G$ and every semantic rule

$$a(k) \quad = \quad f_{r,k,a}(a_1(k_1), \cdots a_i(k_i))$$

for r, the attribute occurrences on the right side of the semantic rule do not occur on the left side of any semantic rule for r. In other words,

$$\{a_1(k_1) \cdots a_i(k_i)\} \subseteq (\{b(0) \mid b \in Inh(z_{r,0})\} \cup \{b(j) \mid j > 0 \text{ and } b \in Syn(z_{r,j})\})$$

□

Normal form requires that the semantic rules for a grammar rule r must not directly define attribute occurrences of r in terms of each other. A second well-formedness property, non-circularity, imposes a weaker condition (no value depends on itself), but on a wider scale (across all the nodes of a parse tree). Neither of these properties necessarily implies the other.

Definition 2.4 Suppose $\langle G, A\rangle$ is an attribute grammar, and r is a grammar rule over G. Given any two attribute occurrences $a(k)$ and $b(j)$ of r, $a(k)$ is *locally dependent on* $b(j)$, denoted $a(k) \lhd_r b(j)$, iff there is a semantic rule for r with $a(k)$ on its left side and $b(j)$ on its right side. The relation $\lhd_r$ over the domain of attribute occurrences of r is called the *local dependency relation* of r.

Suppose $\langle G, A\rangle$ is an attribute grammar, and t is a parse tree over $\langle G, A\rangle$. For each nonterminal node n_0 of t, define a relation $\lhd_{n_0}$ on the domain of node-attribute pairs, as follows. Let $n_1, \cdots n_{ar(n_0)}$ be the children of n_0 from left to right. Then $\langle n_i, a\rangle \lhd_{n_0} \langle n_j, b\rangle$ iff $a(i) \lhd_{rule(n_0)} b(j)$. The *compound dependency relation* for t, denoted $\lhd_t$, is the transitive closure of the union of all relations $\lhd_n$ for nonterminal nodes n in t.

An attribute grammar $\langle G, A\rangle$ is *non-circular* iff, for every parse tree t over $\langle G, A\rangle$, the relation $\lhd_t$ is acyclic. (That is, iff there is no node-attribute pair $\langle n, a\rangle$ for which $\langle n, a\rangle \lhd_t \langle n, a\rangle$.) □

Definition 2.5 An attribute grammar $\langle G, A\rangle$ is *semantically ambiguous* iff there exist a terminal string $w \in L(G)$, an attribute $a \in Syn(s_G)$, and parse trees t and t' for w, such that $f_{root(t)}(a) \neq f_{root(t')}(a)$.

Attribute grammar $\langle G, A\rangle$ is *syntactically ambiguous* iff G is ambiguous. □

Two attributed parse trees for the same string may have different context-free structures, yet assign the same attribute values to the root node; thus syntactic ambiguity does not necessarily imply semantic ambiguity. On the other hand, there may be more than one way to "decorate" a single context-free parse tree (that is, to assign attribute values to the nodes); thus semantic ambiguity does not necessarily imply syntactic ambiguity. However, in a *non-circular* attribute grammar there can only be one way to decorate each context-free tree. Therefore,

Theorem 2.1 Every syntactically unambiguous non-circular attribute grammar is semantically unambiguous. □

As presented by Definition 2.1, an attribute grammar $\langle G, A\rangle$ does not actually reject any terminal string w in the context-free language $L(G)$. The attribute system A only provides a criterion for "decorating" parse trees (that is, assigning attribute values to the nodes). Knuth does not go into detail, but suggests something such as the following.

Definition 2.6 A parse tree t over an attribute grammar $\langle G, A\rangle$ is *invalid* iff $valid \in Syn(s_G)$, $V_{valid} = \{true, false\}$, and $f_{root(t)}(valid) = false$.

The *language defined* by an attribute grammar $\langle G, A\rangle$ is

$$L(\langle G, A\rangle) \;=\; \{w \in L(G) \mid \text{there is a valid parse tree for } w \text{ over } \langle G, A\rangle\}$$

□

Example 2.4 It is now fair game to ask what class of languages are recognized by attribute grammars. The answer depends on the nature of the semantic functions, about which [Knut 68] says nothing. To demonstrate the importance of this point, suppose L is an arbitrary language over an arbitrary alphabet Z. Let $\langle G, A\rangle$ be the following attribute grammar.

$$\begin{array}{lcll} & S & \to & T & valid(0) = (s(1) \in L) \\ & T & \to & TC & s(0) = s(1) \cdot s(2) \\ & T & \to & \lambda & s(0) = \lambda \\ \forall z \in Z, & C & \to & z & s(0) = z \end{array}$$

This attribute grammar is in normal form, non-circular, and syntactically and semantically unambiguous. The language generated by G is $L(G) = Z^*$. And the language defined by $\langle G, A\rangle$ is $L(\langle G, A\rangle) = L$. By strict interpretation of Definition 2.1, every single language L in the uncountably infinite domain $\mathcal{P}(Z^*)$ is defined by some attribute grammar. □

In practice, obviously, uncomputable functions are *not* used in the semantic rules of attribute grammars. In fact, semantic functions may be required to be primitive recursive.[3] Under restrictions of this kind, determining the computational power of the model becomes a complicated issue; see [Dera 88, chap. 7].

2.2.4 Variants

There are several other augmented-CFG formalisms that, although important enough to warrant individual acknowledgment here, are also very similar to Knuth's attribute grammars in substance.

[3] For a formal treatment of the class of primitive recursive functions, see [Roge 67].

Extended attribute grammars

Extended attribute grammars (EAGs) are a modified version of attribute grammars introduced by Watt and Madsen in the late 1970s. The original paper is [Watt 77]. The principal reference used here is [Mads 80]. EAGs figured prominently in the earlier phases of Christiansen's work on adaptable grammars, which will be discussed in §3.3. Both ideas and notations from EAGs also significantly influenced the model proposed in Part II of the current work.

EAGs differ from Knuth's original model in three ways: (1) The semantic function domains are treated more rigorously. (2) Attribute evaluation is integrated into the derivation step relation. (3) Knuth's semantic equations are replaced by a more declarative form.

Semantic domains are specified by means of *many-sorted algebras*. A many-sorted algebra consists of one or more data domains, together with one or more functions over specified combinations of the domains. There is a substantial mathematical technology for the construction of many-sorted algebras. The special case of one-sorted algebras is addressed in §5.2; for the many-sorted case, and more in-depth coverage, see [Wech 92].

Rather than specifying two sets of rules, one context-free and the other semantic, an EAG has a single set of *rule forms*. Each nonterminal occurrence is accompanied by its attributes, listed in some predetermined order; terminals don't have attributes. The notation used is:

$$\langle n \updownarrow x_1 \cdots \updownarrow x_i \rangle$$

Here, n is the nonterminal, $x_1 \cdots x_i$ are polynomials over the appropriate attribute value domains, and $\updownarrow$ should be replaced by either $\uparrow$ or $\downarrow$, signifying that the following attribute is synthesized or inherited, respectively. This construction is called an *attributed nonterminal form*.

A *rule instance* is constructed from a rule form by assigning values to the variables and evaluating the polynomials. (Similarly an *attributed nonterminal instance* from an attributed nonterminal form.) The derivation step relation is defined as for Chomsky grammars, but over the combined alphabet of terminal symbols and attributed nonterminal instances, with rewriting based on the set of rule instances.

Example 2.5 The AG of Example 2.3 corresponds to the following EAG. (To be precise, only the rule forms are listed here; a complete EAG would include the machinery of the many-sorted semantic algebra.) Variables are named v_k, and have domain $\mathbb{N}$.

$$\begin{aligned}
\langle S \uparrow v_1 \rangle &\rightarrow \langle T \downarrow 0 \uparrow v_1 \rangle \\
\langle T \downarrow v_0 \uparrow v_1 \rangle &\rightarrow \langle B \downarrow v_0 \uparrow v_1 \rangle \\
\langle T \downarrow v_0 \uparrow v_1 + v_2 \rangle &\rightarrow \langle T \downarrow v_0 + 1 \uparrow v_1 \rangle \langle B \downarrow v_0 \uparrow v_2 \rangle \\
\langle B \downarrow v_0 \uparrow 0 \rangle &\rightarrow \text{“}\mathbf{0}\text{”} \\
\langle B \downarrow v_0 \uparrow 2^{v_0} \rangle &\rightarrow \text{“}\mathbf{1}\text{”}
\end{aligned}$$

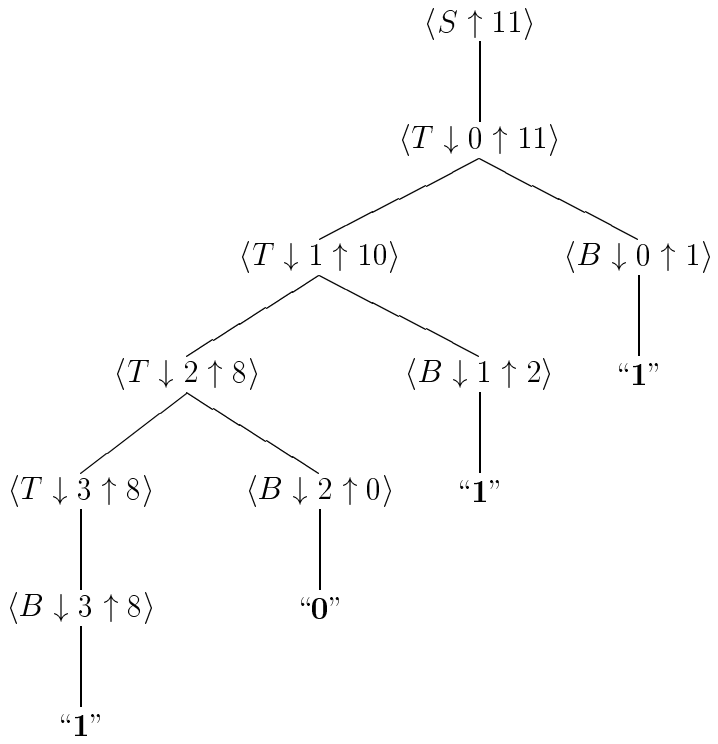

Figure 2.3: Parse tree for **1011**

The parse tree for **1011** under this EAG is shown in Figure 2.3. □

The distinction between inherited and synthesized attributes no longer has any formal significance. This makes for a much more declarative formalism. An EAG rule form constrains its attribute occurrences without necessarily defining them, and moreover may constrain any or all of its attribute occurrences rather than just synthesized attributes on the left and inherited on the right.

Example 2.6 Because attribute evaluation is now part of the derivation relation, attribute constraints can be used to generate non-context-free languages without resorting to a specially designated validity attribute. To illustrate, consider the following EAG. (Assume that all attributes have domain $\mathbb{N}$.)

$$\begin{array}{rcl}
\langle S \rangle & \rightarrow & \langle A \uparrow v_0 \rangle \langle B \uparrow v_0 \rangle \langle C \uparrow v_0 \rangle \\
\langle A \uparrow v_0 + 1 \rangle & \rightarrow & \langle A \uparrow v_0 \rangle a \\
\langle B \uparrow v_0 + 1 \rangle & \rightarrow & \langle B \uparrow v_0 \rangle b \\
\langle C \uparrow v_0 + 1 \rangle & \rightarrow & \langle C \uparrow v_0 \rangle c \\
\langle A \uparrow 0 \rangle & \rightarrow & \lambda \\
\langle B \uparrow 0 \rangle & \rightarrow & \lambda \\
\langle C \uparrow 0 \rangle & \rightarrow & \lambda
\end{array}$$

Here is a typical derivation under the grammar.

$$\begin{aligned}\langle S\rangle &\Rightarrow \langle A\uparrow 2\rangle\langle B\uparrow 2\rangle\langle C\uparrow 2\rangle \Rightarrow \langle A\uparrow 1\rangle a\langle B\uparrow 2\rangle\langle C\uparrow 2\rangle \\ &\Rightarrow \langle A\uparrow 0\rangle aa\langle B\uparrow 2\rangle\langle C\uparrow 2\rangle \Rightarrow aa\langle B\uparrow 2\rangle\langle C\uparrow 2\rangle \stackrel{+}{\Rightarrow} aabb\langle C\uparrow 2\rangle \\ &\stackrel{+}{\Rightarrow} aabbcc\end{aligned}$$

The first step of every derivation is an instantiation of the first rule form. Because that rule form uses the same variable in all three attribute positions, the same synthesized attribute value is assigned to all three branches of the parse tree. Each branch generates a substring whose length equals the value of the synthesized attribute.

Consequently, the language generated by this EAG is $\{a^n b^n c^n \mid n \in \mathbb{N}\}$. Note, incidentally, that the first step of each derivation uniquely determines which terminal string will be generated. □

Definite clause grammars

A grammar formalism based on Prolog was developed by A. Colmerauer in the mid-1970s; the original paper is [Colm 75]. In essence, Colmerauer's "metamorphosis grammars" are an attributed form of Chomsky type 0 grammars. The subclass of these creatures corresponding to attributed Chomsky type 2 grammars, and thus to Knuth's attribute grammars, have been given the name *Definite Clause Grammars* (DCGs) [Pere 80].

Most of the differences between EAGs and DCGs are superficial; DCG notation is heavily influenced by its Prolog origin. Symbols on the right side of a rule are separated by commas, and the rule ends with a period. Each terminal is delimited by square brackets (Prolog notation for a list); an attributed nonterminal is represented by a Prolog-style term, $n(x_1, \cdots x_i)$. Any identifier beginning with an upper-case letter is a variable. A typical DCG rule might be:

$$bitlist(Rank, Value) \quad \rightarrow \quad bit(Rank, Value).$$

However, there is also one major difference from EAGs: DCG notation allows arbitrary Prolog code to be embedded into rules, delimited by braces, {}. This allows a DCG to directly express arbitrary logical constraints on the attributes. On the other hand, it also has the potential to completely circumvent the grammatical structure.

Affix grammars

Affix grammars were "invented" (as [Kost 91] puts it) in 1962 for use in linguistics, but not formalized until 1970. The original formal paper is [Kost 71]; more recent sources are [Kost 91] and [Meij 90].

Affix grammars are still another formalism that embeds the attributes into the grammar rules; they are "affixed" to the nonterminals, using the notation $n + a_1 +$

$\cdots + a_i$. Otherwise, the notation strongly resembles that of W-grammars (which will be discussed in §2.3). A modified version of the formalism, called *extended* affix grammars, was introduced by D. A. Watt in [Watt 74] —three years before he coauthored the original paper on extended attribute grammars. Extended affix grammars are virtually identical to extended attribute grammars, but for superficial differences of notation.

Higher order attribute grammars

Higher order attribute grammars are an extension of AGs in which new branches of the parse tree can be constructed during attribute evaluation, and populated with attributes for subsequent evaluation. They were introduced in 1989; the original paper is [Vogt 89].

A higher-order attribute grammar takes the same form as an ordinary AG, i.e., $\langle G, A \rangle$ with G a CFG and A and attribute system. However, the attribute evaluation process is allowed to invoke itself by means of *nonterminal attributes*.

Formally, a nonterminal attribute α is both a nonterminal and an attribute. As a nonterminal, α is effectively equal to λ. It cannot occur on the left side of a (context-free) rule, and when it occurs on the right side of a rule during parsing it recognizes λ and becomes a leaf of the parse tree.

As an attribute, its semantic domain V_α is the set of context-free parse trees of the grammar. Each occurrence of α in a rule r occurs on the left side of a semantic equation of r. Once an occurrence of α in the parse tree has been evaluated, its value t is attached to the parse tree as a new branch at the α-node. The attributes of this new branch are then subject to evaluation as if they were part of the original parse tree.

The application of the attribute evaluation process to parse trees constructed by the attribute evaluation process may be viewed as a form of recursion. This aspect of higher-order attribute grammars will be discussed (briefly) in §4.4.

2.2.5 Comments

Attribute grammars, and their allies, have a distinct advantage over context-dependent Chomsky grammars as a descriptive medium. All the information produced by an AG processing an input string can be represented in a tree diagram. This is more than just a convenient visual representation. Tree diagrams embody the hierarchical structure that was at the core of the conceptual elegance of CFGs discussed in §1.6.

Moreover, the connection between AGs and their parse trees runs deeper than merely attaching the extra information to tree nodes. The very computations and constraints that determine that information are localized in the tree. (The use of embedded Prolog code in DCGs can violate this principle.) Knuth identified this localization as one of the key advantages of his model [Knut 68].

On the other hand, attribute grammars can easily *obscure* their own descriptions. The more context-dependent a programming language is, the looser its context-free superset will be, and the more heavily it will rely on its semantic rules (or their equivalent in any of the variant formalisms). While the semantic rules become ever more complicated, the context-free rules will actually become simpler, or as [Chri 90] aptly puts it, more inane.

To bring the point home, [Chri 90] cites [Uhl 82], an attribute grammar for Ada, in which a single context-free rule representing general procedure or function call (rule r_084) has two and a half pages' worth of semantic rules. The asymptotic limit of this trend may be seen in Example 2.4 of §2.2.3, where the CFG carries no information at all.

Finally, a few words should be said about synthetic versus inherited attributes. Knuth's distinction between the two, crucial to his original insight, is being undermined or outright dismissed by the newer generation of attributed grammar models.

At the heart of this shift in attitude is a belief that the computational aspect of Knuth's model detracts from its descriptive role. (The synthetic/inherited distinction is, after all, a matter of the direction of computation.) In attribute grammars for nontrivial languages (such as [Uhl 82]), the path of computation is liable to be so complicated that working it out in the first place actually hampers the language designer; [Pere 80] embraces this as an argument in favor of DCGs.

Yet, Example 2.3 (= 2.5) seems actually to be clarified by the distinction. The explanation for this may lie in the nature of the example. A sufficiently simple algorithm can be an aid to understanding, while any algorithm too complicated to grasp as a whole will become a liability.

The issue of declarative versus imperative language description will be revisited several times, ultimately in §4.4.

2.3 W-grammars

In the mid-1960s, Aad van Wijngaarden developed a grammar formalism specifically for the formal definition of programming languages, based on a philosophy of generality with simplicity. His original paper is [Wijn 65]. The new formalism was adopted for a new programming language design project that eventually produced ALGOL 68.

Grammars under this formalism are called *van Wijngaarden grammars*, often shortened to vW-grammars or W-grammars. Some authors use the name *two-level grammars*, but this could lead to confusion since affix-grammarists use it to mean the general class that includes W-grammars, affix grammars, and all of the other variants of AGs as well. The name *W-grammars* is preferred here.

Early presentations of W-grammars were hampered by a great deal of gratuitous terminology, which more recent treatments tend to omit. The material presented here was drawn primarily from [Clea 76], which uses the old terminology; but the terminology used here owes more to [Kost 91].

2.3.1 Explanation

The basic idea of a W-grammar is that, rather than enumerating a finite set of grammar rules over a finite symbol alphabet, one constructs a finite meta-grammar that generates the symbols and rules of the grammar. In this way, one can define a Chomsky-esque grammar with infinitely many nonterminals and rules.

Each grammar symbol is designated by a string of lower-case letters. If the string ends with **symbol**, it designates a terminal symbol; otherwise it designates a nonterminal symbol. (Sic, nonterminals are symbols whose names don't end with **symbol**.) Since symbolic names are being used for the terminals, a complete W-grammar includes a list of typographic representations for terminals.

A rule consists of a nonterminal, followed by a colon, and a list of symbol-sequences separated by semicolons and terminated by a period. Each symbol-sequence is a list of grammar symbols separated by commas. (The commas are needed because embedded whitespace is not significant.)

Example 2.7 Grammar $G_{1.1}$, which generates the language $\{a^n b^n \mid n \in \mathbb{N}\}$, could be represented in van Wijngaarden notation as follows.

Terminal	Representation
letter a symbol	**a**
letter b symbol	**b**

Rules

sentence : ;
letter a symbol,
sentence,
letter b symbol.

By default, the first nonterminal defined will be the start symbol. Hereafter, terminals for the (lower case) letters of the alphabet will be assumed as above. □

A *rule schema* has the same form as a rule, except that both upper- and lower-case letters may be used anywhere that lower-case letters would occur in a rule. The intent is that the schema will be instantiated by replacing sequences of upper-case letters by sequences of lower-case letters until there are no upper-case letters remaining and, thus, a rule has been obtained. A sequence of upper-case letters that is replaced as a unit is called a *meta-variable*. For example, **ALPHA** is presumably a meta-variable in the following rule schema.

token : letter ALPHA symbol.

Meta-variables are defined by means of *meta-rules*. A meta-rule consists of a meta-variable, followed by two colons, a list of strings (of upper- and lower-case letters) separated by semicolons, and a period. The set of meta-rules of a W-grammar form

a CFG whose "nonterminals" are upper-case strings (i.e., meta-variables) and whose "terminals" are lower-case strings.

Each meta-variable has an associated set of lower-case strings to which it can be expanded. Each rule schema has an associated set of rules, obtained by *consistently* substituting lower-case strings for the meta-variables. That is, when a meta-variable occurs more than once in a rule-schema, all of its occurrences must be replaced with the same lower-case string. The target language is defined by the union of the associated rule sets of all the rule schemata.

Some clarification is in order regarding the use of upper- and lower-case strings. First of all, a sequence of lower-case letters designates a symbol *only* when it is delimited by punctuation. For example, in the grammar rule of Example 2.7 (above), the string **letter** occurred twice, but in both cases, it was being used as merely part of a symbol name, not as a symbol name in its own right. Similarly, in the rule schema shown just after that example, **letter** again occurs but is not used as a complete symbol name.

One might suppose that a sequence of upper-case letters designates a meta-variable iff it is surrounded by punctuation and/or lower-case letters, as **ALPHA** in the above rule schema. Unfortunately, this may not be so. There is no way of stipulating within the schema that **ALPHA** must be replaced as a unit; for example, there may be meta-rules

AL :: xy.
PHA :: z.

in which case **ALPHA** is the concatenation of the two meta-variables **AL** and **PHA**. It is the responsibility of the person writing the W-grammar to make sure that no upper-case sequence is ambiguous, in the sense of having more than one possible decomposition into meta-variables.

Example 2.8 The following W-grammar generates the language $\{\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n \mid n \in \mathbb{N}_+\}$.

Meta-rules
N :: n; Nn.
L :: a; b; c.

Rule schemata
s : Na, Nb, Nc.
nL : letter L symbol.
nNL : letter L symbol, NL.

The first meta-rule introduces an infinite number of expansions **n**, **nn**, **nnn**, and so on for the meta-variable **N**. The second meta-rule provides a shorthand that will reduce the number of rule schemata needed to define the grammar.

The first rule schema defines an infinite collection of rules, one for each possible expansion of the meta-variable **N**.

s : na, nb, nc.
s : nna, nnb, nnc.
s : nnna, nnnb, nnnc.

and so on. Note that each of these rules has just three grammar symbols on its right side, all of which are nonterminals. The expansions of **N** did not become grammar symbols in their own right, but are simply embedded in the nonterminals, each of which ends with the letter **a**, **b**, or **c**.

The second rule schema defines just three rules:

na : letter a symbol.
nb : letter b symbol.
nc : letter c symbol.

The third rule schema defines rules that incrementally reduce the nonterminals spawned by the first schema, such as

nna : letter a symbol, na.
nnna : letter a symbol, nna.

□

It is actually rather easy (but tedious, and therefore omitted here) to construct a W-grammar that simulates the action of an arbitrary Chomsky type 0 grammar. Each arbitrary string of Chomsky symbols is represented in the W-grammar by a single nonterminal. Each Chomsky rule is represented by a single rule schema whose instances simulate all of the possible Chomsky derivation steps using that Chomsky rule. Turing machines are just as easy to simulate, with a nonterminal for each configuration and a rule schema for each set of arguments to the transition function. Formally,

Theorem 2.2 Suppose L is a language. Then there exists a W-Grammar W that generates L iff L is recursively enumerable. □

2.3.2 Comments

It has been shown [Demb 78] that to every W-grammar there is an attribute grammar that generates a structurally identical parse tree for every generated string of terminals, and vice versa. [Kost 91] explores this relationship from an affix-grammarist's point of view. He compares three formal definitions of a toy language, using W-grammars, affix grammars, and attribute grammars. His three definitions are almost identical, with meta-variables, affixes, and attributes serving essentially the same formal role. Of course, he had the advantage of choosing both the definition strategy

under each model, and the nature of the toy language itself; nevertheless, it is an impressive demonstration.

There is also a close historical connection between W-grammars and DCGs. Readers versed in Prolog may observe that the notations of W-grammars and Prolog programs have a great deal in common, even to the use of upper-case letters to denote variables. This is not a coincidence. W-grammars were an important intellectual influence on the early development of Prolog; see [Cohe 88].

In terms of technical features, W-grammars are the simplest Turing-powerful model surveyed in Part I of this thesis, save the Chomsky model itself. The derivation process is based entirely on string rewriting by substitution. There are no additional devices, such as the semantic functions of attribute grammars (with their associated semantic domains), or the various imperative adaptation mechanisms of §3.2 in the next chapter.

Admirable as simplicity is in general, it can sometimes become a problem if there is no means of abstraction —in this case, no means of escaping the primitive character of the concrete rewriting mechanism. Spartanism of this kind usually leads to verbosity, as when one attempts to construct a nontrivial program for a Turing machine. This is by no means to advocate complexity; it is merely pointed out that not all simplicity is equally useful.

On the other hand, the W-grammar derivation process is not so simple in concept. It has three distinct aspects that must be kept track of simultaneously: the multi-step rewriting of meta-variables as strings of lower-case letters, by means of meta-rules; the instantiation of rule schemata, by consistent substitution based on completed meta-variable rewrites; and the rewriting of strings of symbols, by application of completely instantiated rules. Note that, besides the immediate conceptual difficulty of this heterogeneous arrangement, the logically earlier phases of the process tend to be obscured when the derivation is represented by a parse tree.

2.4 Control-restricted CFGs

The models described in the preceding two sections account for the lion's share of grammars used for language definition in practice. There is also a vast realm of nonadaptable grammar models that are restricted primarily to the theoretical arena, either because they have failed to catch on as practical tools, or because they have never been considered for that role. As a sampling of this realm, we consider a few models in the family of *control-restricted* context-free grammars; this particular family was chosen because it offers an interesting break from the philosophy of the other models in this chapter. Another interesting case, indexed grammars, will be addressed in the next section.

The material here is drawn principally from [Maye 72], with some simplifications for the sake of brevity. Another work in a somewhat similar vein is [Roze 80]. The name *control-restricted* is a characterization introduced here for convenience.

2.4.1 Explanation

Most of the grammars described in this chapter increase their computational power by appending context-dependent information to the nonterminals as they occur during derivation. However, it is also possible to increase the computational power of a CFG by restricting the order in which the grammar rules are applied.

Matrix grammars

A *matrix grammar* is an ordered pair $\langle G, M \rangle$, where G is a context-free grammar, and $M \in \mathcal{P}_\omega(R_G^+)$ is a finite set of nonempty sequences of rules from R_G. The elements of M are called *matrices*, hence the name of the grammar model.

A derivation over G is considered legal in the matrix grammar $\langle G, M \rangle$ iff the sequence s of rules applied in the derivation can be formed by concatenating elements of M, that is, iff $s \in M^*$. The notation used for matrices is $[r_1, \cdots r_i]$.

Example 2.9 Let $G_{2.9}$ be the following context-free grammar.

$$\begin{array}{lcl} S & \rightarrow & ABC \\ A & \rightarrow & Aa \\ A & \rightarrow & a \\ B & \rightarrow & Bb \\ B & \rightarrow & b \\ C & \rightarrow & Cc \\ C & \rightarrow & c \end{array}$$

Evidently $L(G_{2.9}) = \{a^i b^j c^k \mid i, j, k \in \mathbb{N}_+\}$. Now let M be the following set of matrices over $G_{2.9}$.

$$\begin{array}{lcll} M & = & \{ & [S \rightarrow ABC], \\ & & & [A \rightarrow a,\ B \rightarrow b,\ B \rightarrow c], \\ & & & [A \rightarrow Aa,\ B \rightarrow Bb,\ C \rightarrow Cc] \quad \} \end{array}$$

Under the matrix grammar $\langle G_{2.9}, M \rangle$, every expansion of nonterminal A must be followed by corresponding expansions of nonterminal B and nonterminal C, in that order. Consequently, $L(\langle G_{2.9}, M \rangle) = \{a^n b^n c^n \mid n \in \mathbb{N}_+\}$. □

Unordered scattered context grammars

An *unordered scattered context grammar* is an ordered pair $\langle G, U \rangle$, where G is a context-free grammar, and $U \in \mathcal{P}_\omega(\mathcal{P}_\omega(R_G))$ is a finite set of finite sets of rules from R_G. The elements of U are called *production systems*.

A derivation step *via production system* $P \in U$, denoted $\beta \underset{P}{\Rightarrow} \gamma$, consists of simultaneous application of all the rules in P to nonterminals in β. In other words, if

$$P \ = \ \{\ A_1 \rightarrow \alpha_1,\ \cdots A_i \rightarrow \alpha_i\ \}$$

then there exist strings $\beta_0 \cdots \beta_i$ and a permutation $\langle A_{k_1}, \cdots A_{k_i} \rangle$ of the left sides of the rules in P, such that

$$\begin{aligned} \beta &= \beta_0 A_{k_1} \beta_1 \cdots A_{k_i} \beta_i \\ \gamma &= \beta_0 \alpha_{k_1} \beta_1 \cdots \alpha_{k_i} \beta_i \end{aligned}$$

A derivation over unordered scattered context grammar $\langle G, U \rangle$ is a chain of zero or more derivation steps over elements of U.

Example 2.10 Recalling the previous example, let U be the following set of production systems over $G_{2.9}$.

$$\begin{aligned} U = \{ \; & \{S \to ABC\}, \\ & \{A \to a,\ B \to b,\ B \to c\}, \\ & \{A \to Aa,\ B \to Bb,\ C \to Cc\} \; \} \end{aligned}$$

Under the unordered scattered context grammar $\langle G_{2.9}, U \rangle$, every expansion of nonterminal A must be performed in parallel with corresponding expansions of nonterminal B and C. Consequently, $L(\langle G_{2.9}, U \rangle) = \{a^n b^n c^n \mid n \in \mathbb{N}_+\}$. □

Evidently, the unordered scattered context grammars generate a proper superset of the CFLs —and from Example 2.9, so do the matrix grammars. As it happens, the language classes defined by these two models are identical.

Theorem 2.3 A language L is generated by some matrix grammar iff it is generated by some unordered scattered context grammar. □

Normal CSGs

Another way to restrict the control of a CFG is to associate with each rule r a *local context* (as opposed, presumably, to a scattered context) that must immediately surround the nonterminal being rewritten. One might write $context(n \to \nu) = \langle \alpha, \beta \rangle$, meaning that the rule $n \to \nu$ can only be applied when the occurrence of n being rewritten is immediately preceded by α and immediately followed by β.

This of course can be expressed as a Chomsky grammar rule, $\alpha n \beta \to \alpha \nu \beta$. With the constraint that $\nu \neq \lambda$, the rule is normal context-sensitive. In fact, normal CSGs are one of the gaggle of control-restricted CFGs considered by [Maye 72].

2.4.2 Comments

There is a distinct flavor of control-restriction about the consistent rewriting of metavariables in the rule schemata of a W-grammar. It is perhaps not surprising, then, that control-restricted CFGs share a basic conceptual weakness with W-grammars: part of the derivation process is lost when the derivation is represented by a parse tree.

2.5 Indexed grammars

Indexed grammars were introduced in the late 1960s by A. Aho [Aho 68]. The material here is adapted from [Hopc 79].

2.5.1 Explanation

An *indexed grammar* is a five-tuple $G = \langle Z, T, I, R, s \rangle$, where:

- Z, T, and s are as for Chomsky grammars (Definition 1.3, in §1.2).
- I is a finite alphabet disjoint from Z, that is, $I \cap Z = \emptyset$. Its elements are called *indices*.
- R is a finite set of indexed grammar rules, each of which may take any of the following three forms:

$$\begin{array}{rcl} A & \rightarrow & \alpha \\ A & \rightarrow & Bf \\ Af & \rightarrow & \alpha \end{array}$$

  for any nonterminals A and B, string $\alpha \in Z^*$, and index $f \in I$.

An *indexed nonterminal* is an element of the set NI^*; that is, a nonterminal followed by a string of indices. Indexed nonterminals are roughly analogous to the attributed nonterminals of EAGs (§2.2.4); the value of the index string depends on the form of the grammar rule applied, as follows.

- When applying a rule of the form $A \rightarrow \alpha$, all of the nonterminals in α inherit exactly the index list of A.
- When applying a rule of the form $A \rightarrow Bf$, the index list of B is formed by concatenating index f onto the front of the index list of A.
- When applying a rule of the form $Af \rightarrow \alpha$, the index list of A must begin with f, and all of the nonterminals in α receive the index list of A with the initial index f removed.

In effect, the index list is manipulated as a stack; the first form of rule does not change the stack, the second form pushes an index onto the stack, and the third form pops an index off the stack.

