UNIVERSITY OF SOUTH ALABAMA
SCHOOL OF COMPUTER & INFORMATION SCIENCES

ATTRIBUTED PARSING EXPRESSION GRAMMARS

By

David Boyd Mercer

A Thesis

Submitted to the Graduate Faculty of the
University of South Alabama
in partial fulfillment of the
requirement for the degree of

Master of Science

in

Computer Science

May 2008

Approved: Date:

Chair of Thesis Committee: Dr. Thomas F. Hain

Member of Committee: Dr. David Langan

Member of Committee: Dr. Tom Johnsten

Member of Committee: Dr. Stephen Brick

Chair of Department: Dr. Michael Doran

Director of Graduate Studies: Dr. Roy Daigle

ATTRIBUTED PARSING EXPRESSION GRAMMARS

A Thesis

Submitted to the Graduate Faculty of the
University of South Alabama
in partial fulfillment of the
requirements for the degree of

Master of Science

in

Computer Science

by
David Boyd Mercer
S.B., Massachusetts Institute of Technology, 1992

Copyright ©2008
All rights reserved.

To the glory of God.

ACKNOWLEDGEMENTS

When I returned to school in 2004, I was eager to learn about many of the advances in Computer Science since I had graduated twelve years previously. I had the good fortune of having Dr. Hain as a teacher in my first semester of graduate school. Coincidentally, he was originally from Melbourne, Australia, the city of my birth, and his father was an accounting professor at the University of Melbourne at the same time that my own father was a student there, studying Accounting. In that first semester Algorithms class, we were required to write a final paper. I recall that Dr. Hain rejected my first proposal for the subject of my final paper as being too easy, so to avoid the humiliation of being rejected again, I proposed a subject that built on some of his own prior work. At the end of the semester, Dr. Hain was so pleased with my paper that he suggested that I try to get it published. It had never occurred to me that I might write something worthy of publication, but with Dr. Hain's assistance, our paper was published the following year at the 2005 Conference on Algorithmic Mathematics and Computer Science. Although that class was the only class I ever took of his, we have remained friends ever since, and he has provided much counsel, wisdom, and encouragement to me during my years in the program.

Dr. Langan, too, deserves a lot of credit for this work, as he was the inspiration for its subject. I was struggling with the difficulty of describing some of the more complex data formats used in healthcare electronic data exchange, when Dr. Langan mentioned that context-free grammars may be enhanced by the addition of attributes. It seems so obvious to me now, as it would to anyone who has studied grammar theory, but this lit a light bulb in my head that perhaps this might be a promising path of investigation.

I would like to acknowledge the patience of Drs. Hain and Langan, and the rest of my thesis committee, which remained interested in my work as the years dragged on. They have reviewed revision after revision of this work, providing very valuable input along the way. They have probably read every word in this work a half dozen times, and their attention to detail has been most valuable.

Most of all, I would like to acknowledge the contribution of my wife, who provided me with encouragement throughout this endeavor. Without her support and—most importantly—love, this work would be little more than a resounding gong or a clanging cymbal.

March 5, 2008

TABLE OF CONTENTS

Page

LIST OF TABLES vii

LIST OF FIGURES viii

ABSTRACT ix

1 INTRODUCTION 1

1.1 Motivation 3

1.2 Outline 4

2 CONCEPTS AND DEFINITIONS 5

2.1 Parsing 12

3 RESEARCH ROADMAP 21

3.1 Research Plan 21

3.2 Evaluation Criteria 25

3.3 Existing Models 30

4 ATTRIBUTED PARSING EXPRESSION GRAMMARS 49

4.1 Parsing Expression Grammars 50

4.2 Attributed Grammars 59

4.3 APEGs Defined 61

5 EVALUATION OF APEGS **74**

5.1 Generality 74

5.2 Form of Output 88

5.3 Time and Efficiency 89

5.4 Memory Usage 90

5.5 Fault Tolerance 91

5.6 Ease of Use 91

6 CONCLUSIONS **97**

6.1 Suggestions For Further Research 98

6.2 Suggestions For Related Research 98

REFERENCES **100**

BIOGRAPHICAL SKETCH **102**

LIST OF TABLES

2.1 Examples of Specific Data Description Languages 20

4.1 PEG Parsing Function Examples 58

LIST OF FIGURES

1.1 Possible Use of a UDDL in Healthcare EDI 4

2.1 A Simple Phrase Structure Grammar 10

2.2 Example of a Parse Tree 13

2.3 A Portion of a Parse Tree, Showing the Lexical Analysis 15

2.4 Levels in Making Sense of Data 16

2.5 Different Grammars that Describe the Same Language 17

3.1 Grammar-Dependent Software Models 22

4.1 A Parsing Expression Grammar Produces an Unexpected Result 56

5.1 A Possible Concrete Syntax for the X12 Interchange Header Segment 96

ABSTRACT

Mercer, David Boyd, M.S., University of South Alabama, May 2008, Attributed Parsing Expression Grammars. Chair of Committee: Dr. Thomas F. Hain

Data description is usually written into parsing program code, resulting in significant expense of developing a new parsing routine when a new data format is required. A data description language provides a way to describe data formats so that only a single parser is needed in order to parse data formats describable by the data description language. The need for a universal data description language is described along with a research plan for finding an acceptable one. A Turing-complete grammar model based on the combination of parsing expression grammars with attribute grammars is defined as a possible candidate grammar to form the base of a universal data description language.

CHAPTER 1

INTRODUCTION

When Congress passed the Health Insurance Portability and Accountability Act (HIPAA) of 1996, it mandated the adoption of "standards for transactions, and data elements for such transactions, to enable health information to be exchanged electronically." [§1173(a)(1)]. This was in reaction to the wide variety of proprietary standards and formats used by healthcare payers and providers for the exchange of information electronically. In order for a provider of healthcare services to exchange information with a payer electronically (e.g., filing a claim electronically with the payer, receiving individual insurance coverage information from the payer, etc.), the provider had to implement different electronic data interchange (EDI) systems for each individual payer. The cost of developing such systems was prohibitive for the many hundreds of payers a provider might do business with, so larger providers would implement EDI systems for only a couple of their highest volume payers, and would conduct most of their business with insurance companies by paper and the post. For smaller providers—small hospitals and professional physicians' groups—the cost of implementing even one EDI system could not be justified, so all business was conducted manually. This state of affairs was costly for both payers and providers.

HIPAA should have simplified EDI in healthcare, for it would eliminate the variety of data formats in use, permitting providers (and payers) to implement only one EDI system to support EDI with their partners. In fact, it had the opposite effect. There were four main problems:

1. Payers desupported old formats in order to comply with the law (which prohibited use of other formats), and providers who had developed translators for their payers found themselves having to rewrite them.
2. The transaction sets defined by X12 and selected by the Department of Health and Human Services (DHHS) as the mandated standard are written in English, which when implemented, resulted in a slightly different interpretation of the standards by each payer. Furthermore, the standards themselves permitted a certain amount of ambiguity, encouraging different payer-specific dialects.
3. Some payers chose to implement only a subset of the standards, which, although technically a violation of the law, can be done with impunity because the law provides for no enforcement or penalties for violation of these standards. Other payers took a more liberal view of the law and came up with their own standards, related to the official standards only in name and a written explanation of how to map some of the fields in the official format to their own proprietary one.
4. The law specified no requirements on the physical connection and the communication protocol to be used, resulting in every payer coming up with its own

(and sometimes arcane) connection methods, communication protocols, and data encodings.

It is interesting to note that compliance with the new standards was so onerous that DHHS had to extend the original deadline for compliance by a full year due to the difficulties of payers and providers in implementing the standards. Even after the deadline extension, the Centers for Medicare and Medicaid Services (CMS), the U.S. federal agency which administers Medicare and Medicaid, was unable to comply with the HIPAA-mandated standard for eligibility inquiries until a full twenty-two months after the final deadline had passed.

1.1 Motivation

A universal data description language (UDDL) would provide a language that could be used to describe any data format. While further definition of the terms is required, a UDDL would permit payers to publish their data formats, and providers to implement a single EDI system that could read the each payer's definition and communicate in each payer's individual format, as illustrated in Figure 1.1. While this would not have solved problem 4 above, it would certainly have alleviated problems 1–3. Indeed, it is conceivable that had such a UDDL existed in 1996, there would have been no need for Part C (Administrative Simplification) of HIPAA, as each payer's proprietary format could also have been published in UDDL, or perhaps the law could have focused on solving problem 4, which remains to this day, and is beyond the scope of UDDL. The cost savings would have been tremendous.

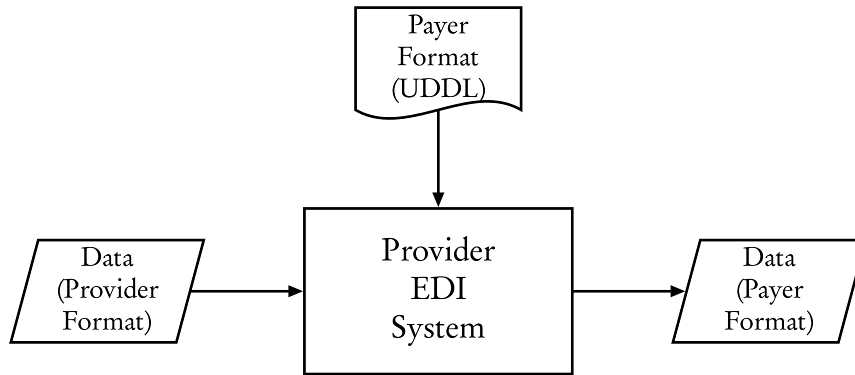

Figure 1.1. Possible Use of a UDDL in Healthcare EDI

1.2 Outline

The next chapter of this work will lay the groundwork for this research by describing the concepts and defining the terminology used in this discipline. Chapter 3 describes current state of knowledge and presents a roadmap for research in this area. Chapter 4 defines attributed parsing expression grammars, while Chapter 5 evaluates the new grammar model with respect to its suitability as a basis for a UDDL. Finally, Chapter 6 presents conclusions and suggestions for further work.

CHAPTER 2

CONCEPTS AND DEFINITIONS

Information in a computer is stored in only one dimension. In order to store multi-dimensional information, it must be mapped to a single dimension in a process called *serialization*. This mapping is not always difficult. For instance, a table with ten rows and ten columns can easily be mapped to a linear array of one hundred elements: the first row is stored in the first ten elements (1–10), the second row is stored in the next ten elements (11–20), and so on until the last row of the table is stored in the last ten elements (91–100) of the hundred-element array. More complicated information (hierarchical trees, for instance), while more difficult to map manually, can be mapped to linear storage quite easily by a computer. This research is directed toward the reverse process, called *parsing*: unmapping a one-dimensional information stream to its original shape.

In the context of this work, *data* is the representation of information in such a way as to be usable by a computer. Because of the serial nature of communication and storage media, we shall assume that data is represented in a one-dimensional *stream* of bits. This is similar to natural languages (such as the language of this manuscript), which reduce complex information structures into linear streams of letters

and words, and punctuation. In this research, I shall consider only one-way streams, particularly read-only streams, in the understanding of data and data formats. A two-way "conversation" on a stream will not be considered, although such a dialogue can be broken into separate monologues, and each incoming monologue can be considered its own data stream. We shall also conveniently choose to ignore the case of data being spread across multiple data streams. Although this is a very real situation (e.g., an order file with customer number references to customer information stored in a separate file), the problem can be solved by considering the multiple streams in series, thus producing a single stream. In the end, however, we might prefer to have a data description language that permits data languages spread over multiple streams. This should be a criterion in deciding between different data description languages.

The analogy to natural languages proves quite helpful, and, in fact, we shall formally define languages in such a way to include both natural languages and computer languages, such as programming languages and data formats.

Definition 2.1. An *alphabet* is a finite set of atomic symbols.

In common usage, we typically envision an alphabet as the letters `A` through `Z`, but we may also choose to add to our alphabet symbols for lowercase letters, punctuation, spaces between words, etc. Furthermore, depending on our level of language definition, our alphabet may be considered words rather than letters, ranging, for instance, from "`aalii`" to "`zyzzogeton`". This does not violate our requirement of

atomicity, since combining the symbols "`there`" and "`fore`" does not produce the symbol, "`therefore`," but rather the sequence of words "`there fore`".

Throughout this document, we shall use the capital Greek letter sigma, Σ, to represent an alphabet. The binary alphabet used by computers is $\Sigma = \{0, 1\}$.

Although strings are often defined simply as a sequence of symbols from an alphabet, such a definition does not define any properties or operations on strings. To avoid having to laboriously define what is meant by the length of a string or the concatenation of two strings, we use more familiar mathematical constructs in our definition.

Definition 2.2. A *string* s over an alphabet Σ is a total function mapping the nonnegative integers less than l into symbols in Σ, for some nonnegative integer l. We call l the length of the string, and write this "$|s| = l$."

We typically write the value of a string by listing its range in sequential order. Thus, if $s = \{[0, a], [1, b], [2, c]\}$, then we could write "$s = abc$." Special notation is sometimes required to represent a zero-length string, also called the *empty string* or *null string*. Although we could (and possibly should) use the empty set symbol ($\emptyset$), most literature in language theory utilizes a lowercase lambda (λ) or epsilon (ε). The latter is becoming more popular so as to avoid confusion with the λ used in λ-calculus, so we shall use ε to represent the empty string. Note that we are not introducing a new symbol into our alphabet, but rather providing ourselves with a way of

representing the empty string. Without such notation, the empty string, $s = \varepsilon$, would be written "$s =$".