More precisely, let $\mathcal{S} = (NI^* \cup T)^*$ be the set of all strings of terminals and indexed nonterminals. The derivation step relation relation is the minimal binary relation over $\mathcal{S}$ that satisfies the following three axioms. Throughout these axioms, assume A, B and f as above, $\beta \in \mathcal{S}$, $\gamma \in \mathcal{S}$, $\delta \in I^*$, and $z_k \in Z$ for all relevant k.

Axiom 2.1 If $(A \to z_1 \cdots z_i) \in R$, then

$$\beta A \delta \gamma \underset{G'}{\Rightarrow} \beta z_1 \delta_1 z_2 \delta_2 \cdots z_i \delta_i \gamma$$

where, for all $1 \leq k \leq i$, $\delta_k = \delta$ if z_k is nonterminal, and $\delta_k = \lambda$ if z_k is terminal. □

Axiom 2.2 If $(A \to Bf) \in R$, then $\beta A \delta \gamma \underset{G'}{\Rightarrow} \beta B f \delta \gamma$. □

Axiom 2.3 If $(Af \to z_1 \cdots z_i) \in R$, then

$$\beta A f \delta \gamma \underset{G'}{\Rightarrow} \beta z_1 \delta_1 z_2 \delta_2 \cdots z_i \delta_i \gamma$$

where again, $\delta_k = \delta$ if $z_k \in N$, and $\delta_k = \lambda$ otherwise. □

The application of these axioms is illustrated by the following example.

Example 2.11 Let G be the following indexed grammar, where f and n are indices.

$$\begin{array}{rcl} S & \to & Nf \\ N & \to & Nn \\ N & \to & ABC \\ An & \to & Aa \\ Bn & \to & Bb \\ Cn & \to & Cc \\ Af & \to & a \\ Bf & \to & b \\ Cf & \to & c \end{array}$$

The language recognized is $L(G) = \{a^n b^n c^n \mid n \in \mathbb{N}_+\}$.

During the derivation process, S is expanded to Nf, and the N then accumulates n indices for some indeterminate period of time. Then N splits up into nonterminals A B and C, each of which inherits all the accumulated indices. Finally, the indices are unloaded one at a time, with appropriate terminal symbols inserted in a one-for-one exchange. Here is a partial derivation for the terminal string $aaabbbccc$.

$$\begin{aligned} \mathrm{S} &\Rightarrow Nf \Rightarrow Nnf \Rightarrow Nnnf \Rightarrow AnnfBnnfCnnf \Rightarrow AnfaBnnfCnnf \\ &\Rightarrow AfaaBnnfCnnf \Rightarrow aaaBnnfCnnf \Rightarrow \cdots \end{aligned}$$

□

Evidently, the indexed grammars generate a proper superset of the CFLs. There is also an upper bound on the power of the indexed grammar model:

Theorem 2.4 If L is generated by an indexed grammar and $\lambda \notin L$, then L is context-sensitive. □

2.5.2 Comments

Indexed grammars are a good example of a grammar model that defies classification.

They do not particularly resemble attribute grammars, although the index list is surely an inherited attribute of nonterminals.

They do bear a striking resemblance to W-grammars. The distribution of indices by the derivation step relation is uncannily similar to the consistent expansion of meta-variables. In Examples 2.11 and 2.8 these two mechanisms play exactly analogous roles; moreover, the indexed rule $N \to Nn$ corresponds to the meta-rule

N :: Nn.

while $Af \to Aa$ corresponds (not quite as perfectly) to the rule schema

nNL : letter L symbol, N L.

There is also a certain flavor of control-restriction about the way indices regulate the application of grammar rules.

Finally, and of particular interest for this thesis, indexed grammars also bear a certain resemblance to the adaptable grammars of H. Christiansen, to be discussed in §3.3, in which the set of rules applicable to a given branch of the parse tree is an inherited attribute of the parent node.

Chapter 3

Adaptable grammars

3.1 Introduction

Although most programming languages are not context-free per se, it has been observed (the earliest reference known to the current author being [Cara 63]) that they are *locally* context-free, in the sense that the set of permissible expressions within a small region in a particular program can be described by a context-free grammar.

Example 3.1 Consider the following toy programming language, in which programs consist only of integer variable declarations and assignments of one variable to another. The context-free syntax is (ignoring whitespace):

$$
\begin{array}{rcl}
\textit{program} & \rightarrow & \text{“}\mathbf{\{}\text{”} \quad \textit{decl-list} \quad \textit{stmt-list} \quad \text{“}\mathbf{\}}\text{”} \\
\textit{decl-list} & \rightarrow & \lambda \\
\textit{decl-list} & \rightarrow & \textit{decl} \quad \textit{decl-list} \\
\textit{decl} & \rightarrow & \text{“}\mathbf{int}\text{”} \quad \textit{id} \quad \text{“}\mathbf{;}\text{”} \\
\textit{stmt-list} & \rightarrow & \lambda \\
\textit{stmt-list} & \rightarrow & \textit{stmt} \quad \textit{stmt-list} \\
\textit{stmt} & \rightarrow & \textit{id} \quad \text{“}\mathbf{=}\text{”} \quad \textit{id} \quad \text{“}\mathbf{;}\text{”} \\
\textit{id} & \rightarrow & \textit{alpha} \\
\textit{id} & \rightarrow & \textit{alpha} \quad \textit{id} \\
\textit{alpha} & \rightarrow & \text{“}\mathbf{a}\text{”} \mid \text{“}\mathbf{b}\text{”} \mid \cdots \mid \text{“}\mathbf{z}\text{”}
\end{array}
$$

The context-dependent restrictions on valid programs are that an identifier cannot be declared more than once, and that an identifier cannot be used in a statement unless it has been declared.

Here are three programs:

```
{ int x; int y; x = y; }
{ int x; int z; x = y; }
{ int x; int x; }
```

All three programs are syntactically correct, in that they satisfy the CFG. However, only the first program is valid. The statement **x=y;** is illegal in the second program, because **y** is undeclared. The declaration **int x;** is illegal the second time it occurs in the third program (assuming left-to-right language analysis).

Nevertheless there seems to be more regularity to the language than the adjective "context-dependent" suggests. Notice that, given a particular expansion for *decl-list*, the set of legal expansions for *stmt-list* can be exactly defined by a context-free grammar. For example, in the second program one would have:

$$\begin{array}{rcl} \textit{stmt-list} & \rightarrow & \lambda \\ \textit{stmt-list} & \rightarrow & \textit{stmt} \quad \textit{stmt-list} \\ \textit{stmt} & \rightarrow & \textit{id} \quad \text{“=”} \quad \textit{id} \quad \text{“;”} \\ \textit{id} & \rightarrow & \text{“}\mathbf{x}\text{”} \mid \text{“}\mathbf{z}\text{”} \end{array}$$

From this point of view, the effect of a declaration **int** α**;** for any particular identifier α is to add a new rule $\textit{id} \rightarrow \alpha$ to the grammar.

(The prohibition against duplicate declarations is much more difficult to describe in terms of simple modifications to the rule set; this will be discussed in §4.3 and §7.1.4.) □

The notion that the rule set of a grammar should be allowed to vary is called *grammar adaptability*. More precisely, a grammar model will be considered adaptable iff it provides for the explicit manipulation of rule sets.

Although a fair amount of research has been done in this area, none of the models proposed have been widely accepted. In fact, some of the researchers seem to have been substantially unaware of previous work in the field. An exception is the recent work of Henning Christiansen, whose survey of the subject [Chri 90] provided most of the basic references for this chapter.

Part of the disunity of the literature is that no two authors use the same terminology. The name "adaptable grammar" used here is taken from [Chri 90].

3.2 Imperative adaptable grammars

There are two major classes of adaptable grammar models, depending on whether rule sets are "manipulated" by imperative or declarative means. This section addresses the imperative approach.

3.2.1 Explanations

Wegbreit's ECFGs

In the late 1960s, Ben Wegbreit developed an adaptable grammar formalism, as part of an "extensible" programming language called EL1. (Extensible languages will

be mentioned in §3.4.) He called his grammars ECFGs (Extensible Context-Free Grammars). The basic reference is [Wegb 70].

An ECFG G consists of a context-free grammar together with a deterministic finite state transducer (see §1.4.2). The input alphabet of the transducer is T_G; the output alphabet is Z_G, plus some reserved symbols that are used for the unambiguous expression of rules.

The output of the transducer is interpreted as a series of instructions for modifying the rule set of the grammar. An output substring $[\![n \to \alpha]\!]$ instructs to add a rule $n \to \alpha$, an output substring $[\![n \not\to \alpha]\!]$ instructs to remove a rule $n \to \alpha$ (or take no action if this rule is not in the rule set). Reserved symbols are also provided for the construction of an infinite number of new nonterminals; let the infinite alphabet of these constructed nonterminals plus Z_G be denoted Y_G.

The derivation relation is not defined over Y_G^*, but over the cartesian product $T_G^* \times Y_G^*$. Elements of this domain are called *instantaneous descriptions* (IDs). If $\pi = \langle w, \alpha \rangle$ is an ID, then R_π denotes the rule set constructed by modifying the initial rule set R_G according to the instructions contained in the output of the transducer on input w.

Suppose $\pi = \langle w, y\alpha \rangle$ is an ID, with $y \in Y_G$. If $y \in T_G$, then $\langle w, y\alpha \rangle \Rightarrow \langle wy, \alpha \rangle$. Otherwise, $\langle w, y\alpha \rangle \Rightarrow \langle w, \omega\alpha \rangle$ iff $(y \to \omega) \in R_\pi$. IDs of the form $\langle w, \lambda \rangle$ cannot occur on the left side of a derivation step.

The language defined by an ECFG G is:

$$L(G) \quad = \quad \{w \mid \langle \lambda, s_G \rangle \stackrel{*}{\underset{G}{\Rightarrow}} \langle w, \lambda \rangle\}$$

It is proven in [Wegb 70] that the class of ECFLs (languages generated by ECFGs) is a proper superset of the context-free languages, and a proper subset of the recursively enumerable languages. The relationship between ECFLs and CSLs is also considered, but inconclusively.

Mason's DTTs

In the early 1980s, K. P. Mason developed a class of adaptable grammars called DTTs (Dynamic Template Translators). The basic reference is [Maso 84]; the material presented here is adapted from [Maso 87].

Whereas Wegbreit uses a transducer to orchestrate modifications to the rule set, Mason attaches modification instructions to the rules themselves, to be performed when the rule is applied during derivation. However Mason's treatment is further complicated because his "rules" are actually rule schemata à la van Wijngaarden.

A *symbol* of a DTT G is a nonempty string over the *symbol alphabet* S_G. Metavariables, called *templates*, are nonempty strings over the *template alphabet* T_G. Terminal symbols are distinguished by means of special prefixes. The working alphabet of the rule schemata is $M_G = S_G(T_G^* S_G)^*$; elements of M_G are called *template symbols*.

There are no explicit meta-rules. When a template t occurs between letters of the symbol alphabet, i.e., $s_1 t s_2$ where $s_k \in S_G$, t may be replaced by a nonempty string u over S_G provided u does not contain s_2. That is, $u \in (S_G - \{s_2\})^+$.

A rule schema has four parts: (1) A nonterminal template symbol, called the *left-hand side*. (2) A string of template symbols, called the *right-hand side*. (3) A string of actions (explained below). (4) A string of template symbols, called the *output*. Every template occurring in the left-hand side or the output string must also occur at least once on the right-hand side. The domain of all rule schemata for G is denoted P_G.

The domain of all possible actions for G is denoted A_G. An action is an instruction of any of the following forms.

add rule schema p
delete rule schema p
add action b to rule schema p
continue

where $b \in A_G$ and $p \in P_G$. The *continue* instruction does nothing. The *add action* instruction concatenates b onto the end of the action string of p. (A number of key issues, relating to the embedding of rule schemata in actions, are not fully addressed in [Maso 87].)

An instantiation of a rule schema p is formed by selecting expansions from S_G^+ for all of the templates that occur in the right-hand side of p, and substituting these expansions consistently throughout all the parts of p.

The state of a derivation consists of a string of symbols (the input), a set of rule schemata, and another string of symbols (the output). Suppose x, y, and w are strings of symbols, s is a nonterminal symbol, and π is a subset of P_G. Then $\langle xsy, \pi, w\rangle \Rightarrow \langle x\sigma y, \pi', vw\rangle$ if there exists an instantiation r of a rule schema $p \in \pi$ such that:

1. The left-hand side of r is s.
2. The right-hand side of r is σ.
3. Performing the actions of r, from left to right, transforms π to π'.
4. The output of r is v.

The output for a terminal string $w \in L(G)$ is simply the third component of the final state of the derivation of w.

The above presentation reformulates much of Mason's model. In particular, the concept of derivation state occurs only implicitly in [Maso 87]. From the idiosyncrasies of Mason's formulation, it seems likely that Mason developed DTTs without knowledge of W-grammars —which the current author finds rather remarkable, given the deep similarities between the two.

It is proven in [Maso 87] that DTTs are Turing-powerful: For every Turing machine M there exists a DTT G such that $L(M) = L(G)$, and vice versa. The proof does not involve any grammar modification at all. Thus, Mason's alternative formulation of W-grammars does not compromise their computational power.

Burshteyn's modifiable grammars

In 1990, Boris Burshteyn introduced an adaptable grammar formalism as a theoretical foundation for a programming language description language called USSA. The USSA meta-language is described in [Burs 92]; the original paper on the formalism is [Burs 90]. Actually, the USSA meta-language seems to have more in common with Christiansen's grammar model (to be discussed in §3.3) than with Burshteyn's. However, since USSA is definitely not a mathematical model, it does not belong in this survey.

Burshteyn calls his grammars *modifiable grammars*, of which he defines two varieties: *top-down modifiable grammars* and *bottom-up modifiable grammars*. In either case, a modifiable grammar consists of a CFG together with a Turing transducer that halts on all inputs. During parsing, each partial derivation is passed to the transducer. The transducer outputs a list of rules to be added, and a list of rules to be deleted. The rule set is modified accordingly, and the parser uses the new rule set to construct another step of the partial derivation.

The difference between top-down and bottom-up modifiable grammars is the order in which the parser constructs the derivation. Here, a *complete derivation* is taken to be the sequence of rules $r_1 \cdots r_n$ in a leftmost derivation of a terminal string w from the start symbol s_G. In a top-down modifiable grammar, complete derivations are constructed from left to right; in a bottom-up modifiable grammar, they are constructed from right to left.

Note that Burshteyn's approach to adaptability is more general than Wegbreit's or Mason's. In comparison to ECFGs, modifiable grammars use a Turing machine to orchestrate adaptability where ECFGs use a finite automaton. In comparison to DTTs, the Turing transducer of a modifiable grammar has the entire partial derivation at its disposal, whereas the adaptation of a DTT depends only on the most recently applied rule. Not surprisingly, both top-down and bottom-up modifiable grammars are Turing-powerful.

3.2.2 Comments

Parse trees cannot effectively capture the information contained in a derivation under any of these models. Moreover, adaptations are liable to have no reasonable correlation at all with the structure of the parse tree. Worse yet, the only thing that adaptations *do* correlate with under these models is the parsing algorithm used.

This is an inherent property of the imperative approach to adaptability. The only purpose served by the rule set at any given moment is to determine the outcome of

the next decision to be made by the parser. Therefore, the grammar designer must know exactly what decisions will be made by the parser, and in what order; otherwise, there is no way of knowing what adaptations to make.

Hence, in particular, Burshteyn's introduction of two different formalisms, one for top-down parsing and one for bottom-up parsing. Even so, he was criticized in the literature ([Robe 91]) because in [Burs 90] he had made the seemingly innocuous assumption that the parser would proceed from left to right in a single pass. Naturally, Burshteyn's reaction [Burs 92] was to generalize his model to allow multipass parsing.

3.3 Christiansen grammars

This section is the complement to the preceding section (3.2) on imperative adaptable grammars. One might infer that Christiansen's is the only non-imperative adaptable grammar formalism in existence, and one would not be far wrong. Burshteyn's USSA metalanguage was noted in §3.2.1. Another non-imperative approach to adaptability occurs in [Hanf 73]. [Chri 90] appropriately characterizes the latter:

> Hanford and Jones (1973) have suggested a concept of dynamic syntax which is a monstrous device based on the λ-calculus. ... Unfortunately the approach is not given a proper formalization or accompanied with an appropriate notation.

3.3.1 Background

In the mid-1980s, Henning Christiansen introduced an adaptable grammar model as part of his licentiat ($\simeq$ Ph.D.) thesis. The original paper is [Chri 85]; a more developed treatment of the model is [Chri 88a]. As presented in these papers, Christiansen's model is a modified form of extended attribute grammars, in which the entire structure of the parse tree is directly dependent on attribute evaluation. (Cf. higher order attribute grammars, §2.2.4.)

In his more recent papers, Christiansen has reformulated his model in terms of definite clause grammars. This is the form of his model described in [Chri 90]. Recall from §2.2.4 that the difference between EAGs and DCGs is mostly one of notation, except that DCGs may embed arbitrary Prolog code in the rules. [Chri 90] notes a need to resort to this feature of DCGs for certain awkward problems; see §7.1.5. His most recent work [Chri 92a, Chri 92b], although it is still readily viewed as addressing his grammar model, is framed in terms of logic meta-programming.

The RAG model proposed in Part II of the current thesis more closely parallels the notation of EAGs than that of DCGs; therefore, Christiansen grammars are presented here using the earlier, EAG-based notation. For the same reason, the notation of EAGs was discussed much more thoroughly than that of DCGs in §2.2.4; hence, the EAG notation does not require additional explanation.

In giving a name to his model, Christiansen made the unfortunate choice of calling his grammars "generative grammars". (Recall the standard meaning of this term from §0.1.) The later, DCG-based formulation was called "generative clause grammars", a name that presents no such difficulty. Nevertheless, since the earlier form will be used here, and since it is convenient to have a general appellation that covers both versions, the name *Christiansen grammars* will be used hereafter.

3.3.2 Explanation

The formal difference between conventional EAGs and Christiansen grammars is deceptively small. Structurally, a Christiansen grammar is just an EAG such that the first attribute of every nonterminal is inherited and has as its semantic domain the set of all Christiansen grammars. This special attribute is called the *language attribute*.

Recall from §2.2.4 that the derivation relation of an EAG is defined over the combined alphabet of terminal symbols and attributed nonterminal instances, with rewriting based on the set of rule instances of the grammar. The derivation relation for Christiansen is just the same, except that rewriting is based on the set of rule instances of the language attribute of the attributed nonterminal instance that is being rewritten.

More formally, suppose α, β, γ are strings over the combined alphabet, and $A = \langle N \downarrow a_1 \updownarrow a_2 \cdots \updownarrow a_i \rangle$ is an attributed nonterminal instance. Observe that a_1 is a Christiansen grammar (since it is the language attribute of A). The relation $\beta A \gamma \Rightarrow \beta \alpha \gamma$ holds iff $A \to \alpha$ is a rule instance in grammar a_1.

The language generated by a Christiansen grammar G is

$$L(G) \quad = \quad \{w \in T_G^* \mid \langle s_G \downarrow G \updownarrow \cdots \rangle \stackrel{*}{\underset{G}{\Rightarrow}} w\}$$

That is, the set of all terminal strings w such that w is derivable from an attributed nonterminal instance A whose nonterminal is the start symbol and whose language attribute is G.

Example 3.2 Consider the following Christiansen grammar.

$$\begin{array}{rcl}
\langle \textit{alpha-list} \downarrow g \uparrow w \rangle & \to & \langle \textit{alpha} \downarrow g \uparrow w \rangle \\
\langle \textit{alpha-list} \downarrow g \uparrow w_1 w_2 \rangle & \to & \langle \textit{alpha} \downarrow g \uparrow w_1 \rangle \ \langle \textit{alpha-list} \downarrow g \uparrow w_2 \rangle \\
\langle \textit{alpha} \downarrow g \uparrow \text{“}\mathbf{a}\text{”} \rangle & \to & \text{“}\mathbf{a}\text{”} \\
 & \vdots & \\
\langle \textit{alpha} \downarrow g \uparrow \text{“}\mathbf{z}\text{”} \rangle & \to & \text{“}\mathbf{z}\text{”}
\end{array}$$

The workings of this grammar should be fairly clear. The language recognized is the set of nonempty strings of lower-case Roman letters; the synthesized attribute is the terminal string recognized.

Now, here is a Christiansen grammar for the toy programming language of Example 3.1. All of the above rule forms are assumed to be present (therefore there is no need to list them again).

$$\begin{aligned}
\langle \textit{program} \downarrow g_0 \rangle &\rightarrow \text{“}\mathbf{\{}\text{”}\ \langle \textit{decl-list} \downarrow g_0 \uparrow g_1 \rangle\ \langle \textit{stmt-list} \downarrow g_1 \rangle\ \text{“}\mathbf{\}}\text{”} \\
\langle \textit{decl-list} \downarrow g \uparrow g \rangle &\rightarrow \lambda \\
\langle \textit{decl-list} \downarrow g_0 \uparrow g_2 \rangle &\rightarrow \langle \textit{decl} \downarrow g_0 \uparrow g_1 \rangle\ \langle \textit{decl-list} \downarrow g_1 \uparrow g_2 \rangle \\
\langle \textit{decl} \downarrow g \uparrow g \,\&\, \textit{new-rule} \rangle &\rightarrow \text{“}\mathbf{int}\text{”}\ \langle \textit{alpha-list} \downarrow g \uparrow w \rangle\ \text{“}\mathbf{;}\text{”} \\
&\quad \text{where}\ \ \textit{new-rule} = \langle \textit{id} \downarrow h \rangle \rightarrow w \\
\langle \textit{stmt-list} \downarrow g \rangle &\rightarrow \lambda \\
\langle \textit{stmt-list} \downarrow g \rangle &\rightarrow \langle \textit{stmt} \downarrow g \rangle\ \langle \textit{stmt-list} \downarrow g \rangle \\
\langle \textit{stmt} \downarrow g \rangle &\rightarrow \langle \textit{id} \downarrow g \rangle\ \text{“}\mathbf{=}\text{”}\ \langle \textit{id} \downarrow g \rangle\ \text{“}\mathbf{;}\text{”}
\end{aligned}$$

The key to this grammar is the synthesizing expression in the rule form for nonterminal *decl*; this expression adds the rule form $\langle id \downarrow h \rangle \rightarrow w$ to whatever grammar g was inherited through the language attribute. Note that the variable h is local to *new-rule*, and therefore is not bound by the instantiation of the surrounding rule form for nonterminal *decl*; the use of multiple levels of variables in Christiansen grammars will be mentioned again in §7.1.2. (The possibility that identifier w had already been declared will not be addressed in this chapter; see §4.3 and §7.1.4.)

The overall effect of the *decl* rule form is to recognize a declaration **int** α**;** (for arbitrary string of letters α), and synthesize a grammar by adding the rule form $id \rightarrow \alpha$ to the inherited grammar. □

3.3.3 Comments

Observe that the behavior of the grammar is no longer dependent on the order in which the derivation is constructed. Each grammar adaptation is localized to a particular branch of the parse tree. A fringe benefit is that block-structured scope is supported in a particularly natural way. (See §7.1.1.)

Christiansen's later work, since it is based on DCGs, omits the distinction between synthesized and inherited attributes. This is not an unreasonable move. Recall the suggestion from §2.2.5 that the synthetic/inherited distinction crosses over from help to hindrance as the prospective path of computation becomes nontrivial. The crossover point may lie very near the extreme simple end of the spectrum: the deepest levels of intuitive appeal can be very intolerant of complexity. Examples in [Chri 88a] suggest that, though a distinct improvement over their nonadaptable counterparts, Christiansen grammars may still exceed the threshold.

However, there is an important insight to be extracted from the earlier formulation of Christiansen grammars. Observe that the language attribute is *inherited*. There is nothing arbitrary about this convention. In any parse tree, the language attribute of a nonterminal node says nothing —at least, nothing remotely finite— about the ancestors or siblings of the node; on the contrary, it is entirely concerned in a clearly delineated way with constraining the relationship between the node and its *children*.

Notice that we are not discussing direction of computation at all, but rather the way in which the language attribute is understood. In a sense, this is just another facet of the meaning of Chomsky context-free rules discussed in §1.6: a rule such as $S \rightarrow ABC$ is primarily understood as a statement about S, not a statement about A, B, or C.

More will be said about the conceptual role of the language attribute in §4.1.

3.4 Comments

The principle of adaptability may be paraphrased as stating that the act of programming amounts to the construction of a new programming language: declarations add to the language, encapsulation subtracts from it. This view is recommended, for example, in [Chri 88b], even the title of which is "Programming as language development". Christiansen observes that this view is not uncommon in the literature of programming methodology (where it is not related to formal grammars); as examples, he cites [Dahl 72, Wino 79].

The notion of programming as language development can be carried further than Christiansen goes. In §4.1.5 it will motivate the exploration of a deep connection between programming language syntax and semantics. This connection is a linchpin of the RAG grammar model proposed in that chapter —and therefore a linchpin of the entire current thesis.

Some mention should be made of "extensible" languages, a trend that seems to have reached its height with two symposia circa 1970, [Chri 69, Schu 71]. The general idea was that the programmer would be given the means to construct a specialized language for any arbitrary application area by extending a base language. In practice, the name "extensible languages" was applied to everything from SIMULA, to PL/I (because of its bewildering macro facility), to compiler-compilers. The trend lost momentum after the symposia [Stan 75].

Although the concept of extensible languages appears quite similar to the adaptability principle, very little of the research done was accompanied by formal grammar models. An exception is Wegbreit's ECFGs; W-grammars might also be excepted, since ALGOL 68 was painted as an extensible language in the earlier of the two symposia. Also, a pathological case was the extensible language AEPL [Katz 71], in which the language processor was driven by a dynamically modifiable set of Chomsky type 0 grammar rules with programs attached to them.

Part II

Recursive Adaptable Grammars (RAGs)

Chapter 4

Introduction to RAGs

The broad purpose of the design for the RAG model was to construct, if possible, a grammar formalism incorporating adaptability without compromising the formal and conceptual elegance of CFGs. This chapter explains how that broad purpose, together with insights gleaned from the survey of existing models, was translated into guiding principles for the design of a grammar model, and how those principles were implemented in the RAG model.

4.1 Design principles

The survey was undertaken primarily in order to isolate factors that contribute to the relative clarity or opacity of a grammar model. Most, though not all, of the following principles were derived directly from the survey.

4.1.1 Parse trees

In one respect, the results of the survey were more consistent than anticipated: In nearly every case, a direct link was traced between clarity and correspondence to parse trees. In retrospect, one might even add to the observations on CFGs in §1.6, that the form of context-free rules owes its success to its correspondence with the parent-children structure of parse trees. This one-to-many rule structure was identified early in the project as a crucial feature for the design of the proposal.

In order to gain additional power, some means must be used to propagate context-dependent information across the structure of the parse tree. The most successful strategy encountered in the survey seems to be the one due to Knuth (see §2.2), in which the distribution of context-dependent information is carried out locally within the context-free rules. Knuth's strategy was also applied to adaptable grammars, with favorable results, by Christiansen (§3.3). Therefore, that strategy was identified for the current proposal as another likely ingredient of a successful design.

Other features of the AG family of grammars, such as the synthesis/inheritance distinction, will be addressed separately below (especially in §4.4).

4.1.2 Orthogonality

Nonorthogonality, in the form of redundant power, between a CFG kernel and a distinct augmenting facility is a problem common to several of the models surveyed. The consequence is competition between the descriptive functions of the conceptually lucid but computationally weak CFG kernel, and the conceptually obscure but computationally strong augmenting facility. As noted of AGs in §2.2.5, computational exigencies tend to win out, undermining clarity. Christiansen's model (§3.3) inherited the nonorthogonality of AGs, although its more natural descriptions of programming languages (as in [Chri 88a]) seem to have somewhat mitigated the immediate symptoms.

Therefore, a grammar model design was sought that would not have a CFG kernel per se. Instead, a modification of the CFG model was envisioned, in which adaptability would occur naturally —on the theory that the features of the model cannot be nonorthogonal to each other if there is only one of them.

4.1.3 Denoting classes of phrases

A second level of nonorthogonality arises in Christiansen grammars between the nonterminal symbols and the language attribute.

It was noted in §3.3.3 that the language attribute of a given nonterminal node constrains the relationship between that node and its children. More precisely, it restricts the set of rule forms that can be used to select the children of that node. It does *not* determine the rule forms that will be used to select later generations of descendants (grandchildren, etc.). Although, in practice, most children will inherit the language attribute of their parent unchanged, there is no necessity that *any* of them do so in general. The relationship between language attributes of parent and children depends on which rule form is selected from the parent's language attribute.

On the other hand, the formal role of the parent's nonterminal symbol is to restrict the set of rule forms that can be used to select the children. In other words, it serves the same purpose, conceptually, as the language attribute. Based on this observation, it was resolved that the proposal should be designed so that *the set of possible terminal phrases for any given branch of the parse tree is determined entirely by a single value occurring at the root of that branch.*

4.1.4 The derivation step relation

In Chomsky grammars, all computation is performed by successive applications of a single, very elementary derivation step relation. Most of the problems observed with existing models in the survey coincide with either a complex derivation step, or a

second computational facility separate from the derivation step. See especially §2.2 on AGs, and §3.2 on imperative adaptable grammars, in both of which the observed problem was nonorthogonality. W-grammars, §2.3, also exhibited a problem with the derivation step; in that case, it was described in terms of correspondence to parse trees.

It was decided that the proposal should be designed to perform all computation by means of an elementary derivation step relation, preferably based on simple rewriting. However, nontrivial expression evaluation was also intended. The desire to satisfy both of these goals led directly to one of the most uniquely characteristic features of the proposal, the "recursive" nature from which the name RAG is derived.

4.1.5 Syntax as semantics

The notion of programming as language development (§3.4) suggests that the entities defined in a program may be thought of as meta-syntactic in nature (in other words, they are statements about the syntax of the language). This view could be treated as an informal principle, but one could also rephrase it more formally as a statement that *the domains of semantic values and meta-syntactic values are identical.*

The proposal takes this unification still further, by requiring that the domain of *syntactic* values, such as terminal strings, is also identical to the combined semantic/meta-syntactic domain. Besides greatly simplifying the model and enhancing orthogonality, this triple unification also facilitates the technical solution to the objectives for the derivation step relation.

The triple unification was foreshadowed by the Christiansen grammar of Example 3.2, in which many of the nonterminals had just two attributes: the inherited language attribute, and a synthesized attribute with either the same (meta-syntactic) domain, or the (syntactic) domain of terminal strings. A more extensive example in [Chri 88a] also supports the unification.

4.2 Explanation

This subsection explains how the RAG model works. Although some motivating references are made to design principles, high-level perspective is often deferred to §4.4.

RAGs are not described here in full generality. Instead, only the subclass of *string RAGs* is presented, in which the terminal syntactic forms are strings over a finite alphabet. The general case is developed rigorously in Chapter 5.

4.2.1 Answers

The combined meta-syntactic/semantic/syntactic domain of a RAG G is called the *answer algebra* of G, and denoted A_G; elements of A_G are called *answers*. Formally,

A_G is a *one-sorted algebra*, defined by a set of operators and a set of identity laws; the formal machinery of one-sorted algebras will be developed in §5.2, and will not be needed until thereafter.

As meta-syntax, answers are essentially functions that map syntactic values into sets of possible semantic values. A syntactic value that maps into the empty set is not part of the language defined by the answer; a syntactic value that maps into a set with more than one element is semantically ambiguous. Operators are provided for constructing new functions from existing ones. For example, most answer algebras include a "union" operator that represents the union of the functions represented by its arguments.

Each RAG also has a designated subalgebra of A_G called the *terminal algebra*, denoted T_G; elements of T_G are called *terminals*. The terminal algebra corresponds to the set of all terminal strings of a Chomsky grammar. The syntactic role in the model is most often played by terminals, although of course any answer *could* serve this purpose.

Another group of operators is needed to support the syntactic role of terminals. Typically, these operators include the letters of an alphabet Z, which are *constants*, i.e., zero-ary operators; the empty string λ, also a constant; and the binary concatenation operator. These operators are related by the usual identities on strings:

$$\begin{aligned} \lambda \cdot v_0 &= v_0 \\ v_0 \cdot \lambda &= v_0 \\ v_0 \cdot (v_1 \cdot v_2) &= (v_0 \cdot v_1) \cdot v_2 \end{aligned}$$

Recall from §0.5 that $\cdot$ is the explicit infix concatenation operator. Usually, the explicit operator will be used only in polynomial expressions over the answer algebra, such as the above (and more generally, polynomial expressions over the query algebra of §4.2.3, below).

4.2.2 Rules

An *unbound rule* is a construction of the form

$$\langle v_0, e_0 \rangle \quad \rightarrow \quad t_0 \langle e_1, v_1 \rangle t_1 \cdots \langle e_{n-1}, v_{n-1} \rangle t_{n-1} \langle e_n, v_n \rangle t_n$$

where $n \geq 0$, the v_k are distinct variables, the t_k are answers (usually, but not necessarily, terminal), and the e_k are polynomials in variables $v_0, \cdots v_n$.

The ordered pairs of polynomials[1] in an unbound rule, $\langle v_0, e_0 \rangle$ and all $\langle e_k, v_k \rangle$, are called *unbound pairs*. They are analogous to the *attributed nonterminal forms* of an EAG (§2.2.4). The left-hand component of each pair plays the same meta-syntactic role as the language attribute in Christiansen grammars. The t_k on the right side of the rule act as syntax, in that they become elements of the (usually terminal) string that is ultimately generated by the grammar.

[1] A variable is considered a polynomial; for the formal definition, see §5.2.

The right-hand component of a pair is usually interpreted as a synthesized attribute whose value encapsulates the semantics of the generated terminal string. In a derivation $\langle a, c\rangle \stackrel{*}{\Rightarrow} b$, with answers $a, b, c \in A_G$, the meta-syntactic value a is said to synthesize (or compute) semantic value c when generating syntactic value b, or equivalently, to assign semantic value c to b. The relevance of the concept of synthesis to the semantic value in a RAG derivation will be explored in §4.4. There are, however, also semantic issues that go far beyond the scope of the current thesis; see §7.2.3.

The domain $\mathcal{R}_G$ of unbound rules is related to the domain of answers by means of a *rule function*, $\rho_G : A_G \to \mathcal{P}_\omega(\mathcal{R}_G)$. When binding the variables of an unbound rule r as above, the leftmost variable v_0 must be bound to some answer $a \in A_G$ such that $r \in \rho_G(a)$.

Example 4.1 Let $G_{4.1}$ be the following recursive adaptable grammar.

$$\rho(S) \;=\; \left\{ \begin{array}{lcl} \langle v_0, \lambda\rangle & \to & \lambda \\ \langle v_0, \lambda\rangle & \to & a\langle v_0, v_1\rangle b \end{array} \right\}$$

By convention, the rule function is specified only for nonterminal operators, i.e., operators in the answer algebra A_G that are not in the terminal algebra T_G. Therefore, by implication, this grammar has one nonterminal constant, S, and three terminal constants, a, b, and λ. (There is also the binary concatenation operator, which is terminal, so that the terminal algebra $T_G = \{a, b\}^*$.) S is the start symbol.

In order to reduce the amount of clutter on the printed page, the following shorthand notation will be used hereafter to specify the rule function for nonterminal constants.

$$\begin{array}{lcll} S\colon & \langle v_0, \lambda\rangle & \to & \lambda \\ & \langle v_0, \lambda\rangle & \to & a\langle v_0, v_1\rangle b \end{array}$$

This convention will greatly enhance the readability of later RAG definitions. Whenever it becomes necessary to define non-constant operators, the more explicit $\rho() = \{\}$ notation will be used.

Since the above unbound rules belong to $\rho(S)$, their instantiations will all bind $v_0 = S$. The partially bound rules are:

$$\begin{array}{lcl} \langle S, \lambda\rangle & \to & \lambda \\ \langle S, \lambda\rangle & \to & a\langle S, v_1\rangle b \end{array}$$

The second rule still has one unbound variable. Technically, it would be legal to bind v_1 to any answer at all, producing a derivation step such as:

$$\langle S, \lambda\rangle \;\Rightarrow\; a\langle S, S\rangle b$$

But this derivation is a dead end. It cannot ever generate a terminal string, because the bound pair $\langle S, S\rangle$ cannot occur on the left side of any bound rule.

Therefore, the only *useful* binding for v_1 is λ. The fully bound rules are then

$$\begin{array}{lcl} \langle S, \lambda \rangle & \rightarrow & \lambda \\ \langle S, \lambda \rangle & \rightarrow & a\langle S, \lambda \rangle b \end{array}$$

The terminals generated are all strings of the form $a^n b^n$. The associated semantic value is always λ, no matter which terminal was generated.

Compare the above fully bound rules with the context-free grammar of Example 1.1. □

The preceding example did not involve the rule set of any answer other than the start symbol. The rule function was largely superfluous. However, in any reasonably well-behaved RAG, every operator in A_G can be understood as a constructor of partial functions (from syntax to semantics), and ρ_G is the tool that expresses that construction. Some canonical examples are:

Proposition 4.1 For a RAG G,

$$\begin{array}{rrcl} \forall a, b \in A_G, & \rho_G(a \sqcup b) & = & \left\{ \begin{array}{lcl} \langle v_0, v_1 \rangle & \rightarrow & \langle a, v_1 \rangle \\ \langle v_0, v_1 \rangle & \rightarrow & \langle b, v_1 \rangle \end{array} \right\} \\ \forall a, b \in A_G, & \rho_G([a, b]) & = & \{\ \langle v_0, b \rangle \ \rightarrow \ \langle a, v_1 \rangle\ \} \\ \forall t \in T_G, & \rho_G(t) & = & \{\ \langle v_0, \lambda \rangle \ \rightarrow \ t\ \} \\ & \rho_G(\emptyset) & = & \{\} \end{array}$$

(where, as explained in §4.2.1 above, A_G and T_G are the answer algebra and terminal algebra of G, respectively). □

The binary operator in the first equation, denoted $a \sqcup b$, is called the *union* operator (because the partial function it constructs is the union of the partial functions represented by its arguments). The binary operator in the second equation, denoted $[a, b]$, is called the *mapping* operator (because the partial function it constructs *maps* syntactic values from the language of its first argument into the semantic value represented by its second argument). The third equation establishes the basic relationship between the syntactic and meta-syntactic uses of terminals: each terminal t generates the singleton language $\{t\}$, and assigns itself the semantic value λ. The last equation introduces $\emptyset$ as a nonterminal constant whose rule set is empty. ($\emptyset$ must be nonterminal, because the previous equation guarantees that every terminal has a singleton rule set.)

The concatenation operator is less easily expressed in terms of the rule function; the formal treatment in §6.3.2, though straightforward, requires a higher degree of formal precision than has been observed in this section. However, the overall effect of concatenation can also be captured by means of the derivation relation.

Proposition 4.2 For all $a_k, b_k \in A_G$, if $\langle a_0, a_1 \rangle \overset{*}{\underset{G}{\Rightarrow}} a_2$ and $\langle b_0, b_1 \rangle \overset{*}{\underset{G}{\Rightarrow}} b_2$ then $\langle a_0 b_0, a_1 b_1 \rangle \overset{*}{\underset{G}{\Rightarrow}} a_2 b_2$. □

Proposition 4.3 For all $a_0, b_0, w_1, w_2 \in A_G$, if $\langle a_0 b_0, w_1 \rangle \overset{*}{\underset{G}{\Rightarrow}} w_2$, then there exist $a_1, a_2, b_1, b_2 \in A_G$ such that $a_1 b_1 = w_1$, $a_2 b_2 = w_2$, $\langle a_0, a_1 \rangle \overset{*}{\underset{G}{\Rightarrow}} a_2$ and $\langle b_0, b_1 \rangle \overset{*}{\underset{G}{\Rightarrow}} b_2$. □

There is, by the way, a formal well-behavedness criterion for RAG rule functions; this criterion will be developed in §6.2. All of the rule functions discussed in this chapter are assumed to be well-behaved, and the conditions placed on them —such as Proposition 4.1— are consistent with that assumption.

The following example illustrates some of the elegance that can accrue from associating a rule set with each terminal.

Example 4.2 Suppose Z is any alphabet, and consider the following RAG.

$$
\begin{array}{lrcl}
S\text{:} & \langle v_0, v_3 \rangle & \rightarrow & \langle W, v_1 \rangle \langle v_1, v_2 \rangle \langle v_1, v_3 \rangle \\
\\
W\text{:} & \langle v_0, \lambda \rangle & \rightarrow & \lambda \\
 & \langle v_0, v_1 v_2 \rangle & \rightarrow & \langle v_0, v_1 \rangle \langle C, v_2 \rangle \\
\\
C\text{:} \quad \forall z \in Z, & \langle v_0, z \rangle & \rightarrow & z
\end{array}
$$

The terminal constants are λ and the elements of Z. The nonterminal constants are S, W, and C. S is the start symbol.

Nonterminal C generates any one letter $z \in Z$, and associates with it the semantic value z.

Building on the effect of C, W generates any string $w \in Z^*$ and assigns it semantic value w. Cf. the Christiansen grammar of Example 3.2 (specifically, nonterminal *alpha-list*).

Finally, at the top level, S uses W to generate an arbitrary $w \in Z^*$. The generated string w is bound to v_1. Each of the remaining two pairs also generates w, because (from Proposition 4.1):

$$\rho(w) \quad = \quad \{ \ \langle w, \lambda \rangle \quad \rightarrow \quad w \ \}$$

The remaining variables, v_2 and v_3, are both bound to λ. The overall effect is that S generates exactly the language $L_G(S) = \{www \mid w \in Z^*\}$, and associates semantic value λ indiscriminately with every string generated. □

An answer that assigns semantic value λ to every string it generates is said to *predicate* the language it generates (or equivalently, the answer is said to be a predicate). The start symbol S in the above example predicates the language $\{www \mid w \in Z^*\}$. By Proposition 4.1, each terminal string $t \in T_G$ predicates the singleton language $\{t\}$.

4.2.3 Queries

One of the objectives mentioned in §4.1 was that a means be provided for evaluating nontrivial expressions by successive application of the derivation step relation.

A means has already been described for performing nontrivial computations by means of derivation. Given $a, b, c \in A_G$, the derivation relation $\langle a, c\rangle \overset{*}{\underset{G}{\Rightarrow}} b$ may be understood to say that the meta-syntactic value a computes semantic value c when recognizing syntactic value b. (In general, there may be more than one possible value of c for given a and b.)

In order to meet the design objective, we introduce a special binary operator called the *query operator*, written $a:b$, that denotes a semantic value associated by meta-syntactic value a with syntactic value b. That is,

Proposition 4.4 For all $a, b, c \in A_G$, $a:b \overset{*}{\underset{G}{\Rightarrow}} c$ iff $\langle a, c\rangle \overset{*}{\underset{G}{\Rightarrow}} b$. □

(This proposition will be developed formally in Chapter 5 as Theorem 5.12.)

Note that, technically, the query operator is *not* part of the answer algebra A_G. It belongs to a larger algebra Q_G, called the *query algebra*, of which A_G is a proper subalgebra. When a formal condition (such as Proposition 4.4) stipulates an *answer*, the query operator is excluded.

The practical use of queries will be illustrated in §4.3.

4.2.4 The derivation step

It may fairly be asked what kind of "elementary" derivation step relation would give rise to Proposition 4.4.

The two derivation relations in the proposition are, in a sense, working in opposite directions. The relation $\langle a, c\rangle \overset{*}{\Rightarrow} b$ implicitly constructs a parse tree from the top down, while $a:b \overset{*}{\Rightarrow} c$ may be thought of as reconstructing the same parse tree from the bottom up. The RAG model embraces this interpretation by introducing a special unary operator called the *(derivational) inverse* operator, denoted $\overline{q}$, as follows.

Just as the query operator is not part of the answer algebra A_G, the inverse operator is not part of the query algebra Q_G. Rather, the inverse operator belongs to a still larger algebra C_G, called the *configuration* algebra, of which Q_G is a proper subalgebra. When a formal condition stipulates a query, the inverse operator is excluded. Figure 4.1 summarizes the hierarchy of algebras associated with G.

The basic properties of the inverse operator are:

$$\forall a \in A_G, \qquad \overline{a} = a$$

$$\forall c \in C_G, \qquad \overline{\overline{c}} = c$$

$$\forall x, y \in C_G, \quad x \Rightarrow y \text{ iff } \overline{y} \Rightarrow \overline{x}$$

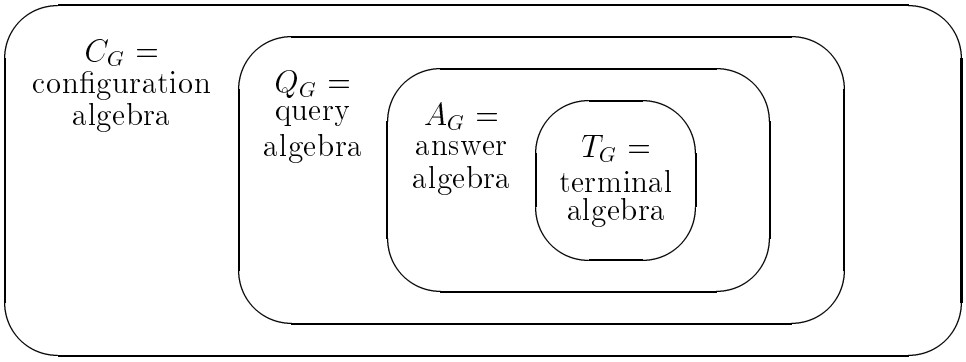

Figure 4.1: Hierarchy of algebras for a RAG G

The configuration algebra is intended to be the largest domain over which the derivation step relation is defined. Therefore, since RAG derivations always involve binary pairs, the binary pairing operator $\langle , \rangle$ is included in the configuration algebra. Given that binary pairs belong to C_G, the above properties of the inverse algebra imply immediately that

$$\forall a, b, c \in A_G, \quad \langle a, c \rangle \overset{*}{\underset{G'}{\Rightarrow}} b \quad \text{iff} \quad b \overset{*}{\underset{G'}{\Rightarrow}} \overline{\langle a, c \rangle}$$

(In fact, this result would hold for any $a, c \in C_G$ provided that $b \in A_G$.)

The derivation step relation of a RAG can be applied to a subexpression in just the same way that the step relation of a Chomsky grammar can be applied to a substring (Axiom 1.2):

Proposition 4.5 Suppose σ is an operator in C_G with arity $ar(\sigma) = n \geq 1$, and σ is not the inverse operator. Further, suppose $c_1, \cdots c_n \in C_G$, and $c_k \Rightarrow c'_k$ for some $1 \leq k \leq n$. Then

$$\sigma(c_1, \cdots c_{k-1}, c_k, c_{k+1}, \cdots c_n) \quad \underset{G'}{\Rightarrow} \quad \sigma(c_1, \cdots c_{k-1}, c'_k, c_{k+1}, \cdots c_n)$$

□

In particular, $b \overset{*}{\underset{G'}{\Rightarrow}} \overline{\langle a, c \rangle}$ implies $a : b \overset{*}{\underset{G'}{\Rightarrow}} a : \overline{\langle a, c \rangle}$. Proposition 4.4 now requires only a simple rewriting axiom,

$$\forall a, c \in A_G, \quad a : \overline{\langle a, c \rangle} \quad \underset{G'}{\Rightarrow} \quad c$$

so that $a : b \overset{+}{\underset{G'}{\Rightarrow}} a : \overline{\langle a, c \rangle} \underset{G'}{\Rightarrow} c$.

The complete axioms for the derivation step relation are enumerated and discussed in §5.3 (Definition 5.23 and following). Proposition 4.4 is restated and proven in §5.4 (Theorem 5.12 and following).

4.3 Examples using the query operator

The toy programming language of Examples 3.1 and 3.2 had two context-dependent restrictions: an identifier cannot be declared more than once, and an identifier cannot be used in a statement unless it has been declared. The principle of adaptability neatly addressed the latter restriction, but the former restriction was glossed over in both examples.

To enforce this restriction, there must be a way to verify that an identifier has *not* already been declared. This amounts to recognizing the complement of the language of identifiers formed by the previous declarations.

Now, the RAGs generate exactly the recursively enumerable languages, and the recursively enumerable languages are not closed under complementation; so there is no universal way to generate the complement of a language in the RAG model. However, a specialized solution can be constructed for the immediate purpose.

Example 4.3 The following construction develops a very limited form of language complementation. In the process, a number of RAG techniques are illustrated, including the use of embedded queries in unbound rules.

Let $Z = \{\text{“}\mathbf{a}\text{”}, \cdots \text{“}\mathbf{z}\text{”}\}$ be a subset of T_G. Here are some elementary operators that will be useful during the construction.

$$\textit{letter}\colon\quad \forall z \in Z,\quad \langle v_0, \lambda\rangle \;\rightarrow\; z$$

$$\textit{echo}\colon\quad \forall z \in Z,\quad \langle v_0, z\rangle \;\rightarrow\; z$$

Nonterminal constant *letter* generates any single letter in alphabet Z, and assigns it semantic value λ. (That is, *letter* predicates Z). Nonterminal constant *echo* also generates any single letter $z \in Z$, but assigns it semantic value z.

The construction will require string-handling operators analogous to the letter-handling operators *letter* and *echo*. Rather that define specialized nonterminal constants for the purpose, let *star* be a nonterminal unary operator (i.e., a unary operator in A_G but not in T_G), as follows.

$$\rho(\textit{star}(a)) \;=\; \left\{ \begin{array}{rcl} \langle v_0, \lambda\rangle & \rightarrow & \lambda \\ \langle v_0, v_1 v_2\rangle & \rightarrow & \langle a, v_1\rangle\langle \textit{star}(a), v_2\rangle \end{array} \right\}$$

Here, by convention, the argument a of the operator is understood to be universally quantified over A_G.

For each answer $a \in A_G$, operator *star* constructs a new answer $\textit{star}(a) \in (A_G - T_G)$ that generates the concatenation of zero or more strings generated by a, and synthesizes the concatenation of the semantic values synthesized by a. For example,

$$\begin{array}{lrcl} \textit{star}(\textit{letter})\colon & \langle v_0, \lambda\rangle & \rightarrow & \lambda \\ & \langle v_0, v_1 v_2\rangle & \rightarrow & \langle \textit{letter}, v_1\rangle\langle \textit{star}(\textit{letter}), v_2\rangle \end{array}$$

Thus, $star(letter)$ predicates the language Z^*.

Similarly, $star(echo)$ also generates language Z^*, but assigns to each generated $w \in Z^*$ the semantic value w. That is, for all $a, b \in A_G$,

$$(star(echo) : a) \stackrel{*}{\Rightarrow} b \quad \text{iff} \quad a \in Z^* \text{ and } b = a$$

For each terminal constant in Z, define a nonterminal constant, as follows.

$$\forall z \in Z, \quad notlet_z\text{:} \quad \forall y \in (Z - \{z\}), \quad \langle v_0, \lambda \rangle \quad \rightarrow \quad y$$

Each nonterminal $notlet_z$ has exactly twenty five rules, and predicates the alphabet $Z - \{z\}$. That is, for all $z \in Z$ and $a, b \in A_G$,

$$(notlet_z : a) \stackrel{*}{\Rightarrow} b \quad \text{iff} \quad a \in (Z - \{z\}) \text{ and } b = \lambda$$

Building on this,

$$notlet\text{:} \quad \forall z \in Z, \quad \langle v_0, notlet_z \rangle \quad \rightarrow \quad z$$

Nonterminal $notlet$ has exactly twenty-six rules. It generates the alphabet Z, and associates semantic value $notlet_z$ with each $z \in Z$. That is, for all $a, b \in A_G$,

$$(notlet : a) \stackrel{*}{\Rightarrow} b \quad \text{iff} \quad a \in Z \text{ and } b = notlet_a$$

Suppose $a, b, c \in A_G$, and $(notlet : a) : b \stackrel{*}{\Rightarrow} c$. Then there must be some $d \in A_G$ such that

$$(notlet : a) : b \quad \stackrel{*}{\Rightarrow} \quad d : b \quad \stackrel{*}{\Rightarrow} \quad c$$

where $notlet : a \stackrel{*}{\Rightarrow} d$. But $notlet : a \stackrel{*}{\Rightarrow} d$ implies that $a \in Z$ and $d = notlet_a$; and furthermore, $notlet_a : b \stackrel{*}{\Rightarrow} c$ implies that $b \in (Z - a)$ and $c = \lambda$.

Thus, for all $a, b, c \in A_G$,

$$((notlet : a) : b) \stackrel{*}{\Rightarrow} c \quad \text{iff} \quad a \in Z \text{ and } b \in (Z - a) \text{ and } c = \lambda$$

Finally, using all of the above operators,

$$\begin{aligned} not\text{:} \quad & \langle v_0, letter \cdot star(letter) \rangle \quad \rightarrow \quad \lambda \\ & \langle v_0, \lambda \sqcup ((notlet : v_1) \cdot star(letter)) \\ & \qquad \sqcup (v_1 \cdot (not : v_2)) \rangle \quad \rightarrow \quad \langle echo, v_1 \rangle \langle star(echo), v_2 \rangle \end{aligned}$$

The first rule generates terminal string λ, and assigns it a semantic value that predicates the language Z^+. The second rule cannot generate λ; therefore, for all $w, a \in A_G$,

$$((not : \lambda) : w) \stackrel{*}{\Rightarrow} a \quad \text{iff} \quad w \in Z^+ \text{ and } a = \lambda$$

The right side of the second rule generates an arbitrary nonempty string over Z, and binds v_1 to the first letter of the string and v_2 to the rest of the string. The associated semantic value has three parts:

- λ is an answer that predicates the terminal string λ.
- $((notlet : v_1) \cdot star(letter))$ predicates any nonempty string over Z that does not begin with the letter v_1.
- $(v_1 \cdot (not : v_2))$ predicates the letter v_1 concatenated with some string predicated by $not : v_2$

By induction, this rule predicates every string over Z that does not equal $v_1 v_2$. That is, for all $w, a, b \in A_G$,

$$((not : w) : a) \stackrel{*}{\Rightarrow} b \quad \text{iff} \quad w \in Z^* \text{ and } a \in (Z^* - \{w\}) \text{ and } b = \lambda$$

Thus, nonterminal not generates the language Z^*, and for each generated string $w \in Z^*$, synthesizes a predicate for the language $Z^* - \{w\}$.

Figure 4.2 lists the definitions of the nonterminal operators from this example. □

$$\rho(star(a)) \;=\; \left\{ \begin{array}{rcl} \langle v_0, \lambda \rangle & \rightarrow & \lambda \\ \langle v_0, v_1 v_2 \rangle & \rightarrow & \langle a, v_1 \rangle \langle star(a), v_2 \rangle \end{array} \right\}$$

$$\begin{array}{lrcl} letter\text{:} & \forall z \in Z, \quad \langle v_0, \lambda \rangle & \rightarrow & z \\ echo\text{:} & \forall z \in Z, \quad \langle v_0, z \rangle & \rightarrow & z \\ \forall z \in Z, \quad notlet_z\text{:} & \forall y \in (Z - \{z\}), \quad \langle v_0, \lambda \rangle & \rightarrow & y \\ notlet\text{:} & \forall z \in Z, \quad \langle v_0, notlet_z \rangle & \rightarrow & z \\ not\text{:} & \langle v_0, letter \cdot star(letter) \rangle & \rightarrow & \lambda \\ & \langle v_0, \lambda \sqcup ((notlet : v_1) \cdot star(letter)) & & \\ & \qquad \sqcup (v_1 \cdot (not : v_2)) \rangle & \rightarrow & \langle echo, v_1 \rangle \langle star(echo), v_2 \rangle \end{array}$$

Figure 4.2: Grammar for a limited language complementation operator

Using the tools developed in the preceding example, it is fairly straightforward to construct a RAG that generates exactly the toy language of Examples 3.1 and 3.2. The construction in the following example deliberately imitates the grammars used in those two examples as much as possible, so as to provide a comparison and contrast of basic functionality between the grammar models.

Example 4.4 The following construction develops a RAG that generates exactly the toy language of Examples 3.1 and 3.2. The entire construction from Example 4.3 is assumed, including the alphabet Z.

In the Christiansen grammar for the language, each declaration added a rule form $\langle id \downarrow h \rangle \rightarrow w$ to the grammar. Because of the nonterminal *id* on its left side, the new rule form could only be applied to rewrite occurrences of that nonterminal.

The analogous problem in the RAG construction, of separating the sublanguage of identifiers from that of declarations, is handled by passing an 'environment' of identifiers as a parameter to unary operator *decl*, and synthesizing a new environment as the semantics of the declaration. A further separation must be maintained, though, within each environment, between the increasing sublanguage of declared identifiers, and the descreasing sublanguage of undeclared identifiers. The encoding of both sublanguages within a single answer is accomplished by using a query on the single answer to extract one or the other sublanguage. To avoid confusion, special "reserved" query keys are introduced for the purpose, that cannot occur in any terminal string because they are nonterminals.

$$\mathit{undef}\colon \quad \langle v_0, \lambda \rangle \quad \rightarrow \quad \mathit{undef}$$

$$\mathit{def}\colon \quad \langle v_0, \lambda \rangle \quad \rightarrow \quad \mathit{def}$$

Recall that, by convention, the rule sets of terminal symbols are not explicitly listed when specifying a RAG. Therefore, the explicit listing of rule sets for *undef* and *def* unambiguously designates those symbols as nonterminals.

Here is the start symbol of the grammar.

$$\begin{array}{lrcl} \mathit{prog}\colon & \langle v_0, \lambda \rangle & \rightarrow & \text{“}\mathbf{\{}\text{”}\ \langle \mathit{decl\text{-}list}([\mathit{undef}, \mathit{star}(\mathit{letter})] \sqcup [\mathit{def}, \emptyset]), v_1 \rangle \\ & & & \quad \langle \mathit{star}(\mathit{stmt}(v_1)), v_2 \rangle\ \text{“}\mathbf{\}}\text{”} \end{array}$$

Let $d \in A_G$ be the expression $[\mathit{undef}, \mathit{star}(\mathit{letter})] \sqcup [\mathit{def}, \emptyset]$ from the above rule. Then

$$\rho(d) \quad = \quad \left\{ \begin{array}{rcl} \langle v_0, \mathit{star}(\mathit{letter}) \rangle & \rightarrow & \langle \mathit{undef}, v_1 \rangle \\ \langle v_0, \emptyset \rangle & \rightarrow & \langle \mathit{def}, v_1 \rangle \end{array} \right\}$$

The latter two rules synthesize predicates for, by intent, respectively all undeclared identifiers $(d\colon \mathit{undef})$, and all declared identifiers $(d\colon \mathit{def})$. Since d represents an environment with no prior declarations, all identifiers are undeclared and none are declared.

These two rules cannot be applied directly to the parse tree, since they do not generate terminal strings. They are meant to be passed up and down the parse tree —for example, they will be inherited by the rules in $\rho(\mathit{decl\text{-}list}(d))$— and will only affect the derivation if they are deliberately invoked by a query embedded in a rule.

Here is the definition of operator *decl-list*.

$$\rho(\mathit{decl\text{-}list}(e)) \quad = \quad \left\{ \begin{array}{rcl} \langle v_0, e \rangle & \rightarrow & \lambda \\ \langle v_0, v_2 \rangle & \rightarrow & \langle \mathit{decl}(e), v_1 \rangle\ \langle \mathit{decl\text{-}list}(v_1), v_2 \rangle \end{array} \right\}$$

The operator is parameterized on an environment e (i.e., a nonterminal, such as d above, that synthesizes predicates for declared and undeclared symbols). For an

empty list of declarations, e is synthesized unchanged; for a nonempty list, e is passed down to the first declaration, which synthesizes a new environment e' that becomes the environment for the rest of the list.

(All of this bears some ressemblance to the common practice of passing symbol tables down and up the parse tree in an ordinary attribute grammar. However, the uniform data domain of the answer algebra sheds a different light on the technique, highlighting the deep connections among meta-syntax, syntax, and semantics.)

The following operators facilitate modification of the environment.

$$\rho(and(a,b)) \;=\; \{\; \langle v_0, (a:v_1)\cdot(b:v_1)\rangle \;\rightarrow\; \langle star(echo), v_1\rangle \;\}$$

Given two predicates a and b, binary operator *and* constructs a new predicate $and(a,b)$ such that $L(and(a,b)) = L(a) \cap L(b) \cap Z^*$.

$$\rho(combine(a,b)) \;=\; \left\{ \begin{array}{rcl} \langle v_0, and(a:undef, b:undef)\rangle & \rightarrow & undef \\ \langle v_0, (a:def) \sqcup (b:def)\rangle & \rightarrow & def \end{array} \right\}$$

Given two environments a and b, binary operator *combine* creates a new environment; an identifier is undeclared in $combine(a,b)$ iff it is undeclared in both a and b, and declared in $combine(a,b)$ iff it is declared in either a or b (or both).

$$\textit{make-env}\colon\;\; \langle v_0, [undef, not:t] \sqcup [def, v_1]\rangle \;\rightarrow\; \langle star(echo), v_1\rangle$$

Nonterminal constant *make-env* generates any string t over alphabet Z, and synthesizes a new environment, in which identifier t is declared and all other identifiers are undeclared.

Here, then, is the definition of operator *decl*.

$$\rho(decl(e)) \;=\; \left\{ \begin{array}{rcl} \langle v_0, combine(e, \textit{make-env}:v_1)\rangle & & \\ \quad \rightarrow \;\; \text{“}\textbf{int}\text{”}\; \langle echo\cdot star(echo), v_1\rangle & & \\ \qquad\qquad \langle (e:undef):v_1, v_2\rangle\; \text{“}\textbf{;}\text{”} & & \end{array} \right\}$$

If identifier v_1 has already been declared, the expression $(e:undef):v_1$ is unevaluable; i.e., it cannot generate an answer. Otherwise, it evaluates (in some number of derivation steps) to λ, and thus has no effect on the terminal string generated.

Note that, since $\lambda\cdot a = a$ for all $a \in A_G$, the expression $(e:undef):v_1$ would have had exactly the same effect no matter where it was put in the rule: It would disable the rule for declared terminals, and have no effect for undeclared terminals. One could even have written

$$\begin{array}{l} \langle v_0, combine(e, \textit{make-env}:v_1)\rangle \\ \qquad\qquad \rightarrow \;\; \text{“}\textbf{int}\text{”}\; \langle echo\cdot star(echo)\cdot((e:undef):v_1), v_1\rangle\; \text{“}\textbf{;}\text{”} \end{array}$$

which works just as well from a logical point of view, since the derivation still cannot generate a terminal unless v_1 is bound to an undeclared identifier. (However, this rule is *circular*; see §6.6.)

$$\rho(and(a,b)) \quad = \quad \{\ \langle v_0, (a:v_1)\cdot(b:v_1)\rangle \ \rightarrow\ \langle star(echo), v_1\rangle\ \}$$

$$\rho(combine(a,b)) \quad = \quad \left\{ \begin{array}{rcl} \langle v_0, and(a:undef, b:undef)\rangle & \rightarrow & undef \\ \langle v_0, (a:def)\sqcup(b:def)\rangle & \rightarrow & def \end{array} \right\}$$

$$\rho(decl\text{-}list(e)) \quad = \quad \left\{ \begin{array}{rcl} \langle v_0, e\rangle & \rightarrow & \lambda \\ \langle v_0, v_2\rangle & \rightarrow & \langle decl(e), v_1\rangle\ \langle decl\text{-}list(v_1), v_2\rangle \end{array} \right\}$$

$$\rho(decl(e)) \quad = \quad \left\{ \begin{array}{l} \langle v_1, combine(e, make\text{-}env:v_1)\rangle \\ \quad \rightarrow\ \text{“}\mathbf{int}\text{”}\ \langle echo\cdot star(echo), v_1\rangle \\ \qquad\qquad \langle (e:undef):v_1, v_2\rangle\ \text{“}\mathbf{;}\text{”} \end{array} \right\}$$

$$\rho(stmt(e)) \quad = \quad \{\ \langle v_1, \lambda\rangle \ \rightarrow\ \langle e:def, v_1\rangle\ \text{“}\mathbf{=}\text{”}\ \langle e:def, v_2\rangle\ \text{“}\mathbf{;}\text{”}\ \}$$

$$prog\text{:} \quad \langle v_0, \lambda\rangle \ \rightarrow\ \text{“}\mathbf{\{}\text{”}\ \langle decl\text{-}list([undef, star(letter)]\sqcup[def, \emptyset]), v_1\rangle\ \langle star(stmt(v_1)), v_2\rangle\ \text{“}\mathbf{\}}\text{”}$$

$$make\text{-}env\text{:} \quad \langle v_0, [undef, not:v_1]\sqcup[def, v_1]\rangle \rightarrow\ \langle star(echo), v_1\rangle$$

$$undef\text{:} \quad \langle v_0, \lambda\rangle \ \rightarrow\ undef$$

$$def\text{:} \quad \langle v_0, \lambda\rangle \ \rightarrow\ def$$

Figure 4.3: Grammar for a toy programming language

Here is the definition of operator *stmt*.

$$\rho(stmt(e)) \quad = \quad \{\ \langle v_0, \lambda\rangle \ \rightarrow\ \langle e:def, v_1\rangle\ \text{“}\mathbf{=}\text{”}\ \langle e:def, v_2\rangle\ \text{“}\mathbf{;}\text{”}\ \}$$

Here, the identifiers are generated by the predicate for declared identifiers.

Figure 4.3 lists the definitions of all the nonterminal operators from this example. Note that the start symbol, *prog*, is a predicate that generates exactly the set of valid programs; in this case, programs have no semantics to speak of, since they produce no output and have no side effects. □

4.4 Comments

A number of high-level design issues could not be effectively discussed in the early sections of this chapter, because they cannot be properly appreciated without the

complete formal machinery of the RAG model as background. These issues are now discussed.