The set of all strings (including the empty string) over an alphabet Σ is written Σ^*.

Definition 2.3. The *concatenation* of two strings, s_1 and s_2, produces a single string, $s_1 s_2$.

$$\text{concat} : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$$

$$\text{concat}(s_1, s_2) = s_1 s_2 = s_1 \cup \{[n,a] \mid [n - |s_1|, a] \in s_2\}$$

It follows from this definition that

$$(s_1 s_2)(n) = \begin{cases} s_1(n) & \text{if } n < |s_1| \\ s_2(n - |s_1|) & \text{if } n \geq |s_1| \end{cases}$$

The reverse of concatenation is partitioning.

Definition 2.4. If $s = s_1 s_2$, then s_1 is a *left partition* and s_2 is a *right partition* of s.

$$\text{part} : \Sigma^* \rightarrow \Sigma^* \times \Sigma^*$$

$$\text{part}_\text{L}(s, l) = \{[n,a] \mid [n,a] \in s \ \wedge \ n < l\}$$

$$\text{part}_\text{R}(s, l) = \{[n,a] \mid [n+l,a] \in s \ \wedge \ n \in \mathbb{N}\}$$

Definition 2.5. A *language* is a set of strings from a finite alphabet.

$$L \subseteq \Sigma^*$$

English, for example, is a language: some strings, such as, "The cat sat on the mat," are in the English language, while others, such as "The on mat cat sat the," are not. English is an example of a natural language. A language designed specifically for conveying computerized data is called a *data language*.

Any language of practical interest is not just an arbitrary set of strings, but rather a set of strings that follow certain rules (*syntax*) and convey meaning (*semantics*). A language *acceptor* is a machine that is able to determine whether a given string is in the language or not—no small task for some languages, especially natural languages, which often have very complex rules regarding what constitutes a string in the language.

A *grammar* is a formal way to define a language precisely. Most grammars are *generative grammars*, which define how to construct every string in the language; a string is in the language defined by a generative grammar if and only if the string can be constructed by following the rules of the grammar. An alternative to generative grammars is *analytic grammars*, which define how to analyze an input string to determine whether the string is in the language. Despite the close association of analytic grammars to parsing, the overwhelming majority of research on grammars and parsing has been based on generative grammars. Most (generative) grammar models are based on the phrase-structure grammar model (Chomsky, 1956), which is defined informally as a start symbol and a set of substitution rules. (Phrase-structure grammars are defined formally in Section 3.3.2.3.) Figure 2.1 is an example of a phrase-structure grammar that defines a language of sentences like "The cat sat on the mat."

sentence	→	*capitalized subject predicate period*
subject	→	*noun-phrase*
noun-phrase	→	*article noun*
article	→	`a`
article	→	`the`
noun	→	`cat`
noun	→	`mat`
predicate	→	*verb*
predicate	→	*verb prepositional-phrase*
verb	→	`sat`
verb	→	`lay`
prepositional-phrase	→	*preposition noun-phrase*
preposition	→	`on`
preposition	→	`next to`
period	→	`.`
capitalized `a`	→	`A`
capitalized `the`	→	`The`

(a)

sentence	↦	*capitalized* <u>*subject*</u> *predicate period*
	↦	*capitalized* <u>*noun-phrase*</u> *predicate period*
	↦	*capitalized* <u>*article*</u> *noun predicate period*
	↦	<u>*capitalized* `a`</u> *noun predicate period*
	↦	`A` <u>*noun*</u> *predicate period*
	↦	`A cat` <u>*predicate*</u> *period*
	↦	`A cat` <u>*verb*</u> *period*
	↦	`A cat sat` <u>*period*</u>
	↦	`A cat sat.`

(b)

Figure 2.1. A Simple Phrase Structure Grammar (a) An example grammar that generates sentences like "`The cat sat on the mat.`" and "`A mat lay next to the cat.`" *(b)* An example of its use in generating the sentence "`A cat sat.`". The underlined symbols in each step are the symbols to be substituted in the next step by a rule in the grammar.

It is an unfortunate property of our universe that no grammar model can describe every language. Indeed, there are languages out there that cannot be described by human language. This can be proven.

Theorem 2.6. *There exist languages that cannot be described by human language.*

Proof. Let Σ_{h} be the set of all symbols, letters, characters, glyphs, and ideographs used in all human communication past, present, and future. There is an infinite number of strings that can be produced using this alphabet, though this number could, in theory, be counted, thus $|\Sigma_{\text{h}}^*| = \aleph_0$. Let $\Sigma_L \subseteq \Sigma_{\text{h}}$ be the alphabet from which we shall form languages we wish to describe with our human language. The number of languages that can be formed from the alphabet Σ_L, $\mathscr{P}(\Sigma_L^*)$, is infinite and uncountable: $\mathscr{P}(\Sigma_L^*) = 2^{\aleph_0}$. Since $\aleph_0 < 2^{\aleph_0}$, there are more languages in Σ_L^* than there are possible ways to arrange the symbols of human language. Therefore, even if every possible arrangement of human symbols described a language (which they most certainly do not), there would still be languages that were not described. □

Fortunately, languages that cannot be described by humans are unusual in the realm of electronic data interchange, so we can ignore them. There also exist languages that can be described, but for which membership of a string in the language cannot be always be determined (e.g., the language consisting of a single string of `0`'s whose length is the longest string of consecutive `0`'s in the decimal representation of π, 3.14159...). These languages are also not used in EDI, so luckily we may ignore

them, too. In fact, we may concern ourselves only with the subset of languages for which the membership of a string in the language can definitely be determined one way or another. These languages are called *recursive languages*.

Although we have proven that no grammar model can possibly describe every possible language, not all grammar models are equal. Some grammar models are considered "stronger" than others, meaning they can describe a proper superset of languages than "weaker" models. If two grammar models can describe the same set of languages, they are said to be equally strong (or weak). Of course, this comparison is not always possible, since it is possible for a grammar model to neither be equal to nor a proper subset or superset of another grammar model. In such cases, it cannot be said that one model is stronger than the other.

2.1 Parsing

A language *parser* is a specialized type of acceptor that is also able to provide a syntactic analysis of its input, producing a model of the information contained in the string to facilitate understanding its meaning. Commonly, the information model produced by a parser is a *parse tree* (Figure 2.2) or a sequence of instructions for creating or navigating a parse tree, but there is no requirement that the model be a hierarchical tree structure at all. For purposes of this research, we shall consider an *information model* to be a labeled directed graph. A parse tree, therefore, is just a specialized subset of possible information models.