In a CFG, meta-syntax is represented by the nonterminals, and syntax is represented by the terminals. The purpose of the grammar rules —and therefore of the derivation step relation— is to express the relationship between meta-syntax and syntax.

In an attribute grammar (§2.2), terminals and nonterminals still satisfy the same roles; but in addition there are attributes, whose role is essentially semantic. Each grammar rule must express a multiplicity of relationships between meta-syntax, syntax, and various aspects of semantics. Although these relationships individually are localized in the parse tree (§2.2.5), this does not mean that they can be fully *understood* locally. Understanding a single attribute may require study of a large number of relationships distributed across a large number of grammar rules. ([Uhl 82] is a case in point.)

This difficulty is, in part, a consequence of the distributed nature of attribute grammar semantics. There is in general no upper bound on the number of attributes associated with each parse tree node; and the number of possible paths that attribute evaluation may take through the parse tree grows exponentially with the number of attributes.

Under the RAG model, each parse tree node has a single semantic value. Of course, it is *technically* possible to place the same restriction on attribute grammars. As shown by Example 2.4, one could arbitrarily require of an attribute grammar that every symbol have just one synthesized attribute and no inherited attributes. However, this would be at best a lopsided arrangement in terms of computational power. Arbitrarily powerful semantic functions would be needed, and these would overwhelm the merely context-free meta-syntactic capacity of the CFG nonterminals.

The RAG model suffers no such power imbalance. Meta-syntax and semantics are both Turing-powerful —in fact, they both derive their power from the way that they interact with each other, synthesized semantics depending on inherited meta-syntax and vice versa.

Moreover, the singularity of RAG semantic values is *not* arbitrary. As demonstrated above by Example 4.4, additional information is readily packaged within a single semantic or meta-syntactic value. Therefore, to provide an additional facility for the same purpose (i.e., arbitrary numbers of attributes) would directly violate the principle of orthogonality in the design (§4.1.2). This observation was foreshadowed by the discussion of nonterminals and language attributes in §4.1.3.

Another way of saying much the same thing is that the essential purpose of unbound rules, and therefore of the RAG derivation step relation itself, is to express the ternary relationship among meta-syntax, syntax, and semantics. These three elements are fundamental to the model; any additional attribute values would be logically superfluous, and would therefore distract from the clarity of description of

the three fundamentals (as well as exponentially increasing the potential complexity of relationships).

Orthogonality also enters into another formal constraint on RAGs. Recall from §2.2 that Knuth provided semantic equations only for synthesized attributes on the left side of a rule, and inherited attributes on the right. The requirement was lifted in EAGs, with the result that ostensibly synthesized attributes could actually be used to restrict the syntactic expansion; this result was illustrated by Example 2.6.

However, the definition of *unbound rule* in §4.2.2 reinstates Knuth's restriction. This is a consequence specifically of the imposition of orthogonality on meta-syntax in §4.1.3. The design principle of that subsection expressly forbids the very kind of syntactic manipulation that Example 2.6 exploits; there must be no way of using the semantic component of an unbound pair (on the right side of an unbound rule) to restrict the syntactic expansion.

The concepts of inheritance and synthesis have been referred to throughout the chapter, and even in the preceding paragraphs of this section. The issues involved warrant some discussion.

It was observed in §3.3.3 that the language attribute in a Christiansen grammar —and thus, by analogy, the left-hand component of an unbound pair in a RAG— is *conceptually* an inherited attribute. Similarly, the right-hand component of a RAG unbound pair is *conceptually* synthesized; the latter is especially clear from the query evaluation derivation (developed in §4.2.4),

$$a : b \quad \stackrel{+}{\Rightarrow} \quad a : \overline{\langle a, c \rangle} \quad \Rightarrow \quad c$$

in which the computation of the semantic value is carried out by, in effect, constructing a parse tree *from the bottom up*.

Nevertheless, the RAG model is not —necessarily— imperative in nature. It merely enforces a normal form in which the distinction between (conceptually inherited) meta-syntax and (conceptually synthesized) semantics is clearly delineated.

On the other hand, the discussion of this issue in §2.2.5 suggested that a sufficiently trivial algorithm can actually be an aid to understanding. Christiansen grammars were perceived to be above the necessary threshold of simplicity. RAGs may actually stay below the threshold; at least, the restriction to two "attributes" should greatly alleviate the problem.

The related issue of *non-circularity* will be addressed in Sections 6.6 and 7.1.3.

A few words are in order regarding the concept of a "recursive" grammar. It was remarked offhandedly in §4.1.4 that the recursive nature of RAGs is a consequence of the way the RAG derivation relation is used to evaluate expressions; but really, three different kinds of grammar recursion occur in this thesis.

A Chomsky grammar (Definition 1.3, §1.2) is said to be recursive iff there is some nonterminal that can occur in a string derived from itself; that is, G is recursive iff

there exists some $n \in N_G$ and $x, y \in Z_G^*$ such that $n \stackrel{+}{\underset{G}{\Rightarrow}} xny$. This kind of recursion is not only common but almost unavoidable, because a non-recursive Chomsky grammar can only generate a finite number of strings. Naturally, all of the complete examples of Chomsky grammars in this thesis are recursive.

Another form of recursion occurs in higher-order attribute grammars (§2.2.4). Recall that the distinguishing feature of that grammar model is its introduction of *nonterminal attributes*. Evaluation of a nonterminal attribute attaches a new branch to the parse tree, and this branch is subsequently available for attribute evaluation. Thus, in a sense, the parse-tree evaluation process calls itself recursively.

The RAG query operator provides yet another kind of recursion. Abstractly, a RAG parser would take a start symbol and a syntactic string as input, and either reject the input, or output a semantic value. In effect, each time a query operator occurs during parsing, the parse process calls itself recursively.

Since query evaluation in RAGs is analogous to attribute evaluation in an ordinary attribute grammar, query recursion is roughly analogous to the recursive attribute evaluation of a higher-order attribute grammar. However, query recursion is inherently simpler, because there is only *one* way to construct a parse tree —via the parser. A higher-order attribute grammar requires *two* distinct mechanisms for attribute evaluation: a parser to construct the initial parse tree, and separate mathematical tools for the construction of new branches.

Note also that the authors of higher-order attribute grammars develop a generalization of AG non-circularity for the express purpose of precluding infinite recursion [Vogt 89]. In constrast, the current thesis embraces unbounded query recursion as a means of achieving Turing-powerful computation; see especially the proof of Theorem 6.17 in §6.5. Of course, the current thesis does not address the question of parsing techniques, which is really the *only* major concern in [Vogt 89].

Chapter 5

Unrestricted RAGs

5.1 Introduction

This chapter defines the recursive adaptable grammar model in its fullest generality. Like Chomsky type 0 grammars, unrestricted RAGs are of interest primarily because of certain interesting subclasses, which will be developed in the next chapter.

The computational power of the RAG model will not be considered here, but deferred to §6.5.

Readers familiar with the work of the ADJ group on many-sorted algebra [ADJ 79] may skip most of §5.2, but should note the following differences:

1. The algebras used here are one-sorted (Definition 5.2).

2. The definition (5.5) of Σ-morphism does not require that the domain and codomain be Σ-algebras, provided that their signatures are supersets of Σ.

3. The definition (5.13) of extension allows the introduction of additional elements into the (one-sorted) carrier. Also, the additional elements are not required, in general, to conform to any Σ-equations that may have held on the original algebra.

4. The auxiliary concept of an *invariant* algebra (Definition 5.6) is introduced, for the sake of its use in guaranteeing the existence of initial extensions (Theorem 5.7).

5.2 One-sorted algebra

This section presents basic material on one-sorted algebra. In the literature, this subject is usually called "universal algebra". A generalization of universal algebra called *many-sorted algebra* [Wech 92] is of particular interest in computer science. Oddly, common usage seems to treat many-sorted algebra as a separate subject rather

than as a new branch of universal algebra. In the interests of clarity, the name "one-sorted algebra" is preferred here. (But note that usage in the bibliography follows that of the literature being referenced, so that the name "universal algebra" occurs several times therein.)

In the following development, emphasis is placed on techniques for the construction of new algebras, which is the only use to which the material is put in the current thesis. The formulation and presentation of the material were chosen with this use particularly in mind.

An additional motive for this section was that the author did not have access to a concise entry-level treatment of the subject in the literature. Much of the material here was adapted from [ADJ 79], which applies abstract algebra to the rigorous specification of program semantics; however, that paper is marred by ambiguous wording in some of the key definitions. A superior reference is [Wech 92]; however, it is difficult to extract basic one-sorted algebra from Wechler without also sifting through large quantities of incidental material. Other sources consulted include [Grät 68] and [Jaco 89], which address one-sorted algebra from a purely mathematical perspective.

5.2.1 Algebras

Definition 5.1 A *signature* Σ is a set of operators together with a function assigning to each operator σ in the set a nonnegative integer $ar(\sigma)$, called the *arity* of σ. $\Box$

The set of all operators of arity n in signature Σ is denoted Σ_n. The notation $\sigma \in \Sigma$ is shorthand meaning $\exists n \geq 0$ such that $\sigma \in \Sigma_n$. Similarly, for signatures Σ and Σ', $\Sigma \subseteq \Sigma'$ means that $\Sigma_n \subseteq \Sigma'_n$ for all $n \geq 0$; $\Phi = \Sigma \cup \Sigma'$ means $\Phi_n = \Sigma_n \cup \Sigma'_n$ for all $n \geq 0$; and so on.

Definition 5.2 Let Σ be a signature. Then a Σ-*algebra* A consists of a set, called the *carrier* of A, and, for each integer $n \geq 0$ and operator $\sigma \in \Sigma_n$, a function σ_A of arity n over the carrier of A. A "function of arity 0" is a constant, that is, an element of the carrier of A. $\Box$

The carrier of A is ambiguously denoted A. Thus, for all $n \geq 1$ and $\sigma \in \Sigma_n$, $\sigma_A : A^n \to A$. When $\sigma \in \Sigma_0$, $\sigma_A \in A$.

An expression composed from operators in signature Σ is called a Σ-term. For example, if $\Sigma_0 = \{0\}$, $\Sigma_1 = \{succ\}$, and $\Sigma_n = \emptyset$ for all $n \geq 2$, then the Σ-terms are: 0, $succ(0)$, $succ(succ(0))$, and so on. Different authors choose to construct Σ-terms in various ways. In [ADJ 79], Σ-terms are strings over the alphabet of operators in Σ augmented with special symbols for left paren, right paren, and comma. Alternatively, Σ-terms could be constructed as parse trees, with each node decorated by an operator of appropriate arity. The following definition leaves the construction unspecified.

Definition 5.3 Let Σ be a signature. Then the set T_Σ of Σ-*terms* is the smallest set satisfying the following two conditions.

1. If $\sigma \in \Sigma_0$, then $\sigma \in T_\Sigma$. That is, $\Sigma_0 \subseteq T_\Sigma$.
2. If $\sigma \in \Sigma_n$ for some $n \geq 1$, and $\tau_1, \cdots \tau_n$ are elements of T_Σ, then the construction $\sigma(\tau_1, \cdots \tau_n) \in T_\Sigma$. $\square$

If A is a Σ-algebra, one can imagine "evaluating" any Σ-term in A. More precisely, one may define a function $eval_A : T_\Sigma \to A$, as follows. If $\sigma \in \Sigma_0$ then $eval_A(\sigma) = \sigma_A$. If $\sigma \in \Sigma_n$ for some $n \geq 1$, and $\tau_1, \cdots \tau_n$ are elements of T_Σ, then $eval_A(\sigma(\tau_1, \cdots \tau_n)) = \sigma_A(eval_A(\tau_1), \cdots eval_A(\tau_n))$.

The set T_Σ can be converted into a Σ-algebra by adding operations in the obvious way:

Definition 5.4 Let Σ be a signature. The *term algebra* of Σ, denoted T_Σ, is constructed as follows. The carrier of algebra T_Σ is the set of all Σ-terms. If $\sigma \in \Sigma_0$ then $\sigma_{T_\Sigma} = \sigma$. If $\sigma \in \Sigma_n$ for some $n \geq 1$, and $\tau_1, \cdots \tau_n$ are Σ-terms, then $\sigma_{T_\Sigma}(\tau_1, \cdots \tau_n) = \sigma(\tau_1, \cdots \tau_n)$. $\square$

In the last equation in the above definition, note that the left side is a function being applied to an n-tuple of Σ-terms, while the right side is just a single Σ-term.

5.2.2 Initial algebras

The usual definition of a morphism requires that the domain and codomain have the same signature. This often leads to fumbling later, when one or both domains must be laboriously pared down by explicitly discarding operations until both algebras conform to the desired signature. (See for example [MacL 88, pages 129–131], [ADJ 79, page 99].) The definition below is more flexible.

Definition 5.5 Let A and B be algebras with signatures Σ_A and Σ_B respectively, and Σ a signature such that $\Sigma \subseteq (\Sigma_A \cap \Sigma_B)$. Then a function $h : A \to B$ is a Σ-*morphism* iff

1. For all operators $\sigma \in \Sigma_0$, $h(\sigma_A) = \sigma_B$.
2. For all integers $n \geq 1$, operators $\sigma \in \Sigma_n$, and tuples $\langle a_1, \cdots a_n \rangle \in A^n$, $h(\sigma_A(a_1, \cdots a_n)) = \sigma_B(h(a_1), \cdots h(a_n))$. $\square$

As usual, an injective (i.e., one-to-one) Σ-morphism is a Σ-monomorphism; surjective (i.e., onto), a Σ-epimorphism; and bijective (i.e., one-to-one *and* onto), a Σ-isomorphism.

For any Σ-algebra A, the function $eval_A$ described in §5.2.1 is a Σ-morphism from the term algebra T_Σ to A. In fact, it is the *only possible* Σ-morphism from T_Σ to

A, because the definition of Σ-morphism dictates exactly which element of A each Σ-term will map into.

If A and B are Σ-algebras, and $h : A \to B$ a Σ-morphism, then $h \circ eval_A = eval_B$. Here again, the definition of Σ-morphism dictates, for every a in the image of $eval_A$, exactly which element of B it must map into. If $eval_A$ is surjective (that is, its image is the entire set A), then h is unique, i.e., there is only one possible Σ-morphism from A to B.

Definition 5.6 An algebra A is *invariant* iff $eval_A$ is surjective. □

Lemma 5.1 Suppose A and B are algebras, A is invariant, and Σ is a signature. Then there is at most one Σ-morphism from A to B. □

Invariance provides a sufficient criterion for the *uniqueness* of a Σ-morphism from A to B, but does not guarantee that any such morphism exists. For example, suppose $\tau_1 \neq \tau_2$ are Σ-terms, and let $a_i = eval_A(\tau_i)$, $b_i = eval_B(\tau_i)$. Every Σ-morphism $h : A \to B$ must have $h(a_1) = b_1$ and $h(a_2) = b_2$. If $a_1 = a_2$ but $b_1 \neq b_2$, then there is no way of satisfying both equations, and there are no Σ-morphisms from A to B.

These observations lead naturally to the concept of an initial algebra. To formulate this concept, we will need the auxiliary concept of a *category* of algebras. (Category theory is a field of study in its own right, with a variety of applications to computer science; see e.g. [Pier 91]. Oddly enough, even in category theory, categories are still only an auxiliary concept. [MacL 71])

Definition 5.7 Let Σ be a signature. A *category of Σ-algebras* is a class of Σ-algebras together with all possible Σ-morphisms between them. □

The reader with mathematical scruples should note the word *class* in the above definition. A *class* is a collection of things, but only certain kinds of things are permitted to be elements of a class. In particular, not all classes can be elements of classes; a class that can be an element of a class is a *set*. This distinction is used to avoid paradoxes in set theory; for example, the class of all sets that do not contain themselves is not itself a set; hence it cannot belong to itself [MacL 71, page 23].

Definition 5.8 Let Σ be a signature, $\mathbf{C}$ a category of Σ-algebras, and I a Σ-algebra in $\mathbf{C}$. Then I is *initial* on $\mathbf{C}$ iff for every algebra $A \in \mathbf{C}$ there is exactly one Σ-morphism from I to A in $\mathbf{C}$. □

In abstract category theory, an object A in a category $\mathbf{C}$ is initial on $\mathbf{C}$ iff there is exactly one morphism from A to any object $B \in \mathbf{C}$, and final on $\mathbf{C}$ iff there is exactly one morphism *to* A *from* any object $B \in \mathbf{C}$. The initial objects on $\mathbf{C}$ form an equivalence class under isomorphism (and likewise the final objects on $\mathbf{C}$).

Theorem 5.2 Suppose Σ is a signature, and $\mathbf{C}$ a category of Σ-algebras. If I and J are both initial on $\mathbf{C}$, then they are Σ-isomorphic to each other. If I is initial on $\mathbf{C}$ and Σ-isomorphic to $J \in \mathbf{C}$, then J is initial on $\mathbf{C}$. □

We will generally speak of *the* initial algebra on a category, as if there were only one (which there is, up to isomorphism).

Theorem 5.3 For every signature Σ, the term algebra T_Σ is initial on the category of all Σ-algebras. □

Given any Σ-algebra A, the unique Σ-morphism from T_Σ to A is $eval_A$. Note that T_Σ is invariant (i.e., $eval_{T_\Sigma} : T_\Sigma \to T_\Sigma$ is surjective).

5.2.3 Specifications

Central to this subsection is the concept of a variable, which is essentially just a symbol that isn't in the signature of interest. A variable set over a signature Σ is an ordered set of symbols disjoint from Σ.

Definition 5.9 Let Σ be any signature, and Y a variable set over Σ. Then $\Sigma(Y)$ denotes the signature constructed by adjoining the variables of Y to the nullary operators of Σ. That is, $\Sigma(Y)_0 = \Sigma_0 \cup Y$, and $\Sigma(Y)_n = \Sigma_n$ for all $n \geq 1$. □

Elements of the term algebra $T_{\Sigma(Y)}$ are called *polynomials* over Σ. The *arity* of a polynomial π, denoted $ar(\pi)$, is the number of distinct variables that occur in π.

The notation $A(Y)$ will sometimes be used to denote the term algebra of polynomials in variables Y over the signature of algebra A. If Σ is the signature of A, then $A(Y) = T_{\Sigma(Y)}$. (This notation is convenient if the signature of A has not been stated explicitly.)

Also, suppose $\pi_0, \cdots \pi_n \in A(Y)$ are polynomials, and $ar(\pi_0) = n$. Let $X = \{x_1, \cdots x_n\}$ be the distinct variables in π_0, where X preserves the ordering of Y. Then the construct $\pi_0(\pi_1, \cdots \pi_n)$ denotes the polynomial obtained by substituting π_1 for every occurrence of x_1 in π_0, π_2 for every occurrence of x_2 in π_0, and so on.

Recall that, in any Σ-algebra A, each n-ary operator σ has an associated n-ary function $\sigma_A : A^n \to A$. In the same way, each n-ary polynomial π has an associated n-ary function $\pi_A : A^n \to A$.

Theorem 5.4 Let A be a Σ-algebra, Y a variable set over Σ, and $\theta : Y \to A$ a function. Then there is exactly one Σ-morphism $\overline{\theta} : T_{\Sigma(Y)} \to A$ such that, for all $y \in Y$, $\overline{\theta}(y) = \theta(y)$. □

What is really going on in this theorem is the quite ordinary activity of substituting values for variables and evaluating the resulting expression. Function θ assigns the values to the variables, and morphism $\overline{\theta}$ performs the substitution and evaluates.

Definition 5.10 Let A be a Σ-algebra, Y a variable set over Σ, and $\pi \in T_{\Sigma(Y)}$ a polynomial of arity n. Then π_A is an n-ary function over A, as follows.

Let $x_1, \cdots x_n$ be the distinct variables of π, in the order imposed by Y. Let $a_1, \cdots a_n$ be elements of A. Finally, let $\theta(x_i) = a_i$ for all $1 \leq i \leq n$. Then $\pi_A(a_1, \cdots a_n) = \overline{\theta}(\pi)$. $\Box$

Categories of algebras over a signature Σ are usually defined by listing a set of axioms that must hold on all algebras in the category. For example, a category that includes the algebra of real numbers might use such axioms as distributivity of multiplication over addition, the existence of a multiplicative inverse for every nonzero element, and so on. A particularly useful form of axiom is the *identity law*, which is an equation that must hold for all values on each algebra in the category.

Definition 5.11 For any signature Σ, a Σ-*equation* is an ordered pair of polynomials over Σ.

Further, suppose A is a Σ-algebra, $e = \langle \pi, \pi' \rangle$ a Σ-equation. Let Y be the set of all variables that occur in e. Then A *satisfies* e iff, for every possible assignment $\theta : Y \to A$, $\overline{\theta}(\pi) = \overline{\theta}(\pi')$. $\Box$

In normal usage, Σ-equations are usually written as $\pi = \pi'$ rather than $\langle \pi, \pi' \rangle$.

Definition 5.12 A *specification* is an ordered pair $\langle \Sigma, \mathcal{E} \rangle$, where Σ is a signature and $\mathcal{E}$ is a set of Σ-equations.

Let $\Omega = \langle \Sigma, \mathcal{E} \rangle$ be a specification. An Ω-algebra is any Σ-algebra that satisfies all of the Σ-equations in $\mathcal{E}$. $\Box$

For signatures $\Omega = \langle \Sigma, \mathcal{E} \rangle$ and $\Omega' = \langle \Sigma', \mathcal{E}' \rangle$, the notation $\Omega \subseteq \Omega'$ is shorthand meaning that $\Sigma \subseteq \Sigma'$ and $\mathcal{E} \subseteq \mathcal{E}'$.

Theorem 5.5 Let Ω be a specification. Then there exists an initial algebra on the category of all Ω-algebras. Further, the initial algebra is invariant. $\Box$

(Most authors, when stating this theorem, do not remark on the invariance of the initial algebra. It is mentioned here because it will be important for Theorem 5.7, in the next subsection.)

Example 5.1 Suppose Z is an alphabet. Consider the following specification. Let $\Sigma_0 = Z \cup \{\lambda\}$, $\Sigma_2 = \{\cdot\}$, and $\Sigma_n = \emptyset$ for all other n. Let $\mathcal{E}$ be the set containing the following three Σ-equations:

$$\begin{aligned} v_0 \cdot \lambda &= v_0 \\ \lambda \cdot v_0 &= v_0 \\ v_1 \cdot (v_2 \cdot v_3) &= (v_1 \cdot v_2) \cdot v_3 \end{aligned}$$

where the v_i are variables. The initial algebra on the category of all $\langle \Sigma, \mathcal{E} \rangle$-algebras consists of the set of all strings over alphabet Z together with the binary concatenation function. This algebra is called the *string algebra* over Z, and is denoted Z^*. $\Box$

5.2.4 Extensions

There are several mutually incompatible definitions of the concept of *extension of an algebra* in the literature; see for example [MacL 88], [ADJ 79], [Grät 68]. The definition below is a generalization of both [ADJ 79] and [Grät 68].

Definition 5.13 Let A be a Σ-algebra. An algebra B is an *extension* of A iff there is a Σ-monomorphism (that is, a one-to-one Σ-morphism) from A to B. □

Note that B does not have to be a Σ-algebra. However, since B is the codomain of a Σ-morphism, $\Sigma \subseteq \Sigma_B$.

If there is exactly one Σ-monomorphism from A to B, it is convenient to assume that the carrier of A is a subset of the carrier of B, so that the unique Σ-monomorphism maps each $a \in A$ into itself. One then says that A is a *subalgebra* of B, and writes $A \subseteq B$.

Lemma 5.6 Suppose A and B are algebras, and B is an extension of A. If A is invariant, then $A \subseteq B$. □

Definition 5.14 Let Σ, Σ_A, and Σ_B be signatures, A a Σ_A-algebra and B a Σ_B-algebra. Then B is a Σ-*extension* of A iff B is an extension of A and $\Sigma_B - \Sigma_A \subseteq \Sigma \subseteq \Sigma_B$.

If $\Omega = \langle \Sigma, \mathcal{E} \rangle$ is a specification, B is a Σ-extension of A, and B satisfies all the equations in $\mathcal{E}$, then B is an Ω-*extension* of A. □

The usefulness of these definitions depends on the following theorem.

Theorem 5.7 Suppose Ω is a specification, A an algebra, and $\mathbf{C}$ the category of all Ω-extensions of A. If A is invariant and $\mathbf{C}$ is nonempty, then there exists an initial algebra on $\mathbf{C}$. Further, the initial algebra is invariant. □

A proof of this theorem will be given later in the subsection.

The requirement that A be invariant is essential to the proof (because the proof uses Lemma 5.1). This is why Theorems 5.5 and 5.7 both note that their initial algebras are invariant. Any algebra whose existence is implied by either theorem is automatically suitable for subsequent extension via Theorem 5.7.

Accordingly, both theorems will be used in later sections, especially §5.3, to construct algebras from specifications. The string algebra Z^* of Example 5.1 (in §5.2.3) was constructed this way, via Theorem 5.5. However, a subtlety arises when Theorem 5.7 is used for the purpose.

Suppose $\Omega_A = \langle \Sigma_A, \mathcal{E}_A \rangle$ and $\Omega_B = \langle \Sigma_B, \mathcal{E}_B \rangle$ are specifications, A is the initial Ω_A-algebra, and B is the initial Ω_B-extension of A. Then A satisfies all the equations in $\mathcal{E}_A$, and B satisfies all the equations of $\mathcal{E}_B$. But there is no requirement for B to

satisfy the equations in $\mathcal{E}_A$. Those equations apply only to the construction of A; they may or may not continue to hold for values in $B - A$.

The remainder of the section sketches a proof of Theorem 5.7 (and, incidentally, of 5.5). It may help to clarify the preceding material; but it is not strictly necessary for later sections, and may be skipped without significant detriment thereto.

Definition 5.15 Let A be a Σ-algebra. A *congruence* on A is an equivalence relation $\equiv$ on the elements of A with the following property. Suppose $n \geq 1$ and $\sigma \in \Sigma_n$. If $a_i, b_i \in A$ and $a_i \equiv b_i$ for all $1 \leq i \leq n$, then $\sigma_A(a_1, \cdots a_n) \equiv \sigma_A(b_1, \cdots b_n)$. □

Theorem 5.8 Let R be a relation on Σ-algebra A. Then there is a least congruence $\equiv_R$ containing R, called the *congruence generated by R on A*. □

In other words, $\equiv_R$ is implied by R, and every other congruence implied by R is also implied by $\equiv_R$. The proof (not fully elaborated here) defines $\equiv_R$ to be the intersection of all congruences on A containing R, and verifies that $\equiv_R$ so defined is a congruence on A.

Definition 5.16 Let A be a Σ-algebra, $\equiv$ a congruence on A. The *quotient algebra* $A/\equiv$ is a Σ-algebra defined as follows. The elements of $A/\equiv$ are the equivalence classes of $\equiv$ on A. Suppose $n \geq 0$, $\sigma \in \Sigma_n$, and $a_1, \cdots a_n$ are elements of A. Then $\sigma_{A/\equiv}(\overline{a_1}, \cdots \overline{a_n}) = \overline{\sigma_A(a_1, \cdots a_n)}$. □

Here, $\overline{a}$ denotes the equivalence class of a. $\sigma_{A/\equiv}$ maps the equivalence classes of the a_i into the equivalence class of $\sigma_A(a_1, \cdots a_n)$. This definition of $\sigma_{A/\equiv}$ works only because $\equiv$ is a congruence. By the definition of congruence, σ_A will map all equivalent choices of the a_i into the same equivalence class.

Now, consider a Σ-equation $e = \langle \pi, \pi' \rangle$ with variables Y. e defines the following relation on T_Σ. Let τ and τ' be any Σ-terms. Then $\tau R_e \tau'$ iff there is some assignment $\theta : Y \to T_\Sigma$ such that $\overline{\theta}(\pi) = \tau$ and $\overline{\theta}(\pi') = \tau'$.

By Theorem 5.8, there is a least congruence implied by this relation. More generally, each specification $\Omega = \langle \Sigma, \mathcal{E} \rangle$ generates a congruence $\equiv_\Omega$ via the union of the relations defined by $\mathcal{E}$. Since $\equiv_\Omega$ is a congruence on T_Σ, there is a quotient algebra, denoted $T_\Omega = T_\Sigma/\equiv_\Omega$. From the construction in Definition 5.16, it follows trivially that T_Ω is invariant.

Theorem 5.9 For every specification Ω, the quotient algebra T_Ω is initial on the category of all Ω-algebras. Also, T_Ω is invariant. □

A complete proof of this theorem takes about a page and a half; see [ADJ 79]. The essence of the proof is that any Σ-morphism induces a congruence on its domain, $a \equiv b$ iff $h(a) = h(b)$. In particular, each Σ-algebra A induces a *unique* congruence $\equiv_A$ on

T_Σ; and this congruence contains $\equiv_\Omega$ iff A is an Ω-algebra. The unique morphism from T_Ω to A is essentially just the map of each equivalence class of $\equiv_\Omega$ to the equivalence class of $\equiv_A$ that contains it.

Theorem 5.5 is a weaker form of the same result, so it follows immediately from the above. The proof of Theorem 5.7 requires further explanation.

Proof. (Theorem 5.7) Suppose Σ_A is a signature, A an invariant Σ_A-algebra, and $\Omega_B = \langle \Sigma_B, \mathcal{E}_B \rangle$ a specification. The proof will construct a nonempty category that includes all Ω_B-extensions of A, and has an invariant initial algebra. Then it will show that the initial algebra is an Ω_B-extension of A unless there are no such extensions.

Construct a new specification $\Omega = \langle \Sigma, \mathcal{E} \rangle$ as follows.

$$\begin{aligned} \Sigma &= \Sigma_A \cup \Sigma_B \\ \mathcal{E} &= \mathcal{E}_B \cup \{\langle \tau_1, \tau_2 \rangle \mid \tau_1, \tau_2 \in T_{\Sigma_A} \text{ and } eval_A(\tau_1) = eval_A(\tau_2)\} \end{aligned}$$

Every Ω_B-extension of A is a Σ-algebra. Moreover, suppose B is a Σ-algebra that fails to satisfy all the equations in $\mathcal{E}$. Either B fails to satisfy some $e \in \mathcal{E}_B$, or B fails to satisfy some $e \in \mathcal{E} - \mathcal{E}_A$. In either case, B cannot be an Ω_B-extension of A. Hence, every Ω_B-extension of A is an Ω-algebra.

By Theorem 5.5, T_Ω is invariant and initial on the category of all Ω-algebras, which contains the category of all Ω_B-extensions of A. So if T_Ω belongs to the latter category, it is initial thereon.

On the other hand, suppose T_Ω is not an Ω_B-extension of A. The equations in $\mathcal{E}$ guarantee the existence of a Σ_A-morphism from A to T_Ω. By Lemma 5.1, there is only one such morphism. Call this unique Σ_A-morphism $h : A \to T_\Omega$. If h were injective (one-to-one), then because of the way Ω was constructed, T_Ω would be an Ω_B-extension of A. Therefore, by supposition, h is not injective.

Consider an arbitrary Ω-algebra B. Since T_Ω is the initial Ω-algebra, there is a unique Σ-morphism $g_B : T_\Omega \to B$. Therefore, there is a Σ_A-morphism $g_B \circ h : A \to B$, which is not injective because h is not injective. And by Lemma 5.1, there cannot be any other Σ_A-morphisms from A to B. So B is not an extension of A. So there aren't *any* Ω_B-extensions of A, which completes the proof. □

5.3 RAG definitions

The technical machinery of a recursive adaptable grammar is founded on four algebras, each a superset of the preceding. These algebras are defined below. The

following standard operators will be used.

operator	full name	arity	notation
qry	query operator	binary	$x : y$
pair	pairing operator	binary	$\langle x, y \rangle$
inv	inverse operator	unary	$\overline{x}$

To each of these standard operators σ is associated a signature Σ_σ containing that operator and no other. Thus $qry \in \Sigma_{qry}$, and so on.

Definition 5.17 A *vocabulary* is a pair $V = \langle \Omega_A, \Omega_T \rangle$, where $\Omega_A \supseteq \Omega_T$ are specifications, the initial Ω_A-algebra is an extension of the initial Ω_T-algebra, and neither specification involves the query, pairing, or inverse operator.

The initial Ω_T-algebra is called the *terminal algebra* over V, and denoted T_V. Similarly, the initial Ω_A-algebra is called the *answer algebra* over V, and denoted A_V. Elements of T_V are called *terminals* over V, and elements of A_V, *answers* over V. Elements of the difference set $A_V - T_V$ are called *nonterminals* over V. □

Note that, by Lemma 5.6, $T_V \subseteq A_V$.

The query algebra introduces the query operator.

Definition 5.18 The *query algebra* over a vocabulary V is the initial Σ_{qry}-extension of A_V. The query algebra over V is denoted Q_V. The elements of Q_V are called *queries* over V. □

Because Σ_{qry} places no additional constraints on A_V, the existence of a query algebra for every vocabulary is guaranteed by Theorem 5.7. As noted in §5.2.4, algebras implied under that theorem do not necessarily share the identity laws of their parents. In this case, none of the equations in the specification of the answer algebra are applicable to terms that involve the query operator. This is a deliberate feature of the model; it guarantees that a query invocation cannot be reduced to an answer except via derivation.