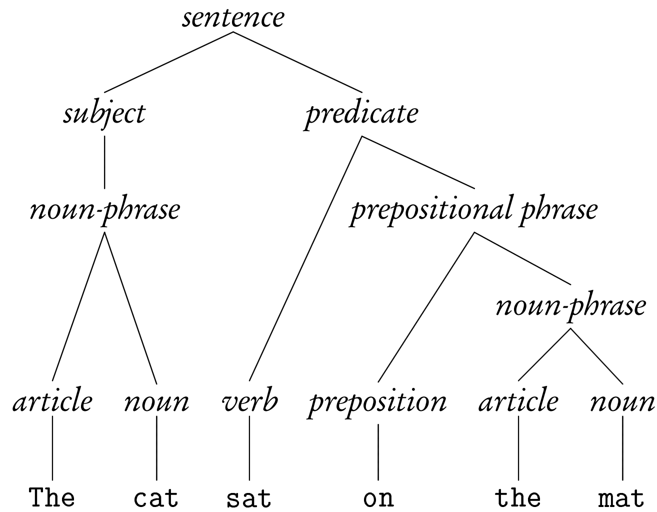

Figure 2.2. Example of a Parse Tree

Parsing has traditionally been considered a two-part process: (1) lexical analysis, and (2) syntactic analysis. *Lexical analysis* reads the units (e.g., characters) of the input and attempts to identify lexemes (e.g., words) from the input. Unimportant characters (e.g., spaces) may be discarded during lexical analysis. *Syntactic analysis* analyzes the lexemes and attempts to demonstrate how the string can be generated from the grammar of the language. The example of Figure 2.2 shows only the syntactic analysis; it was the lexical analysis (not shown) that formed the input into words that could be analyzed syntactically.

A distinction is made between the logical and physical characteristics of data (McGee, 1972). *Logical data characteristics* are those characteristics of the data that are apparent to the user of the data, such as the arrangement of a personnel record with a name and date of birth, and its relationship to departments and organizations. The *physical data characteristics* are those that are not apparent to the user, such as

the character set encoding and, in stored data, the file and directory structure of the database.

Physical data characteristics—and thus physical analysis of the data stream—underlays the lexical analysis, since we can exchange the physical character encoding of data (e.g., between ASCII and EBCDIC) without altering the language's lexical and syntactic definition. Some physical analysis may be performed automatically by the operating system or communication protocol (e.g., the actual mapping of magnetic bits on storage media to bits in the data stream), but there may also be physical analysis performed by the parser. In this research, we are only interested in data description at the application level, which may include some physical analysis of the data stream, although much of the physical analysis will be left to lower levels of the communication protocol and operating system.

There is no clear delineation between those characteristics that are physical and those that are logical, since different users have different interests in the data. Similarly, the boundary between lexical and syntactic is fuzzy and arbitrary, chosen for convenience. The same processes used to form sentences out of words could have been used to form words out of the ones and zeroes of the data stream, as illustrated in Figure 2.3, and some grammar models unify both lexical and syntactic language definition. Exactly where the lines are drawn dividing physical, lexical, and syntactic analysis may be debatable, but the fact that there are multiple levels of analysis in parsing is generally accepted. The question arises as to whether there are only two (lexical and syntactic) or three (physical, lexical, and syntactic), or if there are more.

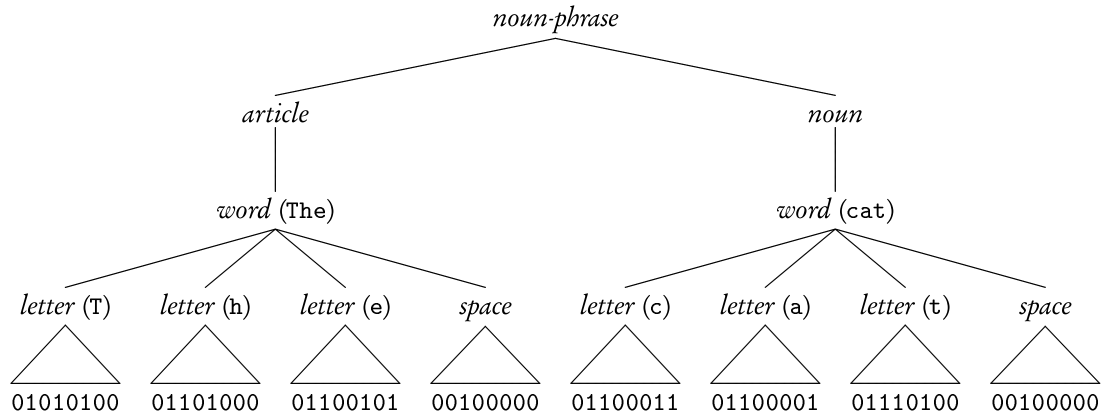

Figure 2.3. A Portion of a Parse Tree, Showing the Lexical Analysis

Probably, there are more. For instance there may be more logical levels above syntactic analysis that handle semantics, or there might be multiple levels of validation. One can imagine in XML, for example, the syntactic analysis parsing an input data stream as a well-formed XML document, but there might be an overlaid level that performs additional validation to ensure that the document adheres to the requirements of a data type definition (DTD) or XML schema. Regardless of the number of conceptual levels involved in converting electrons into meaningful information, our research considers parsing to gulf the vast middle ground between physical analysis, which is usually handled by the operating system, and semantic analysis, which concerns itself with the meaning of the information (Figure 2.4).

Correct parsing requires proper specification of the language's grammar. The same language defined by two different grammars may produce very different information models (Figure 2.5). For example, the language described as "the set of all strings consisting of a sequence of `a`'s followed by a sequence of `b`'s" is the same lan-

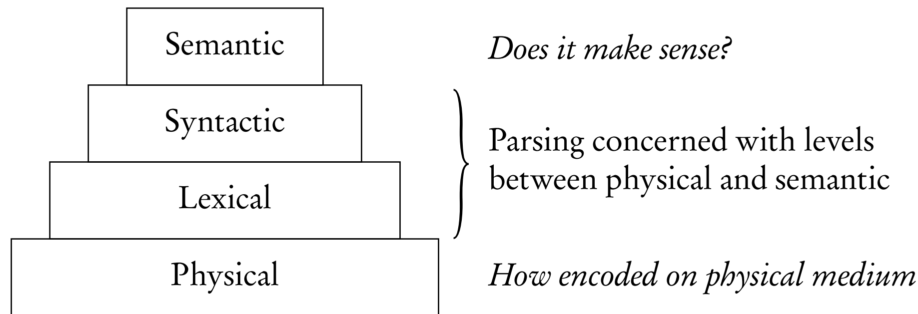

Figure 2.4. Levels in Making Sense of Data

guage as that described as "two equal-length sequences of only `a`'s and only `b`'s respectively, separated by a sequence of any length of either all `a`'s or all `b`'s." However, we would expect the information models generated by the two descriptions to represent the structure differently: in the first case, we would expect the model to represent two sequences, whereas in the second it should represent three. If we consider the first description to be correct (an arbitrary choice for this example, but the choice in real-world applications is not arbitrary), then a model of the information created by the second description is of little use and would be considered incorrect, as it does not correctly reflect the structure of the underlying information.