Again by Lemma 5.6, $A_V \subseteq Q_V$.

The configuration algebra introduces the remaining two operations, pairing and inverse.

Definition 5.19 The *configuration algebra* over a vocabulary V is an extension of the query algebra over V, as follows. Let $\Omega = \langle \Sigma, \mathcal{E} \rangle$ and $\Omega' = \langle \Sigma', \mathcal{E}' \rangle$ be specifications, where

$$\begin{aligned} \Sigma &= \Sigma_{inv} \\ \mathcal{E} &= \{v = \overline{v}\} \end{aligned}$$

$$\begin{aligned} \Sigma' &= \Sigma_{inv} \cup \Sigma_{pair} \cup \Sigma_{qry} \\ \mathcal{E}' &= \{v = \overline{\overline{v}}\} \end{aligned}$$

for variable v. Let A'_V be the initial Ω-extension of A_V. Then the configuration algebra over V is the initial Ω'-extension of A'_V. The configuration algebra over V is denoted C_V, and its elements are called *configurations* over V. $\Box$

Neither signature places any additional constraints on A_V, hence there exists a configuration algebra for every vocabulary. The signature of C_V contains that of Q_V, and places no additional constraints on query terms; hence C_V is also an extension of Q_V. By Lemma 5.6, $Q_V \subseteq C_V$.

The two-step construction of C_V (like the one-step of Q_V before it) takes advantage of the independence of extensions from the Σ-equations of their parents. In the intermediate algebra A'_V, unary inverse is just an identity function. But because C_V is defined as an extension of A'_V, the equation $v = \overline{v}$ does not apply to elements of $C_V - A_V$. Thus the desired result, that $v = \overline{v}$ only when $v \in A_V$.

The following theorem summarizes observations made (and proven) in the preceding text.

Theorem 5.10 If V is a vocabulary, then $T_V \subseteq A_V \subseteq Q_V \subseteq C_V$. $\Box$

The hierarchy of algebras over a vocabulary is used, in the following definitions, to develop recursive adaptable grammars.

Definition 5.20 An *unbound rule* over a vocabulary V is a structure of the form

$$\langle v_0, e_0 \rangle \quad \rightarrow \quad \pi(\langle e_1, v_1 \rangle, \cdots \langle e_n, v_n \rangle)$$

where $n \geq 0$, π is a polynomial of arity n over A_V, the v_k are distinct variables, and the e_k are polynomials in $Q_V(\{v_0 \cdots v_n\})$.

The domain of all unbound rules over V is denoted $\mathcal{R}_V$. $\Box$

It is an important feature of the above definition that π is a polynomial over the *answer* algebra, even though its arguments are not answers. Since $A_V \subseteq C_V$, every polynomial over A_V is also a polynomial over C_V. Thus, π is just a polynomial over C_V that doesn't use any operators outside A_V.

On the other hand, the expression $\pi(\langle e_1, v_1 \rangle, \cdots \langle e_n, v_n \rangle)$ is a polynomial over C_V, but not in general over A_V because, for $n \neq 0$, it involves the pairing operator.

Note that the internal structure of an unbound rule uniquely determines which of the symbols in it are variables, and the ordering of those variables in the variable set. The specific choice of variables is unimportant, and usually, no distinction is made between an unbound rule and its equivalence class under change of variables. (A notable exception occurs in §6.3.2, Definition 6.10, where change of variables is explicitly taken into account.)

Definition 5.21 A *bound rule* over a vocabulary V is a structure of the form

$$\langle a_0, q_0 \rangle \quad \rightarrow \quad \pi(\langle q_1, a_1 \rangle, \cdots \langle q_n, a_n \rangle)$$

where $n \geq 0$, π is a polynomial of arity n over A_V, the a_k are answers over V, and the q_k are queries over V.

For any unbound rule r over V, $\beta_V(r)$ denotes the set of all bound rules obtained from r as follows.

1. Select an assignment of values in A_V to variables in r.
2. Evaluate both sides of r. $\square$

Both sides of an unbound rule are polynomials over the configuration algebra, while those of a bound rule are *elements* of the configuration algebra. Suppose r is an unbound rule with n distinct variables. Each n-tuple of answers uniquely determines a bound rule, which is essentially an ordered pair over C_V. Thus, r effectively maps n-tuples over A_V into pairs over C_V; that is, $r : A_V^n \to C_V \times C_V$. Function β_V then maps each unbound rule r into its image $r(A_V^n) \subset (C_V \times C_V)$.

Definition 5.22 An *unrestricted recursive adaptable grammar* is a four-tuple $G = \langle \Omega_A, \Omega_T, \rho, s \rangle$, where $V = \langle \Omega_A, \Omega_T \rangle$ is a vocabulary, $\rho : A_V \to \mathcal{P}(\mathcal{R}_V)$, $s \in A_V$, and there exists some element $z \in T_V$ such that

$$\forall t \in T_V, \qquad \rho(t) \;=\; \{\langle v_0, z \rangle \to t\}$$

ρ is called the *rule function* of G, and s the *start symbol* of G.

Subscript G may be used in place of subscript V anywhere that subscript would have occurred; thus answer algebra A_G, function β_G, and so on. Also, the rule function of an arbitrary grammar G is denoted ρ_G, and the start symbol, s_G. $\square$

For each answer $a \in A_G$, $\beta_G(a)$ denotes the following set of bound rules.

$$\beta_G(a) \;=\; \{\langle a, q \rangle \to c \mid (\langle a, q \rangle \to c) \in \beta_G(r) \text{ for some } r \in \rho_G(a)\}$$

Note that this definition requires the left-hand variable on the *lhs* of each rule to be bound to a.

The union of $\beta_G(a)$ for all answers $a \in A_G$ is denoted $\beta(G)$. That is,

$$\beta(G) \;=\; \bigcup_{a \in A_G} \beta_G(a)$$

Definition 5.23 The *derivation step relation* for a recursive adaptable grammar G is the minimal binary relation $\underset{G}{\Rightarrow}$ over C_G satisfying the following axioms. Throughout the axioms, a, q, and c, with or without subscripts and primes, are assumed to be universally quantified over A_G, Q_G, and C_G, respectively.

Axiom 5.23a If $c_1 \underset{G}{\Rightarrow} c_2$ then $\overline{c_2} \underset{G}{\Rightarrow} \overline{c_1}$. $\diamond$

Axiom 5.23b If $(\langle a, q \rangle \to c) \in \beta(G)$ then $\langle a, \overline{q} \rangle \underset{G}{\Rightarrow} c$. $\diamond$

Axiom 5.23c Suppose $\sigma \neq inv$ is an operator of arity $n \geq 1$ in C_G. Suppose $c \underset{G}{\Rrightarrow} c'$. Let $c_1, \cdots c_n$ be configurations, with $c_k = c$ for some $1 \leq k \leq n$. Let

$$c'_j \;=\; \begin{cases} c' & \text{if } j = k \\ c_j & \text{otherwise} \end{cases}$$

Then $\sigma(c_1, \cdots c_n) \underset{G}{\Rrightarrow} \sigma(c'_1, \cdots c'_n)$. $\diamond$

Axiom 5.23d $a_1 : \overline{\langle a_1, a_2 \rangle} \underset{G}{\Rrightarrow} a_2$. $\diamond$

The *derivation relation* for a recursive adaptable grammar G is the reflexive transitive closure of $\underset{G}{\Rrightarrow}$, denoted $\overset{*}{\underset{G}{\Rrightarrow}}$. The transitive closure of $\underset{G}{\Rrightarrow}$ is denoted $\overset{+}{\underset{G}{\Rrightarrow}}$.

A *derivation* over G is a sequence $c_0, \cdots c_n$ of configurations over G, where $n \geq 0$, and $c_{k-1} \underset{G}{\Rrightarrow} c_k$ for all $1 \leq k \leq n$. Derivations are normally written as chains of derivation step relations; thus,

$$c_0 \underset{G}{\Rrightarrow} c_1 \underset{G}{\Rrightarrow} \cdots \underset{G}{\Rrightarrow} c_n$$

n is called the *length* of the derivation. A derivation of length zero is *trivial*. $\square$

Axiom 5.23a defines the significance of the inverse operator.

Axioms 5.23b and 5.23c correspond closely to the derivation relation of Chomsky grammars. For example, if σ were the binary concatenation operator, with the usual properties, then Axiom 5.23c would lead to the familiar proposition

$$x \underset{G}{\Rrightarrow} y \quad \text{implies} \quad pxq \underset{G}{\Rrightarrow} pyq$$

A subtlety of Axiom 5.23b is that the query on the *lhs* of the rule is inverted in the derivation relation. Starting from a pair of answers $\langle a, b \rangle$, there must be a derivation from q to b, so that $b \overset{*}{\underset{G}{\Rrightarrow}} \overline{q}$ and thus $\langle a, b \rangle \overset{*}{\underset{G}{\Rrightarrow}} \langle a, \overline{q} \rangle \underset{G}{\Rrightarrow} c$.

Axiom 5.23d defines the significance of the query operator. For a rigorous analysis, see Theorem 5.12 and its proof.

Definition 5.24 Given a recursive adaptable grammar G and answer $a \in A_G$, the *language accepted* by a in G is $L_G(a) = \{t \in T_G \mid a : t \overset{*}{\underset{G}{\Rrightarrow}} b \text{ for some } b \in A_G\}$. The language accepted by G is $L(G) = L_G(s_G)$. $\square$

That is, the language accepted by an answer $a \in A_G$ is the set of all terminals that a maps into semantic values, and the language accepted by G as a whole is the language accepted by the start symbol of G.

Definition 5.25 A function $f : X \to A_G$ with domain $X \subseteq A_G$ is *computed* by recursive adaptable grammar G iff

$$\forall x \in X,\ a \in A_G, \qquad s_G : x \overset{*}{\underset{G}{\Rrightarrow}} a \quad \text{iff} \quad f(x) = a$$

$\square$

5.4 RAG theorems

The two results in this section are properties of the derivation relations of all unrestricted recursive adaptable grammars. Theorem 5.11 shows that answers cannot be derived from other answers; Theorem 5.12, that the query operator has the expected behavior.

Theorem 5.11 If G is a RAG, $a \in A_G$, and $c \overset{+}{\underset{G}{\Rightarrow}} a$, then $c \notin A_G$. □

Proof. Suppose G is a RAG, v a variable over C_G. Let Π be the set of all polynomials $\pi \in C_G(\{v\})$ such that v occurs exactly once in π, and the occurrence of v in π is not within an argument of the query, pairing, or inverse operator. (Thus, expressions such as $v : a$ or $\langle a, b \cdot v \rangle$ would not be permitted.)

Consider the following sets.

$$Z_0 = \{\pi(\langle c_1, c_2 \rangle) \mid \pi \in \Pi \text{ and } c_k \in C_G\} \cup \{\pi(c_1 : c_2) \mid \pi \in \Pi \text{ and } c_k \in C_G\}$$

$$\forall n \in \mathbb{N}_+, \quad Z_n = \{\pi(\overline{c}) \mid \pi \in \Pi \text{ and } c \in Z_{n-1}\}$$

$$Z_+ = \bigcup_{k \in \mathbb{N}} Z_{2k}$$

$$Z_- = \bigcup_{k \in \mathbb{N}} Z_{2k+1}$$

Observe particularly that

$$A_G \cup Z_+ \cup Z_- = C_G.$$
$$A_G \cap Z_+ = A_G \cap Z_- = \emptyset.$$

$$\text{If } c \in Z_+ \text{ then } \overline{c} \in Z_-.$$
$$\text{If } c \in Z_- \text{ then } \overline{c} \in Z_+.$$

$$\text{If } \pi \in \Pi \text{ and } c \in Z_+ \text{ then } \pi(c) \in Z_+.$$
$$\text{If } \pi \in \Pi \text{ and } c \in Z_- \text{ then } \pi(c) \in Z_-.$$

$$\text{If } \pi \in \Pi \text{ and } c \in C_G \text{ and } \pi(c) \in A_G \text{ then } c \in A_G.$$
$$\text{If } \pi \in \Pi \text{ and } c \in C_G - Z_+ \text{ and } \pi(c) \in Z_+ \text{ then } \forall x \in C_G,\ \pi(x) \in Z_+.$$
$$\text{If } \pi \in \Pi \text{ and } c \in C_G - Z_- \text{ and } \pi(c) \in Z_- \text{ then } \forall x \in C_G,\ \pi(x) \in Z_-.$$

The following propositions will be proved simultaneously.

$$\text{If } c \in A_G \cup Z_+ \text{ and } c' \overset{*}{\underset{G}{\Rightarrow}} c \text{ then } c' \in Z_+.$$
$$\text{If } c \in A_G \cup Z_- \text{ and } c \overset{*}{\underset{G}{\Rightarrow}} c' \text{ then } c' \in Z_-.$$

Because $\underset{G}{\Rrightarrow}$ is the minimal relation satisfying its axioms, any derivation step $x \underset{G}{\Rrightarrow} y$ must follow from one of Axioms 5.23d, 5.23b by a finite (nonnegative) number of deductive steps via Axioms 5.23a, 5.23c. Proceed by induction on the number of deductive steps.

Base case.

If $x \underset{G}{\Rrightarrow} y$ is implied by Axiom 5.23d, then $x \in Z_0$ and $y \in A_G$. Moreover, $x \notin Z_-$.

If $x \underset{G}{\Rrightarrow} y$ is implied by Axiom 5.23b, then $x \in Z_0$ and $y \in Z_0 \cup A_G$. Moreover, $y \notin Z_-$.

Inductive step.

Suppose $x \underset{G}{\Rrightarrow} y$ is deduced via Axiom 5.23a. Then $\overline{y} \underset{G}{\Rrightarrow} \overline{x}$ can be proven with fewer deductive steps.

If $x \in A_G$, then $x = \overline{x}$. By inductive hypothesis, $\overline{y} \underset{G}{\Rrightarrow} x$ implies $\overline{y} \in Z_+$. Therefore, $y \in Z_-$.

If $y \in A_G$, then $y = \overline{y}$. By inductive hypothesis, $y \underset{G}{\Rrightarrow} \overline{x}$ implies $\overline{x} \in Z_-$. Therefore, $x \in Z_+$.

If $x \in Z_-$ then $\overline{x} \in Z_+$. By inductive hypothesis, $\overline{y} \underset{G}{\Rrightarrow} \overline{x}$ implies $\overline{y} \in Z_+$. Therefore, $y \in Z_-$.

If $y \in Z_+$ then $\overline{y} \in Z_-$. By inductive hypothesis, $\overline{y} \underset{G}{\Rrightarrow} \overline{x}$ implies $\overline{x} \in Z_-$. Therefore, $x \in Z_+$.

Suppose $x \underset{G}{\Rrightarrow} y$ is deduced via Axiom 5.23c. Then

$$\begin{aligned} x &= \sigma(x_1 \cdots x_n) \\ y &= \sigma(y_1 \cdots y_n) \end{aligned}$$

where $\sigma \neq inv$ is an operator of arity $n \geq 1$, and for some $1 \leq k \leq n$,

$$\begin{aligned} & x_k \underset{G}{\Rrightarrow} y_k \\ & \forall j \neq k, \quad x_j = y_j \end{aligned}$$

If $\sigma \in \{qry, pair\}$, then x and y are both in Z_0, and moreover, neither x nor y is in Z_-. Suppose $\sigma \notin \{qry, pair\}$. Let $\pi = \sigma(x_1, \cdots x_{k-1}, v, x_{k+1}, \cdots x_n)$. Then $\pi \in \Pi$, $x = \pi(x_k)$, $y = \pi(y_k)$, and $x_k \underset{G}{\Rrightarrow} y_k$ is proven with fewer deductive steps.

If $x \in A_G$ then $x_k \in A_G$. By inductive hypothesis, $y_k \in Z_-$. Therefore $y \in Z_-$.

If $y \in A_G$ then $y_k \in A_G$. By inductive hypothesis, $x_k \in Z_+$. Therefore $x \in Z_+$.

If $x_k \in Z_-$ then $x \in Z_-$. By inductive hypothesis, $y_k \in Z_-$. Therefore $y \in Z_-$.

If $y_k \in Z_+$ then $y \in Z_+$. By inductive hypothesis, $x_k \in Z_+$. Therefore $x \in Z_+$.

If $x \in Z_-$ but $x_k \notin Z_-$, then $\pi(c) \in Z_-$ for all $c \in C_G$. Therefore $y \in Z_-$.

If $y \in Z_+$ but $y_k \notin Z_+$, then $\pi(c) \in Z_+$ for all $c \in C_G$. Therefore $x \in Z_+$.

This completes the proof of the propositions. It follows immediately that, if $a \in A_G$ and $c \overset{+}{\underset{G}{\Rrightarrow}} a$, then $c \in Z_+$. But $A_G \cap Z_+ = \emptyset$, so $c \notin A_G$. $\square$

The sets Z_+ and Z_- in the above proof are not necessarily disjoint. Most configuration algebras will encompass peculiar configurations, such as $\langle x, y\rangle \cdot \overline{\langle x, y\rangle}$ (where "$\cdot$" denotes concatenation), that belong to both sets. But since configurations in Z_+ cannot be derived from answers, and configurations in Z_- cannot derive answers, configurations in $Z_+ \cap Z_-$ cannot do either. Configurations that belong simultaneously to both sets are said to be *improper*.

Theorem 5.12 If G is a RAG and $x, y, z \in A_G$, then

$$x : y \overset{*}{\underset{G}{\Rightarrow}} z \quad \text{iff} \quad \langle x, z\rangle \overset{*}{\underset{G}{\Rightarrow}} y$$

□

Proof. Suppose G is a RAG, and $x, y, z \in A_G$.

Suppose $\langle x, z\rangle \overset{*}{\underset{G}{\Rightarrow}} y$. By Axiom 5.23a, $\overline{y} \overset{*}{\underset{G}{\Rightarrow}} \overline{\langle x, z\rangle}$. But $y \in A_G$, so $y = \overline{y}$ and $y \overset{*}{\underset{G}{\Rightarrow}} \overline{\langle x, z\rangle}$. By Axiom 5.23c, $x : y \overset{*}{\underset{G}{\Rightarrow}} x : \overline{\langle x, z\rangle}$. By Axiom 5.23d, $x : \overline{\langle x, z\rangle} \overset{*}{\underset{G}{\Rightarrow}} z$. Therefore $x : y \overset{*}{\underset{G}{\Rightarrow}} z$.

On the other hand, suppose $x : y \overset{*}{\underset{G}{\Rightarrow}} z$. Of the four derivation axioms, only two of them can imply a derivation step with a left-hand side of the form $c : d$ — Axioms 5.23c and 5.23d. If such a step is implied by the former axiom, its right side must also have this form; if by the latter, its right side must be an answer.

Since z does not involve the query operator, at least one step in the derivation $x : y \overset{*}{\underset{G}{\Rightarrow}} z$ must be implied by Axiom 5.23d. Thus,

$$x : y \quad \overset{*}{\underset{G}{\Rightarrow}} \quad x' : \overline{\langle x', z'\rangle} \quad \underset{G}{\Rightarrow} \quad z' \quad \overset{*}{\underset{G}{\Rightarrow}} \quad z$$

where $x', z' \in A_G$. But because of Theorem 5.11, the only way one answer can be derived from another is trivially. Therefore $z' = z$, and

$$x : y \quad \overset{*}{\underset{G}{\Rightarrow}} \quad x' : \overline{\langle x', z\rangle} \quad \underset{G}{\Rightarrow} \quad z$$

All of the other steps of the derivation (besides, that is, the last) must be implied by Axiom 5.23c, and so we must have

$$\begin{array}{ll} x & \overset{*}{\underset{G}{\Rightarrow}} \quad x' \\ y & \overset{*}{\underset{G}{\Rightarrow}} \quad \overline{\langle x', z\rangle} \end{array}$$

Again by Theorem 5.11, $x = x'$. But then $y \overset{*}{\underset{G}{\Rightarrow}} \overline{\langle x, z\rangle}$, and by Axiom 5.23a, $\langle x, z\rangle \overset{*}{\underset{G}{\Rightarrow}} y$.
□

Chapter 6

Subclasses of RAGs

The unrestricted RAG model was consciously made far more general than is ordinarily necessary or desirable. This chapter identifies a number of RAG well-behavedness criteria. Subsequent to this chapter (and prior to Chapter 5), all RAGs are assumed to satisfy these criteria unless otherwise stated.

6.1 Stepwise decidability

One of the design objectives for the RAG model was an "elementary" derivation step relation (§4.1.4). Despite its fairly straightforward definition (5.23, in §5.3), the derivation step relation of an unrestricted RAG is not necessarily Turing-decidable.

Definition 6.1 A RAG G is *stepwise decidable* iff the derivation step relation $\overset{}{\underset{G}{\Rightarrow}}$ over C_G is Turing-decidable. □

That is, G is stepwise decidable iff the language $\{\langle c_1, c_2\rangle \mid c_1 \underset{G}{\Rightarrow} c_2\}$ is recursive.

This property is not in itself sufficient for practical use of the model. Some rather nasty computational anomalies can lurk in the underlying framework of a stepwise decidable RAG. Therefore a stronger condition is called for.

Definition 6.2 A RAG G is *strongly stepwise decidable* iff ρ_G is a partial recursive function and equivalence of polynomials over C_G is Turing-decidable. (Polynomials π and π' are equivalent over C_G, denoted $\pi \equiv_{C_G} \pi'$, iff $\pi_{C_G} = \pi'_{C_G}$.) □

There is nothing in the definition of unrestricted RAG (Definition 5.22 in §5.3) that requires any $\rho_G(a)$ to be a finite set. However, strong stepwise decidability does so require (because ρ_G must be partial recursive, and a Turing machine can't produce an infinitely long output in finite time).

It is conjectured that there exist stepwise decidable RAGs that fail both criteria for strong stepwise decidability, and stepwise decidable RAGs that satisfy either one critierion while failing the other. None of these claims will be further investigated

here. The existence of stepwise decidable RAGs that satisfy both criteria —i.e., are strongly stepwise decidable— is proven by example many times in the current thesis. Moreover,

Theorem 6.1 Every strongly stepwise decidable RAG is stepwise decidable. □

Proof. To paraphrase the theorem: Suppose G is a strongly stepwise decidable RAG, and terms over C_G are represented in some finite way (so that their structure can be analyzed in finite time). Then the following predicate D is Turing-decidable.

$$\forall \text{ terms } x, y \text{ over } C_G, \qquad D(x,y) \quad \text{iff} \quad eval_{C_G}(x) \underset{G}{\Rrightarrow} eval_{C_G}(y)$$

where, as described in §5.2.1, $eval_{C_G}$ is the evaluation function for C_G (i.e., it maps each term over C_G into its value in C_G). Assuming G and a representation of terms as above, it will be shown that D can be decided algorithmically in finite time.

A term τ over C_G denotes an answer iff it does not use the query or pairing operator. By assumption, this can be determined in finite time.

Therefore, it is possible to rewrite any term in finite time so as to eliminate all unnecessary occurrences of the inverse operator. (Double inversions are unnecessary, because $c = \overline{\overline{c}}$ for all $c \in C_G$; inversions of answers are unnecessary, because $a = \overline{a}$ for all $a \in A_G$.)

The *leading operator* of a term τ over C_G is the unique operator σ such that $\tau = \sigma(T)$ for some n-tuple T of terms, $n \geq 0$. By assumption, the leading operator of a term can be determined in finite time.

Suppose x and y are terms over C_G. Without loss of generality, assume that x and y contain no unnecessary inversions. $D(x,y)$ must be implied by one of Axioms 5.23a–5.23d. The four cases will be considered separately, as four predicates $D_1, \cdots D_4$.

Case 1: $D(x,y)$ implied by Axiom 5.23a.

If the leading operator of y is *inv*, let $y = \overline{y'}$. Otherwise, $D_1(x,y)$ is false.

If x denotes an answer, let $x = x'$. If the leading operator of x is *inv*, let $x = \overline{x'}$. Otherwise, $D_1(x,y)$ is false.

If x' and y' have been defined successfully, then $D_1(x,y) = D(y',x')$. This simplifies the problem, since y' is simpler than y, and x' is not less simple than x.

Case 2: $D(x,y)$ implied by Axiom 5.23b.

If the leading operator of x is not *pair*, then $D_2(x,y)$ is false. Otherwise, let $x = \langle a, q \rangle$. If a does not denote an answer, then $D_2(x,y)$ is false.

Let $y = \pi(\langle q_1, a_1 \rangle, \cdots \langle q_n, a_n \rangle)$, where π is a polynomial that does not use operator *pair*, and $n \geq 0$. (Every term over C_G has exactly one representation in this form, up to change of variables in π.) If any of the a_k are not answers, then $D_2(x,y)$ is false.

Compute $\rho_G(a)$, and consider in turn each unbound rule $r \in \rho_G(a)$. If r has more or less than n pairs on its *rhs*, then r cannot imply $D_2(x,y)$. Suppose r has exactly

n pairs on its *rhs*. Let $\{v_0, \cdots v_n\}$ be the variables of r, in order. Generate a bound rule r' by assigning $v_0 = a$, and $v_k = a_k$ for all $1 \leq k \leq n$. $D_2(x, y)$ is implied by r iff the *rhs* of r' is $\equiv_{C_G}$ to y and the *lhs* of r' is $\equiv_{C_G}$ to $\langle a, \overline{q} \rangle$.

$D_2(x, y)$ is true iff it is implied by some $r \in \rho_G(a)$.

Case 3: $D(x, y)$ implied by Axiom 5.23c.

For any given term z, there are a finite number (up to change of variables) of decompositions of z of the form $z = \pi(z')$, where π is a unary polynomial with only one occurrence of the variable, and $z \neq z'$. Call these *proper decompositions* of z. Given z, all proper decompositions of z can be enumerated in finite time.

Suppose $x = \pi_x(x')$ and $y = \pi_y(y')$ are proper decompositions. $D_3(x, y)$ is implied via these decompositions iff $D(x', y')$, and $\pi_x(x') \equiv_{C_G} \pi_y(x')$ or $\pi_x(y') \equiv_{C_G} \pi_y(y')$ or both. By definition of proper decomposition, x' and y' are simpler than x and y.

$D_3(x, y)$ is true iff it is implied by some choice of proper decompositions.

Case 4: $D(x, y)$ implied by Axiom 5.23d.

If x does not have the form $x = a_0 : \overline{\langle a_1, a_2 \rangle}$, then $D_4(x, y)$ is false. Otherwise, let the a_k be as shown. If the a_k do not all denote answers, then $D_4(x, y)$ is false. Otherwise, $D_4(x, y)$ iff $a_0 \equiv_{C_G} a_1$ and $a_2 \equiv_{C_G} y$.

$D(x, y)$ iff $D_1(x, y)$ or $D_2(x, y)$ or $D_3(x, y)$ or $D_4(x, y)$. Each of the four cases has been either decided or reduced to a finite number of simpler problems, in finite time. By induction, $D(x, y)$ can be decided in finite time. □

6.2 Answer encapsulation

6.2.1 Weak answer-encapsulation

The concept of encapsulation entails a distinction between public and private information. The public information for an answer domain A_G is captured by relation $\equiv_G$ in the following definition.

Definition 6.3 An *answer-equivalence* for a RAG G is an equivalence relation $\equiv$ on A_G such that, for all $x, x', y, z \in A_G$, if $x \equiv x'$ and $x : y \stackrel{*}{\underset{G}{\Rightarrow}} z$ then there exists $z' \in A_G$ such that $z \equiv z'$ and $x' : y \stackrel{*}{\underset{G}{\Rightarrow}} z'$.

$\equiv_G$ denotes the union of all answer-equivalences for G. That is, for all $x, y \in A_G$, $x \equiv_G y$ iff there exists an answer-equivalence $\equiv$ for G such that $x \equiv y$. □

Relation $\equiv_G$, the union of all answer-equivalences, is itself an answer-equivalence. To prove this, the following lemma is needed.

Lemma 6.2 If E and F are answer-equivalences for RAG G, then the transitive closure of their union is also an answer-equivalence for G. □

Proof. Suppose E and F are answer-equivalences for RAG G. Let $\equiv'$ be the union of relations E and F, and $\equiv$ the transitive closure of $\equiv'$.

Since every answer-equivalence is an equivalence relation, E and F are both reflexive. Therefore, $\equiv'$ and $\equiv$ are reflexive.

Suppose $x, x' \in A_G$, $x \neq x'$, and $x \equiv x'$. By definition of transitive closure, there must be a finite sequence of answers $x_0, \cdots x_n \in A_G$, $n \geq 1$, such that

$$x = x_0 \equiv' x_1 \equiv' \cdots \equiv' x_{n-1} \equiv' x_n = x'$$

Since E and F are both symmetric, $\equiv'$ is symmetric; hence

$$x' = x_n \equiv' x_{n-1} \equiv' \cdots \equiv' x_1 \equiv' x_0 = x$$

Thus $x' \equiv x$, and $\equiv$ is symmetric. By construction, $\equiv$ is transitive. As already noted, $\equiv$ is reflexive. Hence $\equiv$ is an equivalence relation.

Suppose further that $y, z \in A_G$ and $x : y \stackrel{*}{\underset{G}{\Rightarrow}} z$. Since E and F are answer-equivalences, there must exist a sequence of answers $z_0 \cdots z_n \in A_G$ such that

$$z = z_0 \equiv' z_1 \equiv' \cdots \equiv' z_{n-1} \equiv' z_n$$

$$\begin{array}{rcl} x_0 : y & \stackrel{*}{\underset{G}{\Rightarrow}} & z_0 \\ x_1 : y & \stackrel{*}{\underset{G}{\Rightarrow}} & z_1 \\ & \vdots & \\ x_{n-1} : y & \stackrel{*}{\underset{G}{\Rightarrow}} & z_{n-1} \\ x_n : y & \stackrel{*}{\underset{G}{\Rightarrow}} & z_n \end{array}$$

Thus, $z \equiv z_n$ and $x' : y \stackrel{*}{\underset{G}{\Rightarrow}} z_n$, as required. Therefore $\equiv$ is an answer-equivalence. □

Theorem 6.3 For every RAG G, $\equiv_G$ is an answer-equivalence. □

Proof. Suppose G is a RAG.

Trivially, equality on A_G is an answer-equivalence. Therefore $\equiv_G$ is reflexive.

Suppose $x, y \in A_G$ and $x \equiv_G y$. Then there exists an answer-equivalence $\equiv$ such that $x \equiv y$. Since $\equiv$ is an equivalence relation, $y \equiv x$. So $\equiv_G$ is symmetric.

Suppose $x, y, z \in A_G$, $x \equiv_G y$ and $x \equiv_G z$. Then there are answer-equivalences Y and Z such that xYy and xZz. Let $\equiv$ be the transitive closure of the union of Y and Z. Then $y \equiv z$, and by the lemma, $\equiv$ is an answer-equivalence. So $y \equiv_G z$, and $\equiv_G$ is transitive. Hence $\equiv_G$ is an equivalence relation.

Finally, suppose $x, x', y, z \in A_G$, $x \equiv_G x'$, and $x : y \stackrel{*}{\underset{G}{\Rightarrow}} z$. Then there exists an answer-equivalence $\equiv$ such that $x \equiv x'$, and by definition of answer-equivalence, $\exists z' \in A_G$ such that $z \equiv z'$ and $x' : y \stackrel{*}{\underset{G}{\Rightarrow}} z'$. But then $z \equiv_G z'$. Hence $\equiv_G$ is an answer-equivalence. □

Corollary 6.4 Suppose G is a RAG. For all terminals $x, y \in T_G$, $x \equiv_G y$ iff $x = y$. □

Proof. Suppose G is a RAG, $x, y \in T_G$, and $x \equiv_G y$. By the definition of unrestricted RAG, there exists some $z \in T_G$ such that

$$\begin{aligned} \rho_G(x) &= \{\langle v_0, z\rangle \to x\} \\ \rho_G(y) &= \{\langle v_0, z\rangle \to y\} \end{aligned}$$

Therefore, $x : x \overset{*}{\underset{G}{\Rightarrow}} z$. Since $x \equiv_G y$, there must exist some $z' \in A_G$ such that $z \equiv z'$ and $y : x \overset{*}{\underset{G}{\Rightarrow}} z'$. But the only derivation of this form is $y : y \overset{*}{\underset{G}{\Rightarrow}} z$. Hence $y = x$. □

Intuitively, $x \equiv_G y$ iff x and y generate the same syntax and assign it the same semantics; that is, iff x and y have the same "public interface". Encapsulation means that private information cannot be publicly accessed: that when x and y have the same public interface, they are freely interchangeable in any context. Formally,

Definition 6.4 Suppose G is a RAG, and σ an n-ary operator in A_G. σ is *G-safe* iff G has the following property: If $a_i, b_i \in A_G$ and $a_i \equiv_G b_i$ for all $1 \leq i \leq n$, then $\sigma(a_1, \cdots a_n) \equiv_G \sigma(b_1, \cdots b_n)$.