Some grammar models may even be incapable of producing a "correct" information model for some languages. Most grammar models are incapable of modeling information in anything other than a hierarchical tree. Chomsky's phrase-structure grammars, for example, despite their ability to define every recursively enumerable language (that is, for all practical purposes, a superset of all data languages), are inca-

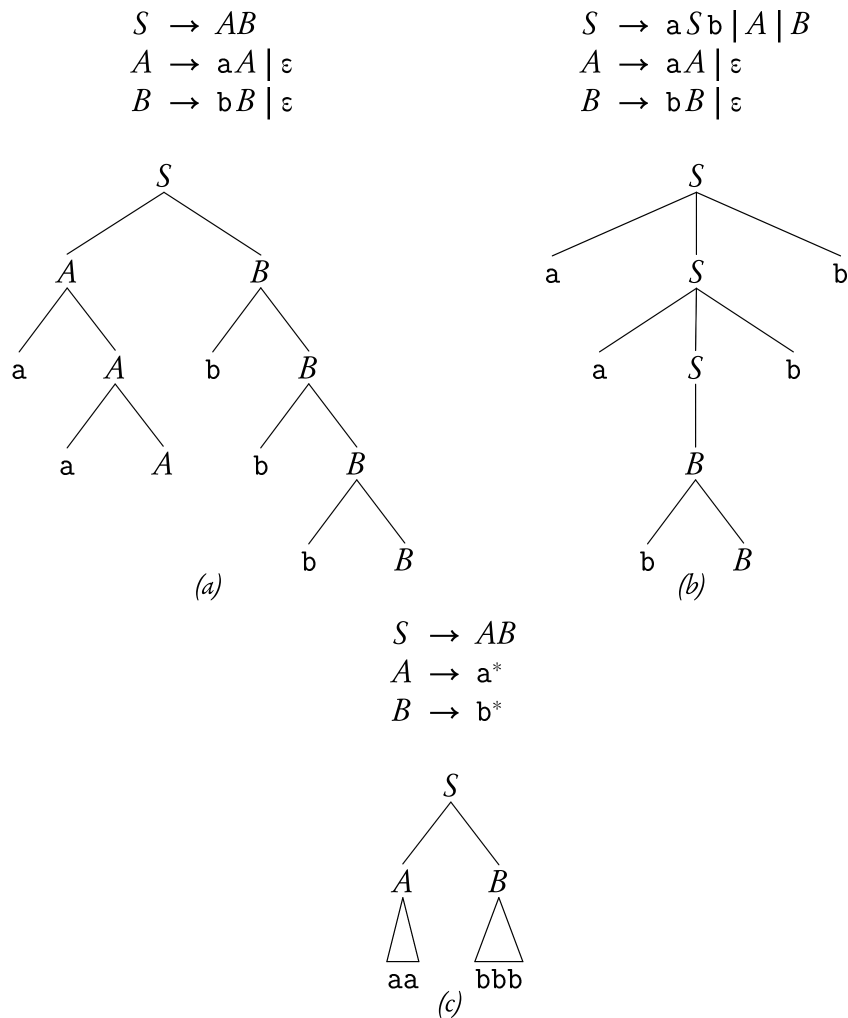

Figure 2.5. Different Grammars that Describe the Same Language. All the grammars in this example describe the same language: a sequence of a's followed by a sequence of b's. Using the (arbitrary) definition of "correct" described in the text, neither of the parse trees created by the phrase-structure grammars *(a)* and *(b)* correctly describes the structure of the language. To create a "correct" parse tree, a regular right part grammar *(c)* must instead be used.

pable of defining sequences as sequences, requiring that such constructs be described implicitly by recursion (Figure 2.5).

The issue of information model "correctness" is an important one in practical applications. Although a language definition may accurately define a particular data language, the definition is of little practical utility (except as a language acceptor) if the resulting model does not correctly describe the structure of the information. Therefore, a universal data description language and its underlying grammar model must be capable not only of accurately describing data languages, but also of correctly describing the structure of the information.

The bits in a stream can be broken down into data elements and structural elements. *Data elements* are those parts of the stream that actually contain information, while *structural elements* delimit or relate data elements. In natural languages, we use punctuation (to delimit) and prepositions (to relate) data elements (nouns and verbs). Data languages often make use of explicit delimiters between data elements, and even in the fixed-width layouts common in COBOL-based systems, we would consider the filling of empty places with spaces to be structural elements. XML makes such frequent use of structural elements that it is not uncommon for the structural elements of an XML document to outweigh the data elements. Structural elements are often discarded during lexical analysis.

In order for a parser to execute its duties, it must contain a model of the language to be parsed, called a *grammar*. A distinction must be made between programs that involve explicit grammars and those that merely encode the grammatical struc-

ture within their program code (Figure 3.1). Both programs are considered *grammar-dependent software*, but programs that utilize explicit grammar definitions are considered *meta-grammarware.*(Klint, Lämmel, & Verhoef, 2005). This distinction is important, as the source of the problems cited in the introduction is that the data language grammars were directly encoded into the parser programs rather than separated into explicit grammar definitions, resulting in great difficulty when the data language changed. It is the purpose of this line of research to define a grammar language that can be used to define data languages, so that meta-grammarware can be created for electronic data interchange.

A *data description language* is a grammar language (a metalanguage—a language for describing languages) for describing data formats so that a parser can be created that will accept both the data stream and the format description, and parse the data stream correctly. Various specific data description languages exist (Table 2.1) and are commonly used, but these DDLs target specific classes of data formats and uses, and so are only helpful when the different data formats are sufficiently similar to be described by the same DDL. A general data description language is intended to describe many significantly different classes of data formats, but a truly universal data description language is a DDL that can be used on all data formats. In order to attain universality of the language, it must somehow be proven that the language can describe every possible data format. This is a high bar to set, but it should not be impossible. Certainly all practical data languages are recursive, so any Turing-complete language

Data Description Language	Specific Application
XML DTD	XML data
EDI Standard Exchange Format	X12 EDI formats
Microsoft's BCP DDL	fixed-width and delimited record-oriented formats
ASN.1	logical data description

Table 2.1. Examples of Specific Data Description Languages

should suffice. The challenge is to create such a language that permits description of data languages in an intuitive way.

CHAPTER 3

RESEARCH ROADMAP

It is the purpose of this line of research to define a grammar language that can be used to define data languages, so that meta-grammarware can be created for electronic data interchange. This vision is illustrated in Figures 3.1(b) and (c).