A RAG G is *weakly answer-encapsulated* iff all of the operators in A_G are G-safe. □

In other words, G is weakly answer-encapsulated iff $\equiv_G$ is a congruence.[1]

A virtue of weak answer-encapsulation is that it facilitates reliable (read: provable) regulation of information flow between different parts of a grammar —and, therefore, between different parts of a program. Some possible ramifications will be suggested in §7.2.3.

The following lemma will be used in the proof of Theorem 6.8 in §6.3.1.

Lemma 6.5 Suppose G is a RAG, and σ is an operator of arity zero in A_G. Then σ is G-safe. □

Proof. Suppose G and σ as in the lemma. Since σ has arity zero, there is only one choice of arguments for σ, namely the empty tuple, $\langle\rangle$. As already noted, equality is trivially an answer-equivalence; hence $\sigma \equiv_G \sigma$. Thus, the criterion for G-safeness in Definition 6.4 is satisfied. □

6.2.2 Strong answer-encapsulation

In order to make it easier to prove weak answer-encapsulation, the criterion of G-safeness will now be generalized, so that an operator can be proven safe over an entire class of RAGs. In the process, a stronger criterion for grammars, called *strong answer-encapsulation*, will be developed.

[1] Congruence on an algebra was defined in §5.2.4, Definition 5.15.

Definition 6.5 Suppose Φ is a signature, such that every Φ_n is countably infinite, and Φ does not include the query, pairing, or inverse operator. A RAG G is *consistent with* Φ iff the signature Σ_{A_G} of the answer algebra of G is contained in Φ, that is, $\Sigma_{A_G} \subseteq \Phi$. The class of all RAGs consistent with Φ is denoted $\mathcal{G}_\Phi$.

A *rule equation over* Φ is an ordered pair $e = \langle \sigma, \alpha \rangle$, where $\sigma \in \Phi_n$ for some $n \geq 0$, and α is a $2n$-ary function as follows. Let $\mathcal{A}_\Phi$ be the term algebra over Φ. Let $\mathcal{R}_\Phi$ be the class of all unbound rules over all $G \in \mathcal{G}_\Phi$. Then

$$\alpha : (\mathcal{A}_\Phi \times \mathcal{P}(\mathcal{R}_\Phi))^n \to \mathcal{P}(\mathcal{R}_\Phi)$$

e is said to *specify* σ.

Suppose $e = \langle \sigma, \alpha \rangle$ is a rule equation over Φ, and $G \in \mathcal{G}_\Phi$. Then G *satisfies* e iff σ is an n-ary operator in A_G and, for all terms $\tau_1, \cdots \tau_n$ over A_G,

$$\rho_G(\sigma(\tau_1, \cdots \tau_n)) \quad = \quad \alpha(\tau_1, \rho_G(eval_{A_G}(\tau_1)), \cdots \tau_n, \rho_G(eval_{A_G}(\tau_n)))$$

□

Rule equations are usually not written as ordered pairs. In keeping with the conventions described in §4.2.2 (Example 4.1), a rule equation $\langle \sigma, \alpha \rangle$ with $ar(\sigma) \geq 1$ is represented as an equation of the form

$$\rho(\sigma(a_1, \cdots a_n)) \quad = \quad \alpha(a_1, \rho(a_1), \cdots a_n, \rho(a_n))$$

where the arguments $a_1, \cdots a_n$ are understood to be universally quantified over A_G. For example (from §4.3, Example 4.3):

$$\rho(star(a)) \quad = \quad \left\{ \begin{array}{rcl} \langle v_0, \lambda \rangle & \to & \lambda \\ \langle v_0, v_1 v_2 \rangle & \to & \langle a, v_1 \rangle \langle star(a), v_2 \rangle \end{array} \right\}$$

In the common case that $ar(\sigma) = 0$ and $\alpha() = \{r_1, \cdots r_m\}$, one *may* use the shorthand notation

$$\begin{array}{ll} \sigma\text{:} & r_1 \\ & r_2 \\ & \vdots \\ & r_m \end{array}$$

as in (from §4.2.2, Example 4.1):

$$\begin{array}{lrcl} S\text{:} & \langle v_0, \lambda \rangle & \to & \lambda \\ & \langle v_0, \lambda \rangle & \to & a \langle v_0, v_1 \rangle b \end{array}$$

Definition 6.6 A *RAG framework* (or simply a *framework*) is an ordered pair $\mathcal{F} = \langle \Phi, \mathcal{E} \rangle$, where $\mathcal{E}$ is a set of rule equations over signature Φ, and every Φ_n contains a countably infinite number of operators not specified by any $e \in \mathcal{E}$.

$\mathcal{F}$ *specifies* an operator σ iff there exists some $e \in \mathcal{E}$ such that e specifies σ. A RAG G *satisfies* $\mathcal{F}$ iff G is consistent with Φ and satisfies all $e \in \mathcal{E}$. The class of all RAGs consistent with $\mathcal{F}$ is denoted $\mathcal{G}_{\mathcal{F}}$. $\mathcal{F}$ is *satisfiable* iff the class $\mathcal{G}_{\mathcal{F}}$ is nonempty. □

Note that a framework $\mathcal{F} = \langle \Phi, \mathcal{E} \rangle$ is satisfiable iff there are no two rule equations in $\mathcal{E}$ that specify the same operator.

Under most circumstances, the signature Φ is incidental. A typical rule equation can be defined without specifically naming more than a finite number of operators from Φ, as in the example above (which uses only binary concatenation, unary *star*, and constant λ). By convention, it is considered sufficient in ordinary usage to ensure that all operators are used consistently, i.e., always with the same arity; the existence of a suitable signature Φ is assumed, and operators from this hypothetical signature are invented on demand.

Definition 6.7 Suppose $\mathcal{F}$ is a framework. Then $\mathcal{F}$ is *safe* iff $\mathcal{F}$ is satisfiable and, for all $G \in \mathcal{G}_{\mathcal{F}}$, every operator specified by $\mathcal{F}$ is G-safe.

A rule equation e over signature Φ is *safe* iff the framework $\langle \Phi, \{e\} \rangle$ is safe.

A RAG G is *strongly answer-encapsulated* iff there exists a safe framework $\mathcal{F}$ such that $G \in \mathcal{G}_{\mathcal{F}}$, and every operator σ in A_G is specified by $\mathcal{F}$. □

The following theorem is a trivial consequence of the above definitions.

Theorem 6.6 Every strongly answer-encapsulated RAG is weakly answer-encapsulated. □

6.3 Proofs of answer-encapsulation

In this section, it is proved that all of the RAG frameworks presented in the thesis —excepting §6.3.3, below— are safe. The primary intent of the section is to provide insight into the nature of safe frameworks (although, of course, it is of no little interest to know that all of the RAGs in the thesis are strongly answer-encapsulated).

The majority of the rule equations in the thesis are covered by the general theorems in §6.3.1. However, no general characterization of safe frameworks is forthcoming. Theorem 6.10 of §6.3.1 is tedious to state, let alone prove, and would be far worse if generalized to handle additional issues such as circular rules or self-referencing operators.

Subsection 6.3.2 provides safeness proofs for all those frameworks presented in the thesis that do not fall under the aegis of the theorems in §6.3.1. Subsection 6.3.3 gives an example of an inherently unsafe rule equation.

6.3.1 General theorems

Three theorems are proven in this subsection. Theorems 6.7 and 6.8 are simple nearly to the point of triviality. In constrast, Theorem 6.10 is (as already remarked) rather tedious.

Theorem 6.7 If $\langle \Phi, \mathcal{E}_1 \rangle$ and $\langle \Phi, \mathcal{E}_2 \rangle$ are safe frameworks, and $\mathcal{F} = \langle \Phi, \mathcal{E}_1 \cup \mathcal{E}_2 \rangle$ is a satisfiable framework, then $\mathcal{F}$ is safe. □

Proof. Suppose $\mathcal{F}_1 = \langle \Phi, \mathcal{E}_1 \rangle$ and $\mathcal{F}_2 = \langle \Phi, \mathcal{E}_2 \rangle$ are safe frameworks, and $\mathcal{F} = \langle \Phi, \mathcal{E}_1 \cup \mathcal{E}_2 \rangle$ is a satisfiable framework. Further, suppose $G \in \mathcal{G}_{\mathcal{F}}$. Then by Definition 6.6, G must satisfy all of the rule equations in $\mathcal{E}_1 \cup \mathcal{E}_2$, hence $G \in \mathcal{G}_{\mathcal{F}_1}$ and $G \in \mathcal{G}_{\mathcal{F}_2}$. Since $\mathcal{F}_1$ and $\mathcal{F}_2$ are both safe, every operator specified in $\mathcal{E}_1 \cup \mathcal{E}_2$ must be G-safe. But then by Definition 6.7, $\mathcal{F}$ is safe. □

Theorem 6.8 Suppose $e = \langle \sigma, \alpha \rangle$ is a rule equation over Φ, and $\sigma \in \Phi_0$. Then $\mathcal{F} = \langle \Phi, \{e\} \rangle$ is a safe framework (that is, e is safe). □

Proof. By Definition 6.5, Φ must have a countably infinite number of operators of each arity; therefore, by Definition 6.6, $\mathcal{F}$ is a framework. No conflict can occur between rule equations in $\mathcal{F}$, because e is the *only* rule equation in $\mathcal{F}$; hence $\mathcal{F}$ is satisfiable. If $G \in \mathcal{G}_{\mathcal{F}}$ is a RAG that satisfies $\mathcal{F}$, then by Lemma 6.5, σ is G-safe; therefore, $\mathcal{F}$ is a safe framework. □

The third theorem involves the auxiliary concept of a safe rule form.

Definition 6.8 Suppose $\mathcal{F} = \langle \Phi, \mathcal{E} \rangle$ is an answer framework, and $\Sigma \subset \Phi$ is the signature made up of exactly those operators specified by $\mathcal{F}$. Then the set of *safe query forms over* $\mathcal{F}$, denoted $\mathcal{Q}_{\mathcal{F}}$, is the minimal set satisfying the following conditions. (Throughout, all variables are assumed to be disjoint from Φ.)

1. If π is a polynomial over Σ, then $\pi \in \mathcal{Q}_{\mathcal{F}}$.
2. If τ is a term over Φ, and $e \in \mathcal{Q}_{\mathcal{F}}$, then $(e : \tau) \in \mathcal{Q}_{\mathcal{F}}$.
3. If $n \geq 1$, $\sigma \in \Sigma_n$, and $e_1, \cdots e_n \in \mathcal{Q}_{\mathcal{F}}$, then $\sigma(e_1, \cdots e_n) \in \mathcal{Q}_{\mathcal{F}}$. □

This definition may also be usefully understood in terms of what it does *not* allow. A safe query form over $\mathcal{F}$ is a polynomial over the query algebra corresponding to Φ, such that (1) no variable occurs on the right side of a query operator, and (2) no operator in $\Phi - \Sigma$ occurs anywhere *except* on the right side of a query operator.

Lemma 6.9 Suppose $\mathcal{F}$ is an answer framework, $e \in \mathcal{Q}_{\mathcal{F}}$ a safe query form of arity n over $\mathcal{F}$, and $G \in \mathcal{G}_{\mathcal{F}}$ a RAG that uses all the operators in e. Further, suppose that for all $1 \leq i \leq n$, $a_i, b_i \in A_G$ and $a_i \equiv_G b_i$. For all $a' \in A_G$, if $e(a_1, \cdots a_n) \overset{*}{\underset{G'}{\Rightarrow}} a'$ then there exists some $b' \in A_G$ such that $e(b_1, \cdots b_n) \overset{*}{\underset{G'}{\Rightarrow}} b'$ and $b' \equiv_G a'$. □

Proof. Suppose $\mathcal{F}$ etc. as in the lemma. Let $\mathcal{F} = \langle \Phi, \mathcal{E} \rangle$, and let Σ be the signature made up of exactly those operators specified by $\mathcal{F}$. Proceed by induction on the structure of e.

Base case. $e \in \mathcal{Q}_\mathcal{F}$ is a polynomial over Σ. Let $a = e(a_1, \cdots a_n)$ and $b = e(b_1, \cdots b_n)$. Since all of the operators in Σ are G-safe, $a \equiv_G b$. The lemma is then satisfied by trivial derivations $a \stackrel{*}{\underset{G}{\Rightarrow}} a$ and $b \stackrel{*}{\underset{G}{\Rightarrow}} b$.

Inductive step. $e \in \mathcal{Q}_\mathcal{F}$ has the form $(e' : \tau)$ or $\sigma(e_1, \cdots e_m)$.

Suppose $e = (e' : \tau)$, where e' is a safe query form over $\mathcal{F}$, and τ is a term over Φ. Thus, $e(a_1, \cdots a_n) = e'(a_1, \cdots a_n) : \tau$. By supposition, $e'(a_1, \cdots a_n) : \tau \stackrel{*}{\underset{G}{\Rightarrow}} a'$. This derivation involves the elimination of a query operator; so, by the definition of the RAG derivation step relation (Definition 5.23 in §5.3), there must be some $a'' \in A_G$ such that

$$e'(a_1, \cdots a_n) : \tau \quad \stackrel{*}{\underset{G}{\Rightarrow}} \quad a'' : \tau \quad \stackrel{*}{\underset{G}{\Rightarrow}} \quad a'$$

and $e'(a_1, \cdots a_n) \stackrel{*}{\underset{G}{\Rightarrow}} a''$. By inductive hypothesis, the latter derivation implies that there must exist some $b'' \in A_G$ such that $e'(b_1, \cdots b_n) \stackrel{*}{\underset{G}{\Rightarrow}} b''$ and $a'' \equiv_G b''$. By the definition of answer-equivalence (Definition 6.3 in §6.2.1), $a'' : \tau \stackrel{*}{\underset{G}{\Rightarrow}} a'$ implies that there exists some $b' \in A_G$ such that $b'' : \tau \stackrel{*}{\underset{G}{\Rightarrow}} b'$. Thus,

$$e'(b_1, \cdots b_n) : \tau \quad \stackrel{*}{\underset{G}{\Rightarrow}} \quad b'' : \tau \quad \stackrel{*}{\underset{G}{\Rightarrow}} \quad b'$$

On the other hand, suppose $e = \sigma(e_1, \cdots e_m)$, where $m = ar(\sigma)$ and, for all $1 \leq j \leq m$, $e_j \in \mathcal{Q}_\mathcal{F}$. By supposition,

$$\sigma(e_1(a_1, \cdots a_n), \cdots e_m(a_1, \cdots a_n)) \quad \stackrel{*}{\underset{G}{\Rightarrow}} \quad a$$

By the definition of the RAG derivation step relation, there must be some $a'_1, \cdots a'_m \in A_G$ such that $a = \sigma(a'_1, \cdots a'_m)$ and, for all $1 \leq j \leq m$, $e_j(a_1, \cdots a_n) \stackrel{*}{\underset{G}{\Rightarrow}} a'_j$. By inductive hypothesis, these derivations imply that there must exist some $b'_1, \cdots b'_m \in A_G$ such that for all $1 \leq j \leq m$, $e_j(b_1, \cdots b_n) \stackrel{*}{\underset{G}{\Rightarrow}} b'_j$ and $a'_j \equiv_G b'_j$. Thus,

$$\sigma(e_1(b_1, \cdots b_n), \cdots e_m(b_1, \cdots b_n)) \quad \stackrel{*}{\underset{G}{\Rightarrow}} \quad \sigma(b'_1, \cdots b'_m)$$

Moreover, $\sigma \in \Sigma$, therefore σ is G-safe, and $\sigma(b'_1, \cdots b'_m) \equiv_G a$. □

Definition 6.9 Suppose $\mathcal{F} = \langle \Phi, \mathcal{E} \rangle$ is an answer framework, $\Sigma \subset \Phi$ is the signature made up of exactly those operators specified by $\mathcal{F}$, and X is a variable set over Φ. Further, suppose r is an unbound rule

$$\langle v_0, e_0 \rangle \quad \rightarrow \quad \pi(\langle e_1, v_1 \rangle, \cdots \langle e_n, v_n \rangle)$$

where $V = \{v_0, \cdots v_n\}$ is disjoint from $\Phi(X)$, π is a polynomial over Φ, and, for all $0 \leq k \leq n$, $e_k \in \mathcal{Q}_\mathcal{F}$ is a safe query form in variables $X \cup V - \{v_0\}$ over Σ. Let $\lhd_r$ be the minimal binary relation over $X \cup V$ such that

1. For all $1 \leq k \leq n$ and $u \in (X \cup V)$, if u occurs in e_k then $u \lhd_r v_k$.
2. For all $x, y, z \in X \cup V$, if $x \lhd_r y$ and $y \lhd_r z$ then $x \lhd_r z$.

If there do not exist any $v \in V$ and $x \in X$ such that $v \lhd_r v$ and $x \lhd_r v$, then r is a *safe rule form over $\mathcal{F}$ parameterized by X*.

The set of all safe rule forms over $\mathcal{F}$ parameterized by X is denoted $\mathcal{R}_{\mathcal{F},X}$. □

In other words, r is an unbound rule, using operators from $\Phi(X)$, such that the e_k are safe query forms over $\mathcal{F}$, and any v_k that is dependent on some $x \in X$ is not also dependent on itself.

Theorem 6.10 Suppose $\mathcal{F} = \langle \Phi, \mathcal{E} \rangle$ is a safe framework; $\Sigma \subset \Phi$ the signature made up of exactly those operators specified by $\mathcal{F}$; $\sigma \in \Phi - \Sigma$ an operator of arity n not specified by $\mathcal{F}$; $X = \{x_1, \cdots x_n\}$ a variable set over Φ; and $R \subseteq \mathcal{R}_{\mathcal{F},X}$ a set of safe rule forms over $\mathcal{F}$ parameterized by X. Let e be the following rule equation over Φ.

$$\rho(\sigma(x_1, \cdots x_n)) \quad = \quad R$$

Then $\mathcal{F}' = \langle \Phi, \mathcal{E} \cup \{e\} \rangle$ is a safe framework. □

Proof. Suppose $\mathcal{F}$ etc. as in the theorem. Since $\mathcal{F}$ is a framework, $\Phi - \Sigma$ must have a countably infinite number of operators of each arity; therefore, $\mathcal{F}'$ is a framework. Since σ is not specified by $\mathcal{F}$, e cannot conflict with any of the other rule equations in $\mathcal{E}$; so since $\mathcal{F}$ is satisfiable, $\mathcal{F}'$ must also be satisfiable.

Suppose $G \in \mathcal{G}_{\mathcal{F}'}$, and, for all $1 \leq i \leq n$, $a_i, b_i \in A_G$ and $a_i \equiv_G b_i$. Let $a = \sigma_{A_G}(a_1, \cdots a_n)$ and $b = \sigma_{A_G}(b_1, \cdots b_n)$. Further, suppose that $a : t \overset{*}{\underset{G}{\Rrightarrow}} a'$ for some $t, a' \in A_G$. It will be shown that there must exist some $b' \in A_G$ such that $a' \equiv_G b'$ and $b : t \overset{*}{\underset{G}{\Rrightarrow}} b'$.

Since $a : t \overset{*}{\underset{G}{\Rrightarrow}} a'$, by Theorem 5.12, $\langle a, a' \rangle \overset{*}{\underset{G}{\Rrightarrow}} t$. By the definition of the RAG derivation step relation, there must be some unbound rule $r \in \rho_G(a)$ and some bound rule $(\langle a, q \rangle \to c) \in \beta_G(r)$ such that $q \overset{*}{\underset{G}{\Rrightarrow}} a'$ and $c \overset{*}{\underset{G}{\Rrightarrow}} t$, thus

$$\langle a, a' \rangle \overset{*}{\underset{G}{\Rrightarrow}} \langle a, \overline{q} \rangle \underset{G}{\Rrightarrow} c \overset{*}{\underset{G}{\Rrightarrow}} t$$

Since G satisfies the rule equation $\rho(\sigma(x_1, \cdots x_n)) = R$, there must be a safe rule form

$$\langle v_0, e_0 \rangle \quad \to \quad \pi(\langle e_1, v_1 \rangle, \cdots \langle e_m, v_m \rangle)$$

in R such that substituting $x_i = a_i$ for all $1 \leq i \leq n$ throughout the rule form yields the unbound rule r. Moreover, there must exist answers $a'_1, \cdots a'_m \in A_G$ such that substituting $v_0 = a$ and $v_j = a'_j$ for all $1 \leq j \leq m$ throughout r yields the bound rule $\langle a, q \rangle \to c$.

By the definition of the RAG derivation step relation,

$$\begin{aligned} c &= \pi(\langle q_1, a'_1\rangle, \cdots \langle q_m, a'_m\rangle) \\ t &= \pi(t_1, \cdots t_m) \end{aligned}$$

where, for all $1 \leq j \leq m$, $\langle q_j, a'_j\rangle \stackrel{*}{\Rightarrow}_G t_j$, and thus, again by Theorem 5.12, $q_j : t_j \stackrel{*}{\Rightarrow}_G a'_j$.

Let r' be the unbound rule that results from substituting $x_i = b_i$ for all $1 \leq i \leq n$ throughout the same rule form as above. An assignment of values b'_j to variables v_j in r' will now be shown, such that all $b'_j \equiv_G a'_j$, and t is derivable from the right side of the resulting bound rule.

Let $b'_0 = b$. For all $1 \leq j \leq m$, if $v_j \triangleleft v_j$ then let $b'_j = a'_j$. There are no circular dependencies between the remaining variables. Choose one of these variables v_j, such that values have already been chosen for all variables on which v_j depends, that is, all $v \in V$ such that $v \triangleleft v_j$.

Let p_j be the value obtained from e_j by substituting for all $x \in X$ and all previously selected $v \in V$, as above. All of these previous substitutions used values $\equiv_G$ to the values that yielded q_j from e_j, and by supposition, e_j is a safe query form over $\mathcal{F}$. Therefore, by Lemma 6.9, there must exist some $b'_j \equiv_G a'_j$ such that $p_j : t_j \stackrel{*}{\Rightarrow}_G b'_j$.

Repeat the procedure until all b'_j have been selected. Finally, applying Lemma 6.9 to the value p_0 obtained from e_0, there must exist some $b' \equiv a'$ such that $p_0 \stackrel{*}{\Rightarrow}_G b'$. Putting all these results together,

$$\langle b, b'\rangle \stackrel{*}{\Rightarrow}_G \langle b, \overline{p_0}\rangle \Rightarrow_G \pi(\langle p_1, b'_1\rangle, \cdots \langle p_m, b'_m\rangle) \stackrel{*}{\Rightarrow}_G \pi(t_1, \cdots t_m) = t$$

Thus, $b : t \stackrel{*}{\Rightarrow}_G b'$. □

It was essential to the above proof that the definition of safe rule form prohibits the use of v_0 in the e_j. For suppose that v_0 occurred in some e_j. The values assigned to v_0 are a and b. But Lemma 6.9 cannot be applied unless all the corresponding arguments to e_j are known to be answer-equivalent; and the whole point of the proof was to *establish* that $a \equiv_G b$.

To put it another way: The essence of a safe rule form would not be changed by substituting $v_0 = \sigma(x_1, \cdots x_n)$ throughout the query forms e_j. But $\sigma \notin \Sigma$, thus σ is not known to be G-safe. Again, showing σ G-safe is the purpose of the proof.

6.3.2 Other safe frameworks

There are six rule equations presented in the thesis that are not covered by the theorems of the preceding subsection. One of these (rule equation 6.4, §6.3.3) is presented specifically as an example of an *un*safe rule equation. The others are:

- binary concatenation, whose equation (to be constructed below) does not use a rule form.

- unary *star*, from Example 4.3, §4.3, and unary *decl-list*, from Example 4.4, §4.3, each of which is self-referencing.
- binary *and* and unary *decl*, from Example 4.4, §4.3, each of which uses a variable on the right side of a query.

The binary concatenation operator

In the introduction to rule functions in §4.2.2, no attempt was made to provide a rule equation for the binary concatenation operator, even though rule equations were provided for several other canonical operators (Proposition 4.1). Instead, concatenation was described indirectly, by Propositions 4.2–4.3:

1. For all $a_k, b_k \in A_G$, if $\langle a_0, a_1 \rangle \stackrel{*}{\Rightarrow}_G a_2$ and $\langle b_0, b_2 \rangle \stackrel{*}{\Rightarrow}_G b_2$ then $\langle a_0 b_0, a_1 b_1 \rangle \stackrel{*}{\Rightarrow}_G a_2 b_2$.
2. For all $a_0, b_0, w_1, w_2 \in A_G$, if $\langle a_0 b_0, w_1 \rangle \stackrel{*}{\Rightarrow}_G w_2$, then there exist $a_1, a_2, b_1, b_2 \in A_G$ such that $a_1 b_1 = w_1,\ a_2 b_2 = w_2,\ \langle a_0, a_1 \rangle \stackrel{*}{\Rightarrow}_G a_2$ and $\langle b_0, b_1 \rangle \stackrel{*}{\Rightarrow}_G b_2$.

If these were the only constraints on concatenation, they could be satisfied by the rule equation

$$\rho(a_1 \cdot a_2) \quad = \quad \{\ \langle v_0, v_1 v_2 \rangle \quad \rightarrow \quad \langle a_1, v_1 \rangle \langle a_2, v_2 \rangle\ \}$$

By Theorem 6.10, this rule equation is safe.

Unfortunately, the situation is complicated by two additional constraints, neither of which is satisfied by the above rule equation.

3. For all $a_1, a_2, a_3 \in A_G,\ \rho_G((a_1 a_2) a_3) = \rho_G(a_1 (a_2 a_3))$.
4. There exists $z \in A_G$ such that, for all $(t_1 t_2) \in T_G$,

$$\begin{aligned} \rho_G(t_1) &= \{\langle v_0, z \rangle \rightarrow t_1\} \\ \rho_G(t_2) &= \{\langle v_0, z \rangle \rightarrow t_2\} \\ \rho_G(t_1 t_2) &= \{\langle v_0, z \rangle \rightarrow t_1 t_2\} \end{aligned}$$

The former constraint arises from the associativity of concatenation, the latter from the definition of unrestricted RAG (Definition 5.22, §5.3).

In order to formulate a rule equation consistent with all four constraints, the following auxiliary definition is introduced.

Definition 6.10 Suppose r is an unbound rule with variables $\{v_0, \cdots v_n\}$, and r' is an unbound rule with variables $\{v'_0, \cdots v'_m\}$. Then the *concatenation* of r and r', denoted $r \cdot r'$, is an unbound rule as follows.

Let $\{v_{n+1}, \cdots v_{n+m}\}$ be variables distinct from all operators and variables in both rules. Let r'' be the unbound rule obtained from r' by substituting $v'_0 = v_0$ and, for all $1 \leq i \leq m,\ v'_i = v_{n+i}$. Finally, let e_0, e''_0 be the unique query polynomials,

and π, π'' the unique configuration polynomials, such that $r = (\langle v_0, e_0 \rangle \to \pi)$ and $r'' = (\langle v_0, e_0'' \rangle \to \pi'')$. Then

$$r \cdot r' \;=\; \langle v_0, e_0 \cdot e_0'' \rangle \to \pi \cdot \pi''$$

□

Technically, this definition is unique only up to choice of the variables $v_{n+1}, \cdots v_{n+m}$. This is considered sufficient because, as noted earlier (following Definition 5.20, §5.3), unbound rules are normally considered identical if they differ only in their choice of variables.

The following rule equation for concatenation is consistent with all four of the desired constraints.

$$\rho(a_1 \cdot a_2) \;=\; \{\, r_1 \cdot r_2 \mid r_1 \in \rho(a_1) \text{ and } r_2 \in \rho(a_2) \,\} \tag{6.1}$$

Moreover,

Theorem 6.11 Rule equation 6.1 is safe. □

For the proof of this theorem, it is convenient to separate out the following lemma.

Lemma 6.12 Suppose G is a RAG that satisfies rule equation 6.1, $a_1, a_2, t, a' \in A_G$, and $(a_1 a_2) : t \stackrel{*}{\Rightarrow}_G a'$. Then there must exist $t_1, t_2, a_1', a_2' \in A_G$ such that $t = t_1 t_2$, $a' = a_1' a_2'$, $a_1 : t_1 \stackrel{*}{\Rightarrow}_G a_1'$, and $a_2 : t_2 \stackrel{*}{\Rightarrow}_G a_2'$. □

Proof. Suppose G etc. as in the lemma.

By the definition of the RAG derivation relation, since $(a_1 a_2) : t \stackrel{*}{\Rightarrow}_G a'$, there must exist an unbound rule $r \in \rho_G(a_1 \cdot a_2)$ and a bound rule $(\langle a_1 \cdot a_2, q \rangle \to c) \in \beta_G(r)$ such that $q \stackrel{*}{\Rightarrow}_G a'$ and $c \stackrel{*}{\Rightarrow}_G t$, thus

$$\langle a_1 a_2, a' \rangle \;\stackrel{*}{\Rightarrow}_G\; \langle a_1 a_2, \overline{q} \rangle \;\Rightarrow_G\; c \;\stackrel{*}{\Rightarrow}_G\; a'$$

By the rule equation, there must exist unbound rules $r_1 \in \rho_G(a_1)$ and $r_2 \in \rho_G(a_2)$, and bound rules $(\langle a_1, q_1 \rangle \to c_1) \in \beta_G(r_1)$ and $(\langle a_2, q_2 \rangle \to c_2) \in \beta_G(r_2)$, such that $q = q_1 \cdot q_2$ and $c = c_1 \cdot c_2$.

Again by the definition of the RAG derivation relation, since $(q_1 \cdot q_2) \stackrel{*}{\Rightarrow}_G a'$ there must exist $a_1', a_2' \in A_G$ such that $a' = a_1' \cdot a_2'$, $q_1 \stackrel{*}{\Rightarrow}_G a_1'$, and $q_2 \stackrel{*}{\Rightarrow}_G a_2'$. Likewise, since $(c_1 \cdot c_2) \stackrel{*}{\Rightarrow}_G t$, there must exist $t_1, t_2 \in A_G$ such that $t = t_1 \cdot t_2$, $c_1 \stackrel{*}{\Rightarrow}_G t_1$, and $c_2 \stackrel{*}{\Rightarrow}_G t_2$. Thus,

$$\begin{array}{ll} a_1 : t_1 & \stackrel{*}{\Rightarrow}_G \; a_1' \\ a_2 : t_2 & \stackrel{*}{\Rightarrow}_G \; a_2' \end{array}$$

as required by the lemma. □

Proof. (Theorem 6.11) Suppose G is a RAG that satisfies rule equation 6.1. Let $\equiv$ be the unique binary relation over A_G such that, for all $a, b \in A_G$, $a \equiv b$ iff there exist $a_1, a_2, b_1, b_2 \in A_G$ such that $a = a_1 a_2$, $b = b_1 b_2$, $a_1 \equiv_G b_1$, and $a_2 \equiv_G b_2$. It will be shown that $\equiv$ is an answer-equivalence; hence the binary concatenation operator is G-safe; hence the rule equation is safe.

Suppose $a_1, a_2, b_1, b_2, t, a' \in A_G$, $a_1 \equiv_G b_1$, $a_2 \equiv_G b_2$, and $(a_1 \cdot a_2) : t \stackrel{*}{\underset{G}{\Rightarrow}} a'$. By the lemma, there must exist $t_1, t_2, a'_1, a'_2 \in A_G$ such that $t = t_1 t_2$, $a' = a'_1 a'_2$, $a_1 : t_1 \stackrel{*}{\underset{G}{\Rightarrow}} a'_1$, and $a_2 : t_2 \stackrel{*}{\underset{G}{\Rightarrow}} a'_2$. By the definition of answer-equivalence, there must exist $b'_1, b'_2 \in A_G$ such that $b_1 : t_1 \stackrel{*}{\underset{G}{\Rightarrow}} b'_1$ and $b_2 : t_2 \stackrel{*}{\underset{G}{\Rightarrow}} b'_2$. By the rule equation, $(b_1 b_2) : t \stackrel{*}{\underset{G}{\Rightarrow}} b'_1 b'_2$. □

The unary *star* operator

The unary *star* operator was defined in Example 4.3, §4.3, by the rule equation

$$\rho(star(a)) \;=\; \left\{ \begin{array}{rcl} \langle v_0, \lambda \rangle & \rightarrow & \lambda \\ \langle v_0, v_1 v_2 \rangle & \rightarrow & \langle a, v_1 \rangle \langle star(a), v_2 \rangle \end{array} \right\} \tag{6.2}$$

In and of itself, this rule equation is not safe; that is, a framework containing this rule equation and no other would not be safe. However, it does form a safe framework when taken in combination with the rule equation for concatenation.

Theorem 6.13 Suppose $\mathcal{F} = \langle \Phi, \mathcal{E} \rangle$ is a framework, and $\mathcal{E}$ consists exactly of rule equations 6.1 and 6.2. Then $\mathcal{F}$ is safe. □

Proof. Suppose $\mathcal{F}$ etc. as in the theorem. Since the rule equations do not conflict, $\mathcal{F}$ is satisfiable. Suppose $G \in \mathcal{G}_{\mathcal{F}}$ is a RAG that satisfies $\mathcal{F}$, $a, b, t, a' \in A_G$, $star(a) : t \stackrel{*}{\underset{G}{\Rightarrow}} a'$, and $a \equiv_G b$. It will be shown that there must exist some $b' \in A_G$ such that $star(b) : t \stackrel{*}{\underset{G}{\Rightarrow}} b'$ and $a' \equiv_G b'$.

Since $star(a) : t \stackrel{*}{\underset{G}{\Rightarrow}} a'$, rule equation 6.2 implies by induction that, for some $n \geq 0$, there exist $t_1, \cdots t_n \in A_G$ and $a'_1, \cdots a'_n \in A_G$ such that $t = t_1 t_2 \cdots t_n$, $a' = a'_1 a'_2 \cdots a'_n$, and, for all $1 \leq i \leq n$, $a : t_i \stackrel{*}{\underset{G}{\Rightarrow}} a'_i$. But $a \equiv_G b$, therefore there must exist $b'_1, \cdots b'_n \in A_G$ such that, for all $1 \leq i \leq n$, $b : t_i \stackrel{*}{\underset{G}{\Rightarrow}} b'_i$. Let $b' = b'_1 b'_2 \cdots b'_n$. Rule equation 6.2 implies by induction that $b : t \stackrel{*}{\underset{G}{\Rightarrow}} b'$. Theorem 6.11 guarantees that $a' \equiv_G b'$. □

Note that, in general, it cannot be assumed that binary concatenation is associative. In the three strings in the proof (t, a', and b'), concatenation is grouped to the *right*, i.e., $t = t_1 \cdot (t_2 \cdot (\cdots (t_{n-1} t_n) \cdots))$ and so on.

The binary *and* operator

The binary *and* operator was defined in Example 4.4, §4.3, by the rule equation

$$\rho(and(a_1, a_2)) \;=\; \left\{ \; \langle v_0, (a_1 : v_1) \cdot (a_2 : v_1) \rangle \;\rightarrow\; \langle star(echo), v_1 \rangle \; \right\} \tag{6.3}$$

Like rule equation 6.2, this rule equation is not safe in itself, but does form a safe framework in combination with the rule equation for concatenation.

Theorem 6.14 Suppose $\mathcal{F} = \langle \Phi, \mathcal{E} \rangle$ is a framework, and $\mathcal{E}$ consists exactly of rule equations 6.1 and 6.3. Then $\mathcal{F}$ is safe. □

Proof. Suppose $\mathcal{F}$ etc. as in the theorem. Since the rule equations do not conflict, $\mathcal{F}$ is satisfiable. Suppose $G \in \mathcal{G}_{\mathcal{F}}$ is a RAG that satisfies $\mathcal{F}$, $a_1, a_2, b_1, b_2, t, a' \in A_G$, $and(a_1, a_2) : t \overset{*}{\underset{G}{\Rightarrow}} a'$, $a_1 \equiv_G b_1$ and $a_2 \equiv_G b_2$. It will be shown that there must exist some $b' \in A_G$ such that $and(b_1, b_2) : t \overset{*}{\underset{G}{\Rightarrow}} b'$ and $a' \equiv_G b'$.

By rule equation 6.3, $and(a_1, a_2) : t \overset{*}{\underset{G}{\Rightarrow}} a'$ implies that there must exist some $t' \in A_G$ such that $star(echo) : t \overset{*}{\underset{G}{\Rightarrow}} t'$ and $(a_1 : t') \cdot (a_2 : t') \overset{*}{\underset{G}{\Rightarrow}} a'$, thus

$$\langle and(a_1, a_2), a' \rangle \overset{*}{\underset{G}{\Rightarrow}} \langle and(a_1, a_2), \overline{(a_1 : t') \cdot (a_2 : t')} \rangle \underset{G}{\Rightarrow} \langle star(echo), t' \rangle \overset{*}{\underset{G}{\Rightarrow}} t$$

Since $(a_1 : t') \cdot (a_2 : t') \overset{*}{\underset{G}{\Rightarrow}} a'$, there must (by definition of the RAG derivation relation) be some $a'_1, a'_2 \in A_G$ such that $a' = a'_1 \cdot a'_2$, $a_1 : t' \overset{*}{\underset{G}{\Rightarrow}} a'_1$, and $a_2 : t' \overset{*}{\underset{G}{\Rightarrow}} a'_2$. But $a_1 \equiv_G b_1$ and $a_2 \equiv_G b_2$, therefore there must exist $b'_1, b'_2 \in A_G$ such that $b_1 : t' \overset{*}{\underset{G}{\Rightarrow}} b'_1$, $b_2 : t' \overset{*}{\underset{G}{\Rightarrow}} b'_2$, $a'_1 \equiv_G b'_2$, and $a'_2 \equiv_G b'_2$. Let $b' = b'_1 \cdot b'_2$. By Theorem 6.11, concatenation is G-safe; therefore, $b' \equiv_G a'$.

By rule equation 6.3, $\langle and(b_1, b_2), \overline{(b_1 : t') \cdot (b_2 : t')} \rangle \underset{G}{\Rightarrow} \langle star(echo), t' \rangle$. But it is already known that $\langle star(echo), t' \rangle \overset{*}{\underset{G}{\Rightarrow}} t$ and $(b_1 : t') \cdot (b_2 : t') \overset{*}{\underset{G}{\Rightarrow}} b'$. Therefore,

$$\langle and(b_1, b_2), b' \rangle \overset{*}{\underset{G}{\Rightarrow}} \langle and(b_1, b_2), \overline{(b_1 : t') \cdot (b_2 : t')} \rangle \underset{G}{\Rightarrow} \langle star(echo), t' \rangle \overset{*}{\underset{G}{\Rightarrow}} t$$

In short, $and(b_1, b_2) : t \overset{*}{\underset{G}{\Rightarrow}} b'$. □

The unary *decl* operator

The unary *decl* operator was defined in Example 4.4, §4.3, by the rule equation

$$\rho(decl(e)) \quad = \quad \left\{ \begin{array}{rl} \langle v_0, combine(e, make\text{-}env : v_1) \rangle & \\ \quad \rightarrow \quad \text{“}\mathbf{int}\text{”} \ \langle echo \cdot star(echo), v_1 \rangle & \\ \qquad\qquad \langle (e : undef) : v_1, v_2 \rangle \ \text{“}\mathbf{;}\text{”} & \end{array} \right\} \tag{6.3A}$$

Again, this rule equation is not safe in itself, but forms a safe framework when combined with certain other rule equations. Specifically, sufficient rule equations are needed to establish the safety of those operators in 6.3A that have variable arguments: concatenation (rule equation 6.1), and *combine*. The safety of *combine* requires, in turn, *and* (rule equation 6.3) and ⊔.

$$\rho(combine(a, b)) \quad = \quad \left\{ \begin{array}{rcl} \langle v_0, and(a : undef, b : undef) \rangle & \rightarrow & undef \\ \langle v_0, (a : def) \sqcup (b : def) \rangle & \rightarrow & def \end{array} \right\} \tag{6.3B}$$

$$\rho(a \sqcup b) \quad = \quad \left\{ \begin{array}{rcl} \langle v_0, v_1 \rangle & \rightarrow & \langle a, v_1 \rangle \\ \langle v_0, v_2 \rangle & \rightarrow & \langle b, v_2 \rangle \end{array} \right\} \tag{6.3C}$$

Theorem 6.14A Suppose $\mathcal{F} = \langle \Phi, \mathcal{E} \rangle$ is a framework, and $\mathcal{E}$ consists exactly of rule equations 6.1, 6.3, 6.3A, 6.3B, and 6.3C. Then $\mathcal{F}$ is safe. □

Proof. Suppose $\mathcal{F}$ etc. as in the theorem. The rule equations don't conflict, so $\mathcal{F}$ is satisfiable. By previous theorems of this section, all the specified operators of $\mathcal{F}$ other than *decl* are safe (Theorems 6.7, 6.8, 6.10, 6.11, and 6.14); so it only remains to show that if the others are safe, so is *decl*.

Suppose $G \in \mathcal{G}_{\mathcal{F}}$ is a RAG that satisfies $\mathcal{F}$, $a, b, t, a' \in A_G$, $decl(a) : t \overset{*}{\underset{G}{\Rightarrow}} a'$, and $a \equiv_G b$. It will be shown that there must exist some $b' \in A_G$ such that $decl(b) : t \overset{*}{\underset{G}{\Rightarrow}} b'$ and $a' \equiv_G b'$.

By rule equation 6.3A, there exist $t_1, t_2, t_1', t_2', e \in A_G$ such that

$t =$ "**int**" t_1 t_2 "**;**"

$\langle echo \cdot star(echo), t_1' \rangle \overset{*}{\underset{G}{\Rightarrow}} t_1$

$\langle (a : undef) : t_1', t_2' \rangle \overset{*}{\underset{G}{\Rightarrow}} t_2$

$make\text{-}env : t_1' \overset{*}{\underset{G}{\Rightarrow}} e$

Since $a \equiv_G b$, for any answer derived from $a : undef$ there is an $\equiv_G$ answer derived from $b : undef$. Therefore, since t_1' is accepted by the former, it must be accepted by the latter, and for any answer derived from $(a : undef) : t_1'$ there must be an $\equiv_G$ answer derived from $(b : undef) : t_1'$, and

$\langle (a : undef) : t_1', t_2'' \rangle \overset{*}{\underset{G}{\Rightarrow}} t_2$

(It also happens that $t_2'' \equiv_G t_2'$, but since t_2' is discarded, we don't need that result.)

Therefore, $decl(b) : t \overset{*}{\underset{G}{\Rightarrow}} combine(b, e)$, so it only remains to show that $combine(b, e) \equiv_G combine(a, e)$; but this follows from the safety of *combine*. □

The unary *decl-list* operator

The unary *decl-list* operator was defined in Example 4.4, §4.3, by the rule equation

$$\rho(decl\text{-}list(e)) \quad = \quad \left\{ \begin{array}{lcl} \langle v_0, e \rangle & \rightarrow & \lambda \\ \langle v_0, v_2 \rangle & \rightarrow & \langle decl(e), v_1 \rangle \, \langle decl\text{-}list(v_1), v_2 \rangle \end{array} \right\} \qquad (6.3D)$$

Since this rule uses *decl* with a variable parameter, the safe framework will have to include all the rule equations from Theorem 6.14A.

Theorem 6.14B Suppose $\mathcal{F} = \langle \Phi, \mathcal{E} \rangle$ is a framework, and $\mathcal{E}$ consists exactly of rule equations 6.1, 6.3, 6.3A, 6.3B, 6.3C, and 6.3D. Then $\mathcal{F}$ is safe. □

The proof is substantially similar to that of Theorem 6.13 (for operator *star*), with some added verbosity because the declarations in the sequence don't all use the same meta-syntax.

6.3.3 Unsafe frameworks

In the early stages of development of the RAG model, the author identified three elementary binary answer operators as being particularly useful: concatenation, union, and mapping. The original intent of the mapping operator was merely to allow the construction of conventional namespaces, i.e., simple mappings from names to values. For this purpose, the following rule equation was envisioned.

$$\rho([a,b]) \;=\; \{\; \langle v_0, b\rangle \;\to\; a \;\} \tag{6.4}$$

Cf. the rule equation for the mapping operator in Proposition 4.1, §4.2.2.

Subsequent investigation showed, however, that rule equation 6.4 leads to some disturbing anomalies. Specifically, in combination with concatenation and union, it allows the construction of a "decompiler" for any given grammar: an answer that performs an injective (one-to-one) map from answers into terminal strings. This observation led directly to the concept of strong answer-encapsulation.

Theorem 6.15 Every framework that includes rule equation 6.4 is unsafe. □

Proof. Suppose $\mathcal{F} = \langle \Phi, \mathcal{E}\rangle$ is a framework that includes rule equation 6.4. If $\mathcal{F}$ is not satisfiable, by definition it is not safe. Suppose $\mathcal{F}$ is satisfiable.

Let $\lambda \in \Phi_0$ be an operator not specified by $\mathcal{F}$. Let e be the following rule equation.

$$\rho(\lambda) \;=\; \{\; \langle v_0, \lambda\rangle \;\to\; \lambda \;\}$$

Let $\mathcal{F}' = \langle \Phi, \mathcal{E} \cup \{e\}\rangle$. Since $\mathcal{F}$ is a framework, Φ_0 contains a countably infinite number of operators not specified by $\mathcal{F}$, hence a countably infinite number of operators not specified by $\mathcal{F}'$. Thus, $\mathcal{F}'$ is a framework. Since λ is not specified by $\mathcal{F}$, e does not conflict with $\mathcal{E}$; since $\mathcal{F}$ is satisfiable, $\mathcal{E}$ has no internal conflicts. Thus, $\mathcal{F}'$ is satisfiable. Suppose $G \in \mathcal{G}_{\mathcal{F}'}$ is a RAG that satisfies $\mathcal{F}'$.

Consider the answers λ, $[\lambda,\lambda]$, and $[[\lambda,\lambda],\lambda]$. By rule equations e and 6.4,

$$\begin{aligned}
\rho_G(\lambda) &= \{\; \langle v_0, \lambda\rangle \;\to\; \lambda \;\} \\
\rho_G([\lambda,\lambda]) &= \{\; \langle v_0, \lambda\rangle \;\to\; \lambda \;\} \\
\rho_G([[\lambda,\lambda],\lambda]) &= \{\; \langle v_0, \lambda\rangle \;\to\; [\lambda,\lambda] \;\}
\end{aligned}$$

The only possible derivations of the form $\langle x, y\rangle \overset{*}{\underset{G}{\Rightarrow}} z$ with $x \in \{\lambda, [\lambda,\lambda], [[\lambda,\lambda],\lambda]\}$ and $y, z \in A_G$ are

$$\begin{aligned}
\langle \lambda, \lambda\rangle &\overset{*}{\underset{G}{\Rightarrow}} \lambda \\
\langle [\lambda,\lambda], \lambda\rangle &\overset{*}{\underset{G}{\Rightarrow}} \lambda \\
\langle [[\lambda,\lambda],\lambda], \lambda\rangle &\overset{*}{\underset{G}{\Rightarrow}} [\lambda,\lambda]
\end{aligned}$$

By the definition of answer-equivalence, $\lambda \equiv_G [\lambda,\lambda]$ but $[\lambda,\lambda] \not\equiv_G [[\lambda,\lambda],\lambda]$.

Suppose the operator specified by rule equation 6.4 is G-safe. Then for all $a, b \in A_G$, $a \equiv_G b$ implies $[a,\lambda] \equiv_G [b,\lambda]$. But with $a = \lambda$ and $b = [\lambda,\lambda]$, $a \equiv_G b$ and $[a,\lambda] \not\equiv_G [b,\lambda]$. Therefore, the operator specified by rule equation 6.4 is not G-safe.

Since G satisfies $\mathcal{F}'$, by construction G must satisfy $\mathcal{F}$; and since the operator specified by rule equation 6.4 is not G-safe, by definition $\mathcal{F}$ is not safe. □

The above proof actually suggests an even stronger result: If G is a RAG that satisfies rule equation 6.4, and T_G is not empty, then G is not weakly answer-encapsulated.

6.4 Conventions

This section gathers together several conventions that are observed in the normal usage of RAGs (such as in Chapter 4, especially §4.3). The conventional notation for specifying RAGs is not repeated here. One simple but important theorem is stated and proved.

Definition 6.11 A RAG is *well-behaved* iff it is both strongly stepwise decidable and strongly answer-encapsulated. □

Definition 6.12 Suppose G is a RAG, A_G is the string algebra over an alphabet Z, and G satisfies rule equation 6.1. Then G is a *string RAG (over alphabet Z)*. □

Recall from Example 5.1, §5.2.3, that the string algebra over Z is the initial algebra with zero-ary operators $Z \cup \{\lambda\}$, the binary concatenation operator, and as Σ-equations the usual properties of concatenation.

Convention 6.1 All RAGs are meant to be well-behaved string RAGs unless otherwise stated. (Whether they satisfy this intention is still subject to proof, however.) □

The definition of unrestricted RAG (Definition 5.22, §5.3) requires that there exist some $z \in T_G$ such that for all $t \in T_G$, $\rho(t) = \{\langle v_0, z\rangle \to t\}$. For the particular case of string RAGs, the value of z is uniquely determined.

Theorem 6.16 If G is a string RAG then, for all $t \in T_G^*$,

$$\rho(t) \;=\; \{\; \langle v_0, \lambda\rangle \;\to\; t \;\}$$

□

Proof. Suppose G is a string RAG. Then $\lambda \in T_G$, and G satisfies rule equation 6.1. The definition of unrestricted RAG requires that there exist some $z \in T_G$ such that

$$\rho(\lambda) \;=\; \{\; \langle v_0, z\rangle \;\to\; \lambda \;\}$$

By rule equation 6.1,

$$\rho(\lambda \cdot \lambda) \;=\; \{\; \langle v_0, z \cdot z \rangle \;\rightarrow\; \lambda \cdot \lambda \;\}$$

Substituting $\lambda \cdot \lambda = \lambda$,

$$\rho(\lambda) \;=\; \{\; \langle v_0, z \cdot z \rangle \;\rightarrow\; \lambda \;\}$$

Therefore, $z \cdot z = z$. But the only element of T_G with this property is λ, so $z = \lambda$. □

Convention 6.2 Unless otherwise stated, rule equation 6.2 and the following other rule equations are satisfied by all RAGs that include the specified operators. (Hence, these operators can be used without having to repeat the rule equations.)

$$\begin{aligned}
\rho(a_1 \sqcup a_2) &= \left\{ \begin{array}{lcl} \langle v_0, v_1 \rangle & \rightarrow & \langle a_1, v_1 \rangle \\ \langle v_0, v_1 \rangle & \rightarrow & \langle a_2, v_1 \rangle \end{array} \right\} \\
\rho([a_1, a_2]) &= \{\; \langle v_0, a_2 \rangle \;\rightarrow\; \langle a_1, v_1 \rangle \;\} \\
\rho(\emptyset) &= \{\}
\end{aligned}$$

□

(The rule equations listed above were first stated in Proposition 4.1, §4.2.2.)

6.5 Computational power

It follows immediately from Theorem 6.1 that every language accepted by a well-behaved RAG is recursively enumerable, and that every function computed by a well-behaved RAG is partial recursive. The following theorems show that every recursively enumerable language is accepted by a well-behaved RAG, and that every partial recursive function is computed by a well-behaved RAG.

Theorem 6.17 For every type 0 Chomsky grammar G, there exists a well-behaved string RAG H such that $L(H) = L(G)$. □

Proof. Let $G = \langle Z, T, R, s\rangle$ be a general Chomsky grammar (see Definition 1.3, §1.2). Let H be the following RAG.

$$\begin{array}{lrcl}
\mathit{echoZ}: & \forall z \in Z, \quad \langle v_0, z\rangle & \to & z \\
\mathit{echoT}: & \forall t \in T, \quad \langle v_0, t\rangle & \to & t \\
\mathit{rule}: & \forall (x \to y) \in R, \quad \langle v_0, x\rangle & \to & y \\
\mathit{step}: & \langle v_0, v_1 v_2 v_3\rangle & \to & \langle \mathit{star}(\mathit{echoZ}), v_1\rangle\langle \mathit{rule}, v_2\rangle\langle \mathit{star}(\mathit{echoZ}), v_3\rangle \\
\mathit{derive}: & \langle v_0, \lambda\rangle & \to & s \\
 & \langle v_0, \mathit{derive} : (\mathit{step} : v_1)\rangle & \to & \langle \mathit{star}(\mathit{echoZ}), v_1\rangle \\
\mathit{start}: & \langle v_0, \mathit{derive} : v_1\rangle & \to & \langle \mathit{star}(\mathit{echoT}), v_1\rangle
\end{array}$$

where nonterminal constant *start* is the start symbol of H. By implication, the terminal alphabet of H is Z, thus the terminal algebra of H is Z^*.

Nonterminal constant *echoZ* recognizes any symbol $z \in Z$, and synthesizes semantic value z; nonterminal constant *echoT* does likewise for T. Nonterminal *rule* recognizes the right side of a Chomsky rule, and synthesizes the left side of that rule. Nonterminal *step* recognizes the right side of a Chomsky derivation step, and synthesizes the left side of that step. Formally, for all $x, y \in A_G$,

$$\mathit{echoZ} : x \overset{*}{\underset{H}{\Rightarrow}} y \quad \text{iff} \quad x = y \in Z$$

$$\mathit{echoT} : x \overset{*}{\underset{H}{\Rightarrow}} y \quad \text{iff} \quad x = y \in T$$

$$\mathit{rule} : x \overset{*}{\underset{H}{\Rightarrow}} y \quad \text{iff} \quad (y \to x) \in R$$

$$\mathit{step} : x \overset{*}{\underset{H}{\Rightarrow}} y \quad \text{iff} \quad y \underset{G}{\Rightarrow} x$$

Suppose $x, a \in A_G$. Then $\mathit{derive} : x \overset{*}{\underset{H}{\Rightarrow}} a$ iff

$$\mathit{derive} : x \quad \overset{*}{\underset{H}{\Rightarrow}} \quad \mathit{derive} : \overline{\langle \mathit{derive}, a\rangle} \quad \underset{H}{\Rightarrow} \quad a$$

The only answer that can be synthesized by *derive* is λ; so $a = \lambda$.

$$\langle \mathit{derive}, \lambda\rangle \quad \overset{*}{\underset{H}{\Rightarrow}} \quad x$$

Now, either $x = s$, or else

$$\langle \mathit{derive}, \lambda\rangle \quad \overset{*}{\underset{H}{\Rightarrow}} \quad \langle \mathit{derive}, \overline{\mathit{derive} : (\mathit{step} : x)}\rangle \quad \underset{H}{\Rightarrow} \quad \langle \mathit{star}(\mathit{echoZ}), x\rangle \quad \underset{H}{\Rightarrow} \quad x$$

$$\mathit{derive} : (\mathit{step} : x) \quad \overset{*}{\underset{H}{\Rightarrow}} \quad \lambda$$

Either way, $s \stackrel{*}{\underset{G}{\Rightarrow}} x$. Thus,

$$L_H(derive) \;=\; \{\omega \in Z^* \mid s \stackrel{*}{\underset{G}{\Rightarrow}} \omega\}$$

The start symbol *start* recognizes just those strings over T that are also recognized by *derive*, and synthesizes semantic value λ. In short, *start* is a predicate for $L(G)$.

$$L(H) \;=\; \{w \in T^* \mid w \in L_H(derive)\} \;=\; \{w \in T^* \mid s \stackrel{*}{\underset{G}{\Rightarrow}} w\} \;=\; L(G)$$

□

Theorem 6.18 For every deterministic Turing machine M with tape alphabet Z and input alphabet T, there exists a well-behaved string RAG G such that for all strings $t \in T^*$ and $z \in Z^*$, $s_G : t \stackrel{*}{\underset{G}{\Rightarrow}} z$ iff $f_M(t) = z$. □

That is, G can synthesize z when generating t iff M can output z when accepting t. See §1.4.2, Definitions 1.9–1.12.

Proof. Let $M = \langle Q, Z, T, \delta, q_0\rangle$ be a Turing machine. Let $\star$ be a symbol not in Z. Let L be the language

$$L \;=\; \{t \star z \mid f_M(t) = z\}$$

Since f_M is computed by M, one can construct a Turing machine M' that accepts language L by the algorithm

1. Run M on t as a subroutine.
2. Compare the output $f_M(t)$ to z. If they are equal, halt; otherwise, loop forever.

Machine M' will halt on input $t \star z$ iff M halts on t with output z. Thus, L is a recursively enumerable language; so there must exist a type 0 Chomsky grammar that generates L.

By the procedure used in the proof of the preceding theorem (6.17), let H be a RAG whose start symbol *startH* is a predicate for L. Let G be the RAG formed by augmenting H as follows, with start symbol *startG* and terminal alphabet $Z \cup \{\star\}$.

$$\begin{array}{lrcl}
in\text{:} & \langle v_0, v_1 v_2\rangle & \rightarrow & \langle in, v_1\rangle\langle in, v_2\rangle \\
 & \forall t \in T \cup \{\lambda\},\quad \langle v_0, t\rangle & \rightarrow & t \\
 & & & \\
out\text{:} & \langle v_0, v_1 v_2\rangle & \rightarrow & \langle out, v_1\rangle\langle out, v_2\rangle \\
 & \forall z \in Z \cup \{\lambda\},\quad \langle v_0, z\rangle & \rightarrow & \lambda \\
 & & & \\
startG\text{:} & \langle v_0, v_2\rangle & \rightarrow & \langle in, v_1\rangle\langle out, v_2\rangle\langle startH : (v_1 \star v_2), v_3\rangle
\end{array}$$

The first unbound pair on the right side of the unbound rule for *startG* recognizes any string $t \in T^*$, and binds $v_1 = t$. The second pair recognizes λ, and randomly binds v_2 to some string $z \in Z^*$. The third pair cannot be resolved unless $t \star z \in L$, in which case it recognizes λ. The overall effect is that *startG* recognizes just those strings $t \in T^*$ on which M halts, and synthesizes semantic value $f_M(t)$. □

6.6 Non-circularity

In the context of extended attribute grammars, non-circularity (Definition 2.4, §2.2.3) meant that the attributes of any given parse tree could be computed in a sequential order. Non-circularity is significant in that setting primarily because it facilitates the automatic generation of AG-based compilers. Christiansen suggested a similar property, in his class of left-to-right Christiansen grammars [Chri 86], in order to facilitate automatic parsing.

RAG-based parsing issues, let alone RAG-based compiling issues, are mostly beyond the scope of the current thesis. However, circular grammar adaptation presents difficulties even on purely *conceptual* grounds. These difficulties will be discussed in §7.1.3. Meanwhile, the current section merely formally defines the concept of non-circularity as it applies to RAGs.

Actually, RAG non-circularity is a much simpler property than its attribute grammar counterpart. Non-circularity of an attribute grammar must be determined, in general, by considering the class of all possible (non-attributed) parse trees for that grammar. But because the RAG model is limited to just two "attributes" (see §4.4), non-circularity can be determined separately for each unbound rule.

Definition 6.13 Suppose r is an unbound rule over a string RAG, of the following form.

$$\langle v_0, e_0 \rangle \quad \rightarrow \quad t_0 \langle e_1, v_1 \rangle t_1 \cdots \langle e_n, v_n \rangle t_n$$

Then r is *non-circular* iff there exists a permutation $k_1, \cdots k_n$ of the integers from 1 to n such that, for all $1 \leq i \leq n$, e_{k_i} uses only variables v_0 and $\{v_{k_j} \mid j < i\}$. (Thus, e_{k_1} uses no variable except v_0, and so on.) Further, r is *left-to-right* iff the identity permutation $k_i = i$ satisfies this condition.

A RAG G is non-circular iff, for all $a \in A_G$ and all $r \in \rho_G(a)$, r is non-circular. G is left-to-right iff, for all such r, r is left-to-right. □

With one exception, all of the unbound rules given in the thesis are left-to-right, including those implicitly generated by rule equations. Even the unsafe framework in §6.3.3 is left-to-right. The exception is a rule given in Example 4.4 of §4.3 (as an alternative implementation for operator *decl*), and specifically identified there as circular.

Chapter 7

Comments

This chapter has two sections. §7.1 compares RAGs to some of the models surveyed in Part I. §7.2 discusses some possible areas for further research involving the RAG model.

7.1 Description of programming languages

Christiansen devoted one section of his survey article ([Chri 90]) to "Difficulties in the adaptable grammar approach [to programming language description]". Some of these he claimed to have solved completely, some partially, and some (he candidly admitted) not at all. The following subsections discuss these difficulties, and especially the relevance of the RAG model thereto.

The principle of grammar adaptability originates with the notion of adding grammar rules as a way of describing the effect of declaring new objects in a program [Cara 63]. There is, however, much more to adaptability in practice than simply adding rules at need. *When* (or, equivalently, *where*) to add rules, when (where) to remove them, and even *which* rules to add/remove can become sticking points. Most of the subsections deal, more or less directly, with some aspect of these problems.

7.1.1 Removing rules at block exit

Most programming languages —very nearly all, in fact— observe some form of block-oriented scope. Many allow multiple nested levels of block-oriented scope. What this means in terms of grammar adaptability is that the scope of new grammar rules is usually limited to a particular branch of the parse tree. So in order to provide a natural medium for describing programming languages, an adaptable grammar model might be expected to provide a graceful way to express this kind of restriction on the range of a grammar rule.

But remarkably enough, most adaptable grammar formalisms do *not* handle block-oriented scope gracefully. Recall from Chapter 3 that most adaptable grammar for-

malisms are imperative. The imperative approach to adaptability requires that explicit instructions be performed to remove each rule at its appropriate block exit; and although this *can* be done in any sufficiently powerful formalism, it is not particularly easy or natural.

Specifically, recall that the three imperative adaptable grammar models surveyed were (§3.2.1):

1. Wegbreit's ECFGs, in which grammar adaptation is orchestrated by a finite automaton that scans the input string.

2. Mason's DTTs, in which grammar adaptation instructions are attached to grammar rules as side-effects.

3. Burshteyn's modifiable grammars, in which grammar adaptation is orchestrated by a Turing machine that scans the partial derivation just before each parser action.

Wegbreit's ECFGs are not sufficiently powerful. Mason's DTTs might be forced to the task, by arranging a dummy nonterminal that will be expanded at block exit; the appropriate grammar modifications could then be attached to the rule that expands the dummy nonterminal. Burshteyn's modifiable grammars can always recreate the appropriate set of grammar rules by re-parsing the entire input string up to the current location, based on the partial derivation provided. Both solutions are clumsy at best.

In contrast, Christiansen grammars and RAGs handle this sort of thing so easily that, if one had never seen an imperative adaptable grammar model, one would never have imagined the problem could exist. Nested block-oriented scope occurs naturally in all attributed grammar models, that is, all grammar models that base their data flow on the structure of context-free parse trees. In short, removing rules at block exit is a solved problem.

7.1.2 Delayed or indirect declarations

Christiansen ([Chri 90]) loosely defines an indirect declaration as "a construct [that] gives access to entities declared somewhere else in the program text", such as the Pascal **with** statement. For example, consider the following Pascal record type declaration.

```
type complex =
  record
    re,im : real
  end;
```

The immediate consequence of this declaration is, in essence, that it is subsequently possible to declare variables of type **complex**; but there are also indirect consequences. Later, once a complex variable has been declared, an expression of type

real can be formed by specifying the record and one of its fields. Also, the record variable can be the argument to a **with** statement, and within *that*, an expression of type **real** can be formed by specifying the field alone, without the name of the record.

This is a much more complicated case than the elementary variable declarations in the toy language of Examples 3.1, 3.2, and 4.4. The record type declaration may still be thought of as synthesizing a new grammar rule; but now, while generating a variable declaration, the new grammar rule must in turn synthesize additional grammar rules, and those additional rules must synthesize still more.

As usual, Wegbreit's ECFGs do not lend themselves to such antics. Also as usual, Burshteyn's modifiable grammars can do it, but only by resorting to the opaque brute force of a Turing machine. Mason's DTTs may be able to handle multiple levels of indirection; [Maso 87] is unclear on this point.

Christiansen grammars definitely can handle multiple levels of indirection. The requisite technique is illustrated in Example 3.2, §3.3.2 —specifically, in the rule form in that example for nonterminal *decl*:

$$\langle decl \downarrow g \uparrow g \,\&\, new\text{-}rule \rangle \;\rightarrow\; \text{“}\mathbf{int}\text{”}\ \langle alpha\text{-}list \downarrow g \uparrow w \rangle\ \text{“}\mathbf{;}\text{”}$$
$$\text{where}\ \ new\text{-}rule \;=\; \langle id \downarrow h \rangle \rightarrow w$$

Here, the rule form for *decl* has a rule form for *id* embedded in it. The embedded rule form could in principle have had another rule form embedded in it, with another embedded in that, and so on up to any (finite) depth of nesting.

When working with a nested Christiansen rule form, some care must be taken to keep track of when the variables are to be bound. In the above example, variables g and w are bound when nonterminal *decl* is expanded, but variable h remains unbound at that time. The embedded subexpression $\langle id \downarrow h \rangle \rightarrow w$ is rewritten by substituting the synthesized terminal string for w. Later, nonterminal *id* may be expanded to this terminal string, and h would be bound at that time.

Under this notation for Christiansen grammars, each variable in a given rule form is bound iff it occurs in either (1) an inherited attribute position on the left-hand side of the rule form, or (2) a synthesized attribute position on the right-hand side of the rule form. In the example, the enclosing rule form binds g because g is inherited by the left-hand side, and w because w is synthesized by the right-hand side. h is not bound by the enclosing rule form, because it occurs only in a *synthesized* attribute position on the *left-hand* side. The embedded rule form for nonterminal *id* does bind h, because h is inherited by its left-hand side.

Note that the explicit distinction between synthesized and inherited attributes is crucial to determining which of the variables are to be bound at what time. As mentioned in §3.3.1, however, Christiansen's recent work reformulates his grammar model in terms of definite clause grammars. (On DCGs, see §2.2.4.) DCGs do not distinguish between synthesized and inherited attributes. Therefore, Christiansen introduces an explicit notation for designating the nesting-level of each variable: variables bound by the outermost rule form (g and w in the example) are prefixed by a single asterisk;

variables bound at the first embedded level (h in the example) are prefixed by two asterisks, variables at the next level by three asterisks, and so on.