3.1 Research Plan

We ought not expect that the definition of a universal data description language will be easy. If it were, because of its great importance and utility, it would already have been done. Indeed, it is the daunting size of the task that has restricted existing DDLs to describing only narrow classes of data. We have therefore devised a research plan intended to aid us in our pursuit of this goal. The plan is modeled on the success of XML, which provides a parser (SAX), a document model (DOM), an addressing scheme (XPath), and a transformation language (XSLT).

1. *Identify candidate grammar models.* These grammar models will become the basis for DDLs that will be developed further. Criteria for identifying promising grammar models are described in Section 3.2.

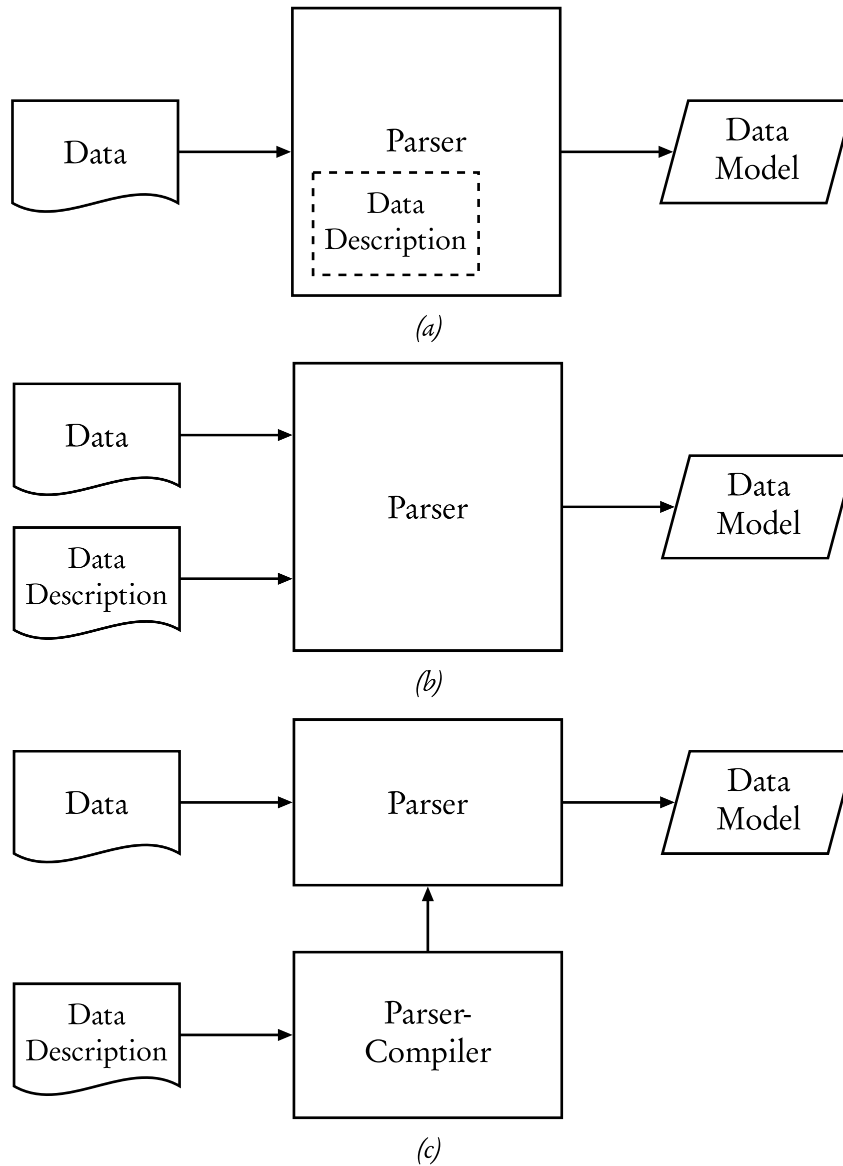

Figure 3.1. Grammar-Dependent Software Models (a) The most common model, in which the grammatical structure is encoded within the program code. *(b)* A more flexible model in which the structure of the input is pulled out into an explicit grammar. *(c)* An alternate model in which the grammar is used to generate a parser specific to the grammar; this is the model used by the popular parsing tools *lex* and *yacc*. Figures (b) and (c) represent meta-grammarware.

2. *Define candidate DDLs.* Based on promising grammars identified in step 1, one or more DDLs are defined based on these grammars and evaluated for practicality. Candidates whose universality has not been proven are to be considered at this stage because a viable universal DDL may not be discovered for some time (or may not exist), and a good-enough candidate now may be better than a perfect candidate later. Step 7 is intended to resolve these issues of universality.
3. *Create parsers for candidate DDLs.* A concrete implementation of a parser based on a candidate DDL must be created to ensure that the DDL is practical.
4. *Create corresponding document models.* Based on the output of the parsers created in step 3, a document model is created to validate the practical utility of the parsers' output. This will frequently be just a representation of the parse tree.
5. *Define a document model addressing scheme.* An addressing scheme for the document model defined in step 4 permits data elements to be addressed and queried from the document model. A usable addressing scheme validates the practicality of the document model. If the document model is simply a tree, this step is trivial. However, for non-tree-based document models, an addressing scheme is desirable to facilitate creation of transformation tools and adoption of the DDL.
6. *Define a transformation language.* One of the goals of data description is to automate translation of data from one representation to another. In order for a

DDL to gain acceptance, it must lend itself to automated parsing and transformation.

7. *Evaluate DDL generality and universality.* If the chosen DDL can be shown to be universal, then we have realized a UDDL. If multiple UDDLs are found, then they may be compared based on aesthetic and other subjective criteria to determine which is the best. Or presuming that they are equivalent, transformations could be defined between the UDDLs, permitting them to be used interchangeably. However, if a candidate DDL was chosen in step 2 that is not proven to be completely universal—that is, there exist (or may exist) data languages that cannot be described by the DDL—then we must make the determination as to whether it is good enough (i.e., whether it only fails in certain pathological cases) or whether certain modifications can be made to the DDL to increase its generality.

Steps 1 and 2 can be seen as generating candidate DDLs, while subsequent steps are evaluation and filter steps, aimed at narrowing the list down to the usable DDLs based on the selection criteria developed in steps 1 and 2. Although the roadmap is sequentially numbered, the order need not be adhered to too rigidly, for there is a certain degree of feedback looping in the process. For example, parser development in step 3 may require refinement to the DDL defined in the previous step; it may be worth performing step 7 earlier in the process; and since there already exist some DDLs, we could use those as a beginning point, skipping steps 1 and 2 altogether.

In an ideal case, however, a single grammar model chosen in step 1 results in the development of a DDL in 2, which subsequently passes the tests of steps 3–7. While this scenario might occur, we must anticipate some failures and setbacks, and looping through the steps. However, these attempts must be documented so that future research and development work on universal DDLs can stand on the discoveries.

3.2 Evaluation Criteria

Inherent in steps 1 and 2 is the definition of selection criteria that would make candidate grammars and DDLs promising. Simply defining a universal DDL is of little use if a UDDL parser cannot be efficiently implemented or if the syntax and semantics of the language are so arcane and complex that describing data formats in the UDDL is overly burdensome. We have therefore laid out the criteria for a practical UDDL in order for this research to be fruitful. The criteria described in the following sections should be considered when selecting and developing UDDL candidates. Admittedly, some of these criteria can only be judged subjectively (e.g., ease of use). Additionally, while many of these criteria are attributes of the implementation of the parser rather than the language, the language does, to a large extent, determine what sort of parsers can be built.

3.2.1 Generality

McGee (1972) lamented some of the limitations of any data description language that might be developed. Firstly, that implementing a DDL that encompassed everything

would come at “an enormous implementation price.” And secondly, that it might be incomplete: that a new structure might arise later that is beyond its capability. Certainly, we would expect that many of the advances in computer hardware in the last thirty-five years would have reduced the impact of this “enormous implementation price,” and it is certainly our goal to develop a data description language that is not prohibitively expensive in computing resources. However, his second point is important, so we must do our due diligence in ascertaining the class of languages our data description language will describe. The language chosen or developed must be general enough to describe any possible data language.

There are various ways to evaluate a grammar against this criterion:

1. Identify the class of languages that UDDL can define, and show that this class of languages encompasses all possible data formats. Although we cannot know for certain what all the possible data languages might be, certainly any language that is not recursive would be a completely impractical data language, so it is reasonable to presume that a grammar model capable of describing any recursive language is sufficiently general.
2. Choose a representative sample of the many widely different data languages and validate that they can be defined by UDDL.
3. Make up some contrived data formats that contain “tricky” constructs, and show that UDDL can define those formats.

While it would be nice to be able to prove the universality of a language using the first approach, unless the language chosen is sufficiently simple, such rigorous

proof may not be possible, requiring us to fall back on the other two (less satisfactory) methods.

It is a necessary attribute of the language that it must be able to describe itself. This is an offshoot of the fact that, once adopted, a data format could be created that consists of a preamble with the UDDL definition of the data, and a main body which contains the data in the format described by the preamble. Therefore, the language must not only be able to describe itself, but it must also be able to use that description to dynamically create a parser for the described language.

3.2.2 Form of Output

The output from a parser based on the language should lend itself to being practical. While this is vague (and subjective), a parser that outputs parsing messages implying a creation or navigation of the parse tree is likely what we expect. With this in mind, top-down parsing is typically more practical than bottom up, since we would like to receive a message saying, "here's the header," followed by a stream of data rather than receiving a stream of data followed by the message, "that was the header." One reason for this is that one of our principles is to limit memory usage, but if we are planning to do something with a particular part of the data stream, then we would have to hold the entire data stream in memory in case we receive a message saying that everything we had just received was the part we were looking for.

The form of the output also refers to the shape of the parse tree produced. Figure 2.5, for example, shows an example where the phrase-structure grammar model is not capable of producing the “correct” parse tree.

3.2.3 Time and Efficiency

The language should enable parsing in reasonable time, ideally in time directly proportional to the length of the input. However, it must be anticipated that some looking-ahead or back-tracking may be necessary, and such algorithms may have poor theoretical worst-cases. However, if the language can be defined so that such looking-ahead and back-tracking is likely not to come anywhere close to the worst-case (e.g., limited look-ahead), then the language may be acceptable.

3.2.4 Memory Usage

Our ideal language is one that permits a directional parser to be built that uses only a constant amount of memory. A directional parser is one that does not require that the entire document be held in memory. This criterion is closely related to time and efficiency, since back-tracking or looking ahead through the entire data document implies that the document should be held in memory. The input to the parser should be seen as a data stream rather than a data file, a data file implying some amount of persistence which would enable unrestricted look-aheads and back-tracking, which is to be avoided. Using a stream-based model, the amount of look-ahead/back-tracking is limited to the size of the look-ahead/back-tracking buffer.

3.2.5 Fault-Tolerance

The language should permit a parser to be created that handles errors in the input stream well. This means that if there is an error in the input stream, the parser should note the error and continue if possible, halting only when there is no possible recovery. There are various use cases: for instance, if a field in a data stream is expected to be a date, and the date parses to February 30, or is all zeroes or all nines (sometimes used to indicate minimum and maximum date values), then the parser should note the error and continue at the next field. This is also important when the input stream is a continuous data stream (e.g., in real-time systems), in which an error in the data stream cannot cause the application to halt.

3.2.6 Ease of Use

A data description language with many arcane and complicated rules regarding its usage, which does not describe data languages intuitively, as a programmer would describe it, which makes it easy to introduce errors and difficult to debug, is unlikely to be accepted by the community of end users. With an acceptable universal data description language, "easy things should be easy and hard things should be possible." (Wall, 1998) In all cases, we would expect fixed-width and delimited record-based layouts to be considered easy, and a UDDL would hopefully make many other classes of data languages "easy".

The "hard things" must also be possible in order to satisfy our requirement that the data description language be universal, but even when such "hard" data languages

are made possible, but not necessarily "easy", this is acceptable because, in practice, a data language is only described once, and so its difficulty can be amortized over a large number of applications. It would be advantageous, however, if the UDDL permitted modular description of data languages, permitting the assumptions of a "hard" class of data languages to be defined only once and then incorporated, imported, or referenced into the description of all languages in that class. The idea here is twofold: that data formats can be built by modifying (i.e., restricting or generalizing) existing definitions; and that data formats can be defined by combining different definition "modules." An example of the former might be XML, which may have a complicated description in our UDDL, but it need be described only once if all XML-based data languages could share the same definition and simply restrict it to their requirements. An example of the latter might be data languages identical to each other except for the character encoding (ASCII vs. EBCDIC) or that one is fixed-width and the other delimited: the core definition of the data language is incorporated in one description, and only the differences from the core need to be written for the variants.

3.3 Existing Models

Having specified the evaluation criteria in Section 3.2 by which we will select our model for further work, we have performed a rigorous search of existing grammar models and data description languages to determine whether any of them satisfy our criteria and might be useful in providing a head-start in our endeavor. Although

no grammar or DDL proved to be satisfactory, an examination of the strengths and weaknesses of each provides us with guidance in developing alternatives.

3.3.1 Data Description Languages

We look first at existing data description languages, because, if such a one exists that satisfies our requirements, it obviates our need to perform roadmap step 1. Presumably, the developers of the DDL have already performed this work.

In 1959, the Department of Defense organized individuals from industry and government involved in data processing activity into the Conference on Data Systems Languages (CODASYL) to guide the development of standard programming languages. Although best known for its development of the programming language COBOL, it organized work in the late 1960's and early 1970's on data definition, beginning with its List Processing Task Force headed by Charles Bachman in 1965. In 1967, the task force renamed itself the Data Base Task Group (DBTG), and in 1969 released specifications for a data description language (DDL) describing the schema of a database. In 1971, the DBTG was split, and data description was assigned to the Data Description Language Committee (DDLC). The work of these CODASYL-affiliated groups developed standards for databases, which was a more immediate problem than that of electronic transactions, which were practically nonexistent at the time.

Similarly, the Association for Computing Machinery (ACM) formed the Special Interest Group on File Description and Translation (SIGFIDET) in 1969. Again,

however, the SIGFIDET's work was geared toward database description rather than EDI. Computer scientists struggled with the specifications of an adequate data definition language. They were able to identify some of the problems, but offer few solutions. McGee (1972) attempted to divide data description between logical and physical data characteristics, but was unable to come up with a solid line delineating the two. He did, however, conclude that logical data description (characteristics of the data that are apparent to the user) was an easier problem than physical data description, which might be true for large-scale database management systems, but I am not sure that I agree with him as it pertains to EDI. Indeed, his approach of regarding data logically first, and then its mapping to the physical media is the reverse of my approach, which is to decode a data file or stream and then apply semantic meaning to the decoded parts. However, McGee did recognize that standardizing data description could facilitate electronic data interchange, even if he did not approach the problem that way.

Sibley and Taylor (1973) also distinguished between the logical and physical characteristics of data, though they came up with a more detailed taxonomy than McGee (1972). In their six-level taxonomy, the first four levels were logical levels, with the final two being physical: (1) primitives, (2) structures, (3) relationships, (4) business objects, (5) file formats, and (6) collections of files. Once again, however, their approach was to map business objects to files rather than the reverse approach of parsing the data stream and then assigning meaning to the parse tree.

3.3.1.1 ASN.1. Abstract Syntax Notation One (ASN.1) is a formal language for describing the logical characteristics of data and communication protocols. It provides several sets of encoding rules for mapping the ASN.1 logical description to a physical stream of bits and bytes, though it does not mandate any one particular rule set for the physical data stream ("transfer syntax") for communication.

As an alternative to a universal data description language, it is insufficient, as it lacks a language for mapping ASN.1 to and from an arbitrary physical encoding. Therefore, it is only applicable when one of the ASN.1 transfer syntaxes is used. This is a common problem among the existing data description languages, which can describe only data that is physically formatted in certain restricted encodings.

3.3.1.2 XML. XML is a data language, capable of conveying hierarchically organized text-based data. Because of its lack of support for binary data, it was considered to have less risk of complications from heterogeneous implementations, so it has been widely adopted by the worldwide web for data transfer.

As with ASN.1, it is insufficient for use as a universal data description language as it is dependent on the use of the specific XML physical encoding. Although it can handle a multitude of character set encodings, it still has a basic physical format which is far from universal. There do exist data description languages specifically for describing the logical characteristics of an XML document, but as these are useful only for XML-encoded data, they have no bearing on XML's suitability as a universal data description language.

XML, however, is valuable as a model for a successful data language. Its success within the programming community serves as the model for the development of a universal data description language as described in Section 3.1.

3.3.1.3 Standard Exchange Format. As with ASN.1 and XML, Standard Exchange Format (SEF) is restricted to only a small class of data languages. SEF is used to describe different implementations of EDI-formatted transactions, but is useless when data is not strictly EDI. Most of the electronic data transmitted today does not conform to the strict EDI specifications of X12 or EDIFACT.

3.3.1.4 Database Management Systems. The *lingua franca* of database management systems (especially relational database management systems), is SQL. SQL is not a DDL *per se*, but rather a language for manipulating and querying data within a database. Many commercial DBMS vendors have created tools (e.g., SQL*Loader, BCP) for the import and export of data to and from their databases. These tools typically take some sort of description of the input or output data formats, and these descriptions are technically DDLs.

However, the DDLs offered by these tools are generally quite restricted. Usually, they can handle "flat" fixed-width and delimited record-based layouts. In some cases they can handle XML, but their support for the more exotic data languages ends there. They are also generally geared toward the mapping of external data formats to the relational database model rather than the creation of descriptive parse trees.

3.3.1.5 Data Script and Packet Types. Although they are distinct data description languages, Data Script (Back, 2002) and Packet Types (McCann & Chandra, 2000) are similarly capable of parsing binary data. However, they do not permit recursion and assume that the data stream is error-free (Fisher & Gruber, 2005). There certainly do exist recursive data types (e.g., bills of material, molecular descriptions), and, as a practical matter, error-handling and recovery is important in a universal data description language.

3.3.1.6 PADS. PADS (Fisher & Gruber, 2005) may be one of the most promising data description languages. Its syntax is closely related to that of C, which, while convenient for implementation in C, is decidedly inconvenient (though certainly possible) for implementation in other programming languages. In order to be useful, the language ought to be broken down to its underlying grammar and rebuilt in a more language-neutral (and simpler) form.

3.3.2 Grammar Models

Compilers have been used to transform high-level programming languages into machine code since the 1950's. Much prior research (Aho, Sethi, & Ullman, 1986) has been done on defining language grammars, semantics, and transformations (which is called compilation when referring to programming language translation), and this research will be useful to us in defining and implementing a universal data description language.

Although traditionally a compiler specializes in parsing only a single language, a "universal data parser" would have to be a generalist, capable of parsing any defined data format. Analogous generalist machines have been built for programming languages, called "compiler-compilers," such as Yacc (Johnson, 1975), proving the existence of such generalist machines. Optimization issues are slightly different in data language parsing versus program compilation. Whereas in modern compiler development it is the performance of the compiled output code which is important, in data language parsing the performance of the actual parsing activity is what is important. In compiler terms, instead of trying to create compilers that produce fast code, we are trying to create fast compilers that produce correct code.

We can leverage much of the prior research in languages and grammars, performed ostensibly for programming language compilation, in our development of a universal data description language. As we shall see, much of the theory of programming language grammars is based on natural language theory developed by scientists such as Chomsky (1956).

3.3.2.1 Regular Expressions. Regular expressions are probably the most commonly used grammar in practical use. Most modern languages have regular expression processing either built into the language or available in libraries. Their relatively simple syntax and efficient implementation make them ideal for pattern matching within applications.

A deterministic finite state automaton can be built to recognize any language described by a regular expression (and vice versa), resulting in recognition in linear

time. However, recognition and parsing are two different things, and parsing would require some back-tracking. Worst-case is therefore exponential time complexity, though, in practice, practical languages (especially data languages) do not generally