Under the RAG model, there is no embedding of rules in other rules. Rather than explicitly synthesizing new rules, one explicitly constructs an answer, with which a set of unbound rules is *im*plicitly associated by the rule function of the grammar. Wherever a Christiansen grammar uses multiple levels of variables, an analogous RAG would use non-constant answer operators instead. For example, the above Christiansen rule form would translate into a RAG unbound rule such as:

$$\langle v_0, [v_1, \lambda] \rangle \quad \rightarrow \quad \text{“}\textbf{int}\text{”} \ \langle \textit{alpha-list}, v_1 \rangle \ \text{“}\textbf{;}\text{”}$$

Here, the embedded rule form in the Christiansen grammar has been replaced by an invocation of the mapping operator, $[v_1, \lambda]$. (The unbound rule for operator *decl* in Example 4.4, §4.3, is further complicated by additional considerations in the grammar; q.v.)

Since there is no nesting of RAG rules, there is no need for multiple levels of variables in the RAG model. All of the variables in an unbound rule are bound simultaneously. Nevertheless, two distinguished levels of variables are commonly used when *specifying* the rule function of a RAG. Most rule equations (§6.2.2, Definition 6.5) take the form

$$\rho(\sigma(a_1, \cdots a_{ar(\sigma)})) \quad = \quad \left\{ \begin{array}{rcl} \langle v_0, e_{1,0} \rangle & \rightarrow & t_{1,0} \langle e_{1,1}, v_1 \rangle t_{1,1} \cdots t_{1,n_1} \\ & \vdots & \\ \langle v_0, e_{m,0} \rangle & \rightarrow & t_{m,0} \langle e_{m,1}, v_1 \rangle t_{m,1} \cdots t_{m,n_m} \end{array} \right\}$$

The distinguished sets of variables are the a_k, which are the parameters to operator σ, and the v_k, which are not bound until a rule from the constructed rule set $\rho(\sigma(a_1, \cdots a_{ar(\sigma)}))$ is applied in a derivation. (On rule equations of this form, see especially Theorem 6.10 in §6.3.1.)

7.1.3 Recursive declarations

In some of his earlier papers ([Chri 85, Chri 86]), Christiansen characterized recursive declarations by rule forms such as the following. He subsequently discovered that the characterization is incorrect ([Chri 88a, Chri 88b]).

$$\begin{array}{rcl} \langle \textit{stmt} \downarrow g \rangle & \rightarrow & \text{“}\textbf{declare}\text{”} \ \langle \textit{decl-list} \downarrow g \,\&\, h \uparrow h \rangle \\ & & \quad \text{“}\textbf{begin}\text{”} \ \langle \textit{stmt-list} \downarrow g \,\&\, h \rangle \ \text{“}\textbf{end}\text{”} \end{array}$$

The trouble with this rule form is that, because variable h is defined circularly in terms of itself, the declarations associated with nonterminal *decl-list* can define themselves to be grammatically valid, with complete disregard for the inherited grammar g. For example, let *bootstrap* be the following set of rule forms.

$$\textit{bootstrap} \quad = \quad \left\{ \begin{array}{rcl} \langle \textit{decl-list} \downarrow g \uparrow g \rangle & \rightarrow & \text{“}\textbf{Cogito, ergo sum.}\text{”} \\ \langle \textit{stmt-list} \downarrow g \rangle & \rightarrow & \text{“}\textbf{Rene Descartes, 1637.}\text{”} \end{array} \right\}$$

Then, for all possible g, the binding $h = g \,\&\, bootstrap$ satisfies the rule form for nonterminal *stmt*. The following is, therefore, a valid statement.

declare Cogito, ergo sum.
begin Rene Descartes, 1637.
end

In the later Prolog-based formulation of his grammar model, Christiansen gets around the problem of recursive declarations by introducing a multi-pass operator that allows a terminal string to be processed twice, with the second pass using the results from the first [Chri 90]. He describes this operator as a "hack", and asserts that "no known, adaptable grammars provide a satisfactory treatment of recursive declarations". The multi-pass operator does fall outside the normal purview of the Christiansen grammar model. Note, however, that a RAG implementation of the operator involves nothing more that a rather mundane double application of the query operator.

$$\rho(\textit{two-pass}(a)) \;=\; \{\; \langle v_0, (a : v_1) : v_1 \rangle \;\rightarrow\; \langle \textit{star}(\textit{echo}), v_1 \rangle \;\}$$

Assume that $\textit{star}(\textit{echo})$ recognizes exactly the terminal strings $w \in T_G$, and synthesizes value w (cf. Example 4.3, §4.3). Then, for all $a \in A_G$, $\textit{two-pass}(a)$ applies a to an arbitrary terminal string, and applies the semantic value synthesized by a to the same terminal string.

It may well be that the RAG model also affords "single-pass" solutions to the problem; however, the development of such a solution is beyond the scope of the current thesis.

Note that the RAG model is also quite capable of expressing the same incorrect solution originally posed by Christiansen:

$$\textit{stmt}\colon\; \langle v_0, v_2 \rangle \;\rightarrow\; \text{“}\textbf{declare}\text{”}\; \langle \textit{decl-list}(v_1), v_1 \rangle \;\; \text{“}\textbf{begin}\text{”}\; \langle \textit{stmt-list}(v_1), v_2 \rangle \;\text{“}\textbf{end}\text{”}$$

Ordinary circular attribute grammars (Definition 2.4, §2.2.3) do not usually exhibit such extraordinarily anomalous behavior, although Theorem 2.1, §2.2.3, did show a connection between circularity and semantic ambiguity. Rather, the more extreme problem seems to be caused not by circularity per se, but by circular adaptation of the grammar. Nevertheless, since the RAG model does not make any crisp distinction between semantics and meta-syntax, the current author tends to view all circular RAGs (see §6.6) with skepticism.

7.1.4 Multiple declarations

The problem of how to grammatically prohibit multiple declarations has been alluded to several times in the thesis, the earliest being in Example 3.1, §3.1. [Chri 90] remarks:

> . . . the only feasible solution here seems [to be] to go back to the traditional way, over-general grammar rules with additional context constraints (sic!) [sic]. The rule for blocks should, then, include an overall judgment of the grammar synthesized from the declarations.

A solution under the RAG model was developed in §4.3. Although it may be somewhat closer to the "traditional way" than what Christiansen was looking for, it is more decentralized than the pessimistic scenario he describes in the above passage. Hopefully, the RAG solution may also provide more conceptual insight into the nature of the language being described.

7.1.5 Visibility

The declaration of a variable with a variable name that is already in use does not always constitute an error. In most languages, for example, if the block of the second declaration is nested within the block of the first, the inner declaration merely supersedes the outer for the duration of the inner block. [Chri 90] treats this issue separately from that of preventing multiple declarations, under the heading "visibility". His solutions to the two problems are entirely different. To handle visibility, he reluctantly resorts to another "hack" (his word), involving the Prolog cut operator.[1] As with the multi-pass operator for recursive declarations (§7.1.3), he expresses fundamental displeasure with his solution. The cut operator is even more alien to his grammar model than the multi-pass operator, which tends to justify his displeasure in using it.

This problem should be amenable to solution under the RAG model using a technique similar to that developed for the prevention of multiple declarations (Example 4.4, §4.3).

7.1.6 Error handling

When a programming language compiler is constructed from an ordinary attribute grammar, the compiler must handle two different kinds of errors: context-free errors, in which the program text cannot be generated by the underlying Chomsky grammar, and context-dependent errors, in which the program text fails to satisfy additional constraints imposed by the attribute system. When generating diagnostic messages for context-dependent errors, the complete structure of the parse tree is already known, and can therefore be exploited to make the messages more informative.

When the compiler is generated from an adaptable grammar, however, the construction of the parse tree is concurrent with —in fact, indistinguishable from— the imposition of context-dependent constraints. Therefore, less information is available

[1] The Prolog cut operator causes the Prolog processor to use, in this case, the first variable it finds with the appropriate name.

for use in handling context-dependent errors. Playing on this observation, [Chri 90] casts doubt on the feasibility of producing meaningful diagnostic messages at all.

The current author considers Christiansen's gloom on this issue misplaced. Compilers based on ordinary attribute grammars manage to give useful diagnostic messages for context-free errors, when the parse tree is not yet fully formed; the same techniques should be equally applicable to compilers based on adaptable grammars. This expectation is supported by the current author's previous experience with a limited adaptable-grammar parser [Shut 87].

7.2 Directions for future research

7.2.1 RAG-based parsing

The development of RAG-driven language processing techniques is an immediate priority. Not only is it a necessary step in establishing the viability of the RAG model for use in programming language design, but a working RAG-based processor would facilitate other areas of RAG research by allowing empirical experiments.

Most conventional CFG-driven parsers[2] use an algorithm called LR parsing, in which the parse tree is constructed from the bottom up; that is, each parent node in the tree is created only after its descendant branches are already complete. But in a Christiansen grammar (§3.3), nothing can be known in general about the structure of the descendant branches without first determining the language attribute of the parent node. Accordingly, when Christiansen discusses parsing based on his grammar model, in [Chri 86], he is concerned almost exclusively with top-down techniques.

RAGs present a similar challenge for bottom-up parsing techniques, for much the same reason: the structure of descendant branches depends in general on the meta-syntactic value applied from the parent node. Some, though not all, of the parsing techniques suggested by [Chri 86] should be applicable.

7.2.2 Theoretical properties of RAGs

Several opportunities for further research are suggested by Chapter 6.

The following conjectures were stated informally in §6.1:

Conjecture 7.1 There exists a stepwise decidable RAG G such that ρ_G is partial recursive, but equivalence of polynomials over C_G is not Turing-decidable.

There exists a stepwise decidable RAG G such that equivalence of polynomials over C_G is Turing-decidable, but ρ_G is not partial recursive.

There exists a stepwise decidable RAG G such that ρ_G is not partial recursive, and equivalence of polynomials over C_G is not Turing-decidable. □

[2] Basic parsing techniques for context-free grammars are described in [Aho 86].

An area of perhaps more practical interest is the determination of safe and unsafe frameworks. The proof of Theorem 6.10, §6.3.1, relied on a great many restrictions, both in the text of the theorem itself and in the prerequisite definitions of safe query form and safe rule form (Definitions 6.8 and 6.9, also in §6.3.1). Generalizations of the theorem are an immediate possibility, and a variety of mutually disjoint generalizations occurred to the author while developing that theorem. On the other hand, based on the fragmentary nature of the generalizations that suggested themselves, the author would not be entirely surprised to learn that

Conjecture 7.2 The class of safe frameworks with partial recursive rule equations is not Turing-decidable. □

Certainly this conjecture cannot be dismissed without rigorous proof to the contrary.

The issue of RAG-based parsing, discussed in the previous section, entails theoretical work on the formal characterization of RAG subclasses amenable to various parsing techniques.

7.2.3 Theory of abstraction

The author was first led to investigate adaptable grammars by a long-standing interest in broad issues of programming language design; specifically, in the nature of *abstraction*, in its most general and comprehensive sense applicable to programming. This encompasses both the construction of new program entities and the hiding of preexisting ones.[3] For some years, the author has been frustrated by the lack of any solid formal basis on which to make a comprehensive comparative analysis of abstraction support in all existing programming language paradigms.

The principle of grammar adaptability seems ideally suited to the purpose. The construction of new entities and the hiding of preexisting ones are, in effect, elementary concepts in adaptable grammars. (See the remarks at the top of §7.1.) The RAG model is, in a sense, the adaptability principle in its pure form: Turing-powerful adaptability in isolation, without any other grammar features that could be avoided. It is hoped that the RAG model will provide a foundation for a formal theory of abstraction, and the pursuit of this possibility is the author's highest long-term research priority.

Note that the basic well-behavedness criterion for rule functions —answer-encapsulation— was developed with this long-term goal specifically in mind. In a full theory of abstraction, answers must be able to represent the run-time behavior of executable entities, such as, for example, subroutines; without answer-encapsulation, any attempt to provide implementation-independence could be undermined at the level of the underlying grammar.

[3]The meaning of the word "abstraction" is a subject worth several pages of discussion in its own right. This thesis is not the place for such an extensive —and borderline philosophical— exposition. However, see the remarks under [Webs 50] in the bibliography.

Bibliography

[ADJ 79] J. A. Goguen, J. W. Thatcher, and E. G. Wagner, "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types", chapter 5 of Raymond T. Yeh, editor, *Current Trends in Programming Methodology* volume IV [*Data Structuring*], Englewood Cliffs: Prentice-Hall, 1979, pp. 80–149.

Several key mathematical definitions in this paper are stated ambiguously, which makes it a treacherous source of information.

The ADJ group was a team at IBM in the 1970s that investigated the application of abstract algebra to program semantics.

[Aho 68] Alfred V. Aho, "Indexed Grammars—An extension of Context-Free Grammars", *Journal of the ACM* 15 no. 4 (October 1968), pp. 647–671.

[Aho 86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques and Tools*, Reading, Massachusetts: Addison-Wesley, 1986.

This is an update of the classic text on the subject. A useful reference, but dry reading.

[Burs 90] Boris Burshteyn, "Generation and Recognition of Formal Languages by Modifiable Grammars", *SIGPLAN Notices* 25 no. 12 (December 1990), pp. 45–53.

This paper defines Burshteyn's Modifiable Grammar formalism. A correction to one of the definitions appears in [Burs 92].

[Burs 92] Boris Burshteyn, "USSA — Universal Syntax and Semantics Analyzer", *SIGPLAN Notices* 27 no. 1 (January 1992), pp. 42–60.

Burshteyn's metacompiler language, ostensibly based on his Modifiable Grammar formalism. Includes a correction to [Burs 90], and suggests a generalization to multipass parsing.

[Cara 63] Alfonso Caracciolo di Forino, "Some Remarks on the Syntax of Symbolic Programming Languages", *Communications of the ACM* 6 no. 8 (August 1963), pp. 456–460.

A lucid articulation and advocacy of the principle of grammar adaptability. Well thought out, and still worth reading.

[Chom 56] Noam Chomsky, "Three Models for the Description of Language", *IRE Transactions on Information Theory* IT-2 no. 3 (September 1956) [*1956 Symposium on Information Theory*], pp. 113–124.

[Chom 59] Noam Chomsky, "On Certain Formal Properties of Grammars", *Information and Computation* 2 no. 2 (May 1958), pp. 137–167.

[Chri 69] Carlos Christensen and Christopher J. Shaw, editors, *Proceedings of the Extensible Languages Symposium*, Boston, Massachusetts, May 13, 1969 [*SIGPLAN Notices* 4 no. 8 (August 1969)].

[Chri 85] Henning Christiansen, "Syntax, semantics, and implementation strategies for programming languages with powerful abstraction mechanisms", *Proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences* volume 2: *Software* (1985), pp. 57–66.

Also appears as: *datalogiske skrifter* no. 1, 1985.

The original paper on Christiansen grammars. An oversight in the treatment of recursive structures is discussed in [Chri 88a].

[Chri 86] Henning Christiansen, "Parsing and compilation of generative languages", *datalogiske skrifter* no. 3, 1986.

An abridged version appears as: Henning Christiansen, "Recognition of generative languages", in H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects* [*Proceedings of a Workshop*, Copenhagen, Denmark, October 17-19, 1985] [*Lecture Notes in Computer Science* 217], New York: Springer-Verlag, 1986, pp. 63-81.

[Chri 88a] Henning Christiansen, "The syntax and semantics of extensible programming languages", *datalogiske skrifter* no. 14, 1988.

Much more developed than the original paper, [Chri 85]. Uses the AG-based notation for Christiansen grammars, but a section re logic programming foreshadows the later development of the DCG-based notation.

[Chri 88b] Henning Christiansen, "Programming as language development", *datalogiske skrifter* no. 15, Roskilde University Centre, February 1988.

Christiansen's licentiat thesis is made up of this and six other separate papers, including [Chri 85], [Chri 86], and [Chri 88a]. This is the one that ties all of them together.

[Chri 90] Henning Christiansen, "A Survey of Adaptable Grammars", *SIGPLAN Notices* 25 no. 11 (November 1990), pp. 35–44.

Papers on adaptable grammars tend to be highly insular, rarely mentioning other work on the subject. This paper is an exception. It provides a refreshingly broad perspective on the subject, including a thoughtful (and candid) assessment of the formalisms surveyed, including Christiansen's own.

[Chri 92a] Henning Christiansen, "Models and resolution principles for logical meta-programming languages", *INRIA* report no. 1594, Institut National de Recherche en Informatique et en Automatique, February 1992.

See remarks for [Chri 92b].

[Chri 92b] Henning Christiansen, "A complete resolution method for logical meta-programming languages", in Alberto Pettorossi, editor, *Meta-Programming in Logic: Third International Workshop* [*Proceedings of META-92*, Uppsala, Sweden, June 10–12, 1992] [*Lecture Notes in Computer Science* 649], New York: Springer-Verlag, 1992, pp. 205–219.

This work can be viewed in terms of meta-programs or adaptable grammars, depending mostly on whether one uses the word "program" or "grammar"; the content is the same either way.

Christiansen has developed a very elegant generalized resolution method for logic meta-programming (roughly, automated proof of general statements about the possibility of proving statements under different sets of axioms).

[Clea 76] J. Craig Cleaveland and Robert C. Uzgalis, *Grammars for Programming Languages* [*Programming languages series* 4], New York: American Elsevier Publishing Company, 1976.

The focus of the book is on W-grammars. The presentation is extremely lucid, which is remarkable for a treatment of W-grammars, especially when burdened with the old terminology. The treatment of the Chomsky hierarchy is also well done.

[Cohe 88] Jacques Cohen, "A View of the Origins and Development of Prolog", *Communications of the ACM* 31 no. 1 (January 1988), pp. 26–36.

[Colm 75] A. Colmerauer, "Les grammaires de metamorphose", Groupe d'Intelligence Artificielle, Université de Marseilles-Luminy, November 1975.

Appears as: "Metamorphosis Grammars", in Leonard Bolc, editor, *Natural Language Communication with Computers* [*Lecture Notes in Computer Science* 63], New York: Springer-Verlag, 1978, pp. 133–189.

The original paper on metamorphosis grammars. Metamorphosis grammars are Prolog-based Chomsky type 0 grammars with attributes. See [Pere 80].

[Dahl 72] O.-J. Dahl and C. A. R. Hoare, "Hierarchical Program Structures", in O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming* [*A.P.I.C. Studies in Data Processing* 8], New York: Academic Press, 1972, pp. 208–220.

The notion of programming as language development is recommended. Incidentally, the programming language used as a framework for this volume is SIMULA.

[Demb 78] Piotr Dembiński and Jan Małuszyński, "Attribute grammars and two-level grammars: A unifying approach", in J. Winkowski, editor, *Mathematical Foundations of Computer Science 1978* [*Proceedings, 7th Symposium*, Zakopane, Poland, September 4-8, 1978] [*Lecture Notes in Computer Science* 64], Berlin: Springer-Verlag, 1978, pp. 143–154.

Proves the equivalence of attribute grammars and W-grammars.

[Dera 85] Pierre Deransart and Jan Małuszyński, "Relating Logic Programs and Attribute Grammars", *Journal of Logic Programming* 2 no. 2 (July 1985), pp. 119–155.

The bottom line of this paper is that logic programs and attribute grammars are the same thing using different notation.

[Dera 88] Pierre Deransart, Martin Jourdan, and Bernard Lorho, *Attribute Grammars: Definitions, Systems and Bibliography* [*Lecture Notes in Computer Science* 323], New York: Springer-Verlag, 1988.

A good reference book on attribute grammars, modulo it is a few years behind the times by now.

[Dera 90] Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications* [*International Conference WAGA*] [*Lecture Notes in Computer Science* 461], New York: Springer-Verlag, 1990.

Includes [Knut 90] and [Meij 90].

[Grät 68] George Grätzer, *Universal Algebra*, Princeton, New Jersey: Van Nostrand, 1968.

A standard mathematics text on one-sorted algebra.

[Hanf 73] K. V. Hanford and C. B. Jones, "Dynamic Syntax: A Concept for the Definition of the Syntax of Programming Languages", *Annual Review in Automatic Programming* 7 no. 2 (1973), pp. 115–142.

[Hopc 79] John E. Hopcroft and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Reading, Massachusetts: Addison-Wesley, 1979.

Concentrates on Chomsky grammars and corresponding automata, in contrast to the grand-tour strategy of [Lewi 81]. Chapter 9 is devoted to the Chomsky hierarchy; unrestricted grammars require only that the left side of each rule be nonempty. Indexed grammars are defined near the end of the last chapter.

[Imme 88] Neil Immerman, "Nondeterministic space is closed under complementation", *SIAM Journal on Computing* 17 no. 5 (October 1988), pp. 935–938.

[Iron 61] Edgar T. Irons, "A syntax-directed compiler for ALGOL 60", *Communications of the ACM* 4 no. 1 (January 1961), pp. 51–55.

Cited by [Knut 90] in crediting Irons for the idea of synthesized attributes.

[Jaco 89] Nathan Jacobson, *Basic Algebra II*, second edition, New York: W. H. Freeman and Company, 1989.

A graduate mathematics textbook, addressing the wide topic of abstract algebra. The first chapter is on category theory, the second on universal algebra. A fairly good treatment of the subject, but like most mathematical texts it relies heavily on examples that non-mathematicians would be unlikely to understand.

[Katz 71] Jacob Katzenelson and Elie Milgrom, "A Short Presentation of the Main Features of AEPL — An Extensible Programming Language", in [Schu 71], pp. 23–25.

[Klin 72] Morris Kline, *Mathematical Thought from Ancient Through Modern Times*, New York: Oxford University Press, 1972.

See remarks under [Wech 92].

[Knut 68] Donald E. Knuth, "Semantics of Context-Free Languages", *Mathematical Systems Theory* 2 no. 2 (June 1968), pp. 127–145.

The original paper on attribute grammars. The test for circularity of AGs is incorrect, and so is equation (2.4). These errors are corrected in [Knut 71]. (Regrettably, modern authors often cite the original paper without mentioning the correction; this is rather like passing about an invalid pointer value in a C program. It could lead to serious problems later, by which time the cause of the error may be devilishly hard to find.)

[Knut 71] Donald E. Knuth, "Semantics of Context-Free Languages: Correction", *Mathematical Systems Theory* 5 no. 1 (March 1971), pp. 95–96.

Corrects the circularity test of [Knut 68], and incidentally also equation (2.4). The earlier circularity test was too strong. AGs that satisfy that test are called *strongly non-circular*; see [Dera 88]. The general AG circularity problem has intrinsically exponential time complexity.

[Knut 90] Donald E. Knuth, "The Genesis of Attribute Grammars", in [Dera 90], pp. 1–12.

Knuth reconstructs both the historical and the intellectual context from which attribute grammars developed. Worth reading.

[Kost 71] C. H. A. Koster, "Affix grammars", in J. E. L. Peck, editor, *ALGOL 68 Implementation* [*Proceedings of the IFIP Working Conference on ALGOL 68 Implementation*, Munich, July 20-24, 1970], Amsterdam: North-Holland, 1971, pp. 358–373.

The original formalization of affix grammars. Modern references are [Kost 91] and [Meij 90].

[Kost 91] C. H. A. Koster, "Affix Grammars for Programming Languages", in Henk Alblas and Bořivoj Melichar, editors, *Attribute Grammars, Applications and Systems* [*International Summer School SAGA*, Prague, Czechoslovakia, June 4–13, 1991] [*Lecture Notes in Computer Science* 545], New York: Springer-Verlag, 1991.

Koster lumps attribute, affix, and W-grammars all into the class of "two-level grammars". In this paper, he underlines the point by applying each of three formalisms to the definition of a toy language. The three definitions are dramatically alike.

Koster was one of the signators of the Revised Report on ALGOL 68.

[Lewi 81] Harry R. Lewis and Christos H. Papadimitriou, *Elements of the Theory of Computation* [*Prentice-Hall Software Series*], Englewood Cliffs: Prentice-Hall, 1981.

Covers a much wider variety of computational models than [Hopc 79]. The material on Chomsky grammars is scattered throughout the text. Unrestricted Chomsky grammars are defined on page 255, with the requirement that there be a nonterminal on the left side of every rule.

[MacL 71] Saunders Mac Lane, *Categories for the Working Mathematician* [*Graduate Texts in Mathematics* 5], New York: Springer-Verlag, 1971.

This is *the* authoritative text on category theory, the bible of the subject. Well crafted and comprehensive. Any non-mathematician who really wants to be able to understand the examples should look at [MacL 88] first. Even without such background, though, [Pier 91] advises: "Mac Lane's writing is sufficiently lucid that following along at 10% comprehension can be as valuable as checking every detail of another book. His volume belongs on the bookshelf of every serious student of the field."

[MacL 88] Saunders Mac Lane and Garrett Birkhoff, *Algebra*, third edition, New York: Chelsea, 1988.

A truly superior text. Abstract algebra from the ground up. Most abstract mathematics literature (such as [Grät 68] and [MacL 71]) expects the reader to be already familiar with such esoterica as modules over a principal ideal domain. This book not only avoids that problem, but cures it. Armed with nothing more than high school algebra, the novice reader may acquire a rather thorough grasp of the subject by reading Mac Lane & Birkhoff (section by section, from the beginning).

[Mads 80] Ole Lehrmann Madsen, "On Defining Semantics by Means of Extended Attribute Grammars", in Neil D. Jones, editor, *Semantics-Directed Compiler Generation* [*Proceedings of a Workshop*, Aarhus, Denmark, January, 1980] [*Lecture Notes in Computer Science* 94], New York: Springer-Verlag, 1980, pp. 259–299.

See also [Watt 77].

[Maso 84] K. P. Mason, *Dynamic Template Translators: A Useful Model for the Definition of Programming Languages*, Ph.D. thesis, University of Adelaide, Australia, 1984.

[Maso 87] K. P. Mason, "Dynamic Template Translators—A New Device for Specifying Programming Languages", *International Journal of Computer Mathematics* 22 nos. 3+4 (1987), pp. 199–212.

Anyone planning to read this paper should be warned that Mason's style is rather opaque, sometimes bordering on the obtuse. Also, a number of significant details of the DTT formalism are missing or incomplete.

[Maye 72] O. Mayer, "Some Restrictive Devices for Context-Free Grammars", *Information and Control* 20 no. 1 (February 1972), pp. 69–92.

Over half a dozen different grammar models are defined, with several variants for each model, and a daunting battery of equivalences and containment relations are proven between language classes.

[Meek 90] Brian Meek, "The Static Semantics File", *SIGPLAN Notices* 25 no. 4 (April 1990), pp. 33–42.

Despite a disclaimer about not having made up his mind, Meek exhibits considerable religious fervor to the effect that the use of the term "static semantics" is not only incorrect, but actively corrupts those who hear it. He is a devout W-grammarist.

The introduction to the paper invites letters to the editor, but no significant response was forthcoming. Two letters did appear, one in July (from David Gries, voicing support for Meek's condemnation of the term), and one in December (having nothing to do with the topic of the article).

[Meij 90] Hans Meijer, "The project on *Extended Attribute Grammars* at Nijmegen", in [Dera 90], pp. 130–142.

[Naur 60] Peter Naur, editor, "Report on the Algorithmic Language ALGOL 60", *Communications of the ACM* 3 no. 5 (May 1960), pp. 299–314.

[Naur 63] Peter Naur, editor, "Revised Report on the Algorithmic Language ALGOL 60", *Communications of the ACM* 6 no. 1 (January 1963), pp. 1–17.

[Pere 80] Fernando C. N. Pereira and David H. D. Warren, "Definite Clause Grammars for Language Analysis—A Survey of the Formalism and a Comparison with Augmented Transition Networks", *Artificial Intelligence* 13 no. 3 (May 1980), pp. 231–278.

Definite clause grammars are Prolog-based attribute grammars, arrived at the hard way as a special case of metamorphosis grammars. See [Colm 75].

[Pier 91] Benjamin C. Pierce, *Basic Category Theory for Computer Scientists* [*Foundations of Computing* series], Cambridge, Massachusetts: MIT Press, 1991.

A short overview of the subject. Has an extensive annotated bibliography.

[Robe 91] George H. Roberts, "A Note on Modifiable Grammars", *SIGPLAN Notices* 26 no. 3 (March 1991), p. 18.

[Roge 67] Hartley Rogers, Jr., *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, 1967.

Republished by the MIT Press, Cambridge, Massachusetts, 1987.

The bible of recursive function theory. There is no significant difference between the 1987 edition and the 1967 original, but for a few corrections.

[Roze 80] G. Rozenberg and D. Wood, "Context-Free Grammars With Selective Rewriting", *Acta Informatica* 13 no. 3 (March 1980), pp. 257–268.

[Schm 86] David A. Schmidt, *Denotational Semantics: A Methodology for Language Development*, Boston: Allyn and Bacon, 1986.

[Schu 71] Stephen A. Schuman, *Proceedings of the International Symposium on Extensible Languages*, Grenoble, France, September 6–8, 1971 [*SIGPLAN Notices* 6 no. 12 (December 1971)].

[Shut 87] John N. Shutt, "A SOCOL Compiler", Major Qualifying Report, Worcester Polytechnic Institute, Worcester, Massachusetts, May 7, 1987.

[Shut 93] John N. Shutt, "Recursive Adaptable Grammars", M.S. Thesis, Computer Science Department, Worcester Polytechnic Institute, Worcester Massachusetts, 1993.

[Stan 75] Thomas A. Standish, "Extensibility in Programming Language Design", *SIGPLAN Notices* 10 no. 7 (July 1975), pp. 18–21.

A retrospective survey of the subject, somewhat in the nature of a postmortem.

[Uhl 82] J. Uhl, S. Drossopoulos, G. Persch, G. Goos, M. Dausmann, G. Winterstein, and W. Kirchgässner, *An Attribute Grammar for the Semantic Analysis of Ada* [*Lecture Notes in Computer Science* 139], New York: Springer-Verlag, 1982.

[Vogt 89] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper, "Higher Order Attribute Grammars", *SIGPLAN Notices* 24 no. 7 (July 1989) [*Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, Oregon, June 21–23, 1989], pp. 131–145.

The original paper on higher order attribute grammars.

[Watt 74] David A. Watt, *Analysis-Oriented Two-Level Grammars*, Ph.D. thesis, University of Glasgow, January 1974.

Introduces Extended Affix Grammars. Compare [Watt 77].

[Watt 77] David A. Watt and Ole Lehrmann Madsen, "Extended attribute grammars", University of Glasgow, Report no. 10, July 1977. (revised version: Computer Science Department, Aarhus University, DAIMI PB-105, November 1979.)

The original paper on extended attribute grammars.

[Webs 50] *Webster's New International Dictionary of the English Language*, second edition, unabridged, 1950.

The authority of Webster's Second on American English is comparable to that of the (original) Oxford English Dictionary on the English variety of the language. Note that Webster's Third is, to say the least, not a worthy successor. Would you trust a dictionary that defines "infer" and "imply" as synonyms?

The definition of *abstraction* in its metaphysical sense is the most lucid and complete explanation of the concept I have encountered. Consultation with personnel at Merriam-Webster has established that this definition dates, with minor rewording, back to Noah Webster's 1828 *American Dictionary of the English Language*. Internal evidence in the text of the definition suggests direct study of the works of John Locke and (possibly) Aristotle.

[Wech 92] Wolfgang Wechler, *Universal Algebra for Computer Scientists* [*EATCS Monographs on Theoretical Computer Science* 25], New York: Springer-Verlag, 1992.

An invaluable collection of mathematical material relevant to computer science. Topics include reduction systems, universal algebra, and algebraic semantics, together with a great deal of basic mathematical machinery that is brought to bear on those areas.

Despite its considerable merits, this book also provides a good illustration of what is wrong with modern abstract algebra. Wechler tries to cover everything that could possibly be of interest related to each topic,

and consequently, anyone looking for focused information for a specific purpose may have difficulty sorting it out from amidst the 99.9% of the material in the book that isn't needed. Abstract algebra has become a morass of specialized definitions and terminology; for perspective on the sorry state of the field, see [Klin 72], chapter 49 (especially sections 1 and 6), and chapter 13 section 3.

Wechler died less than a month after writing the preface.

[Wegb 70] Ben Wegbreit, *Studies in Extensible Programming Languages*, ESD-TR-70-297, Harvard University, Cambridge, Massachusetts, May 1970.

Reprinted as: Ben Wegbreit, *Studies in Extensible Programming Languages* [*Outstanding Dissertations in the Computer Sciences*], New York: Garland Publishing, Inc., 1980.

[Wijn 65] Aad van Wijngaarden, "Orthogonal design and description of a formal language", Technical Report MR 76, Mathematisch Centrum, Amsterdam, 1965.

The original paper on W-grammars.

[Wino 79] Terry Winograd, "Beyond Programming Languages", *Communications of the ACM* 22 no. 7 (July 1979), pp. 391–401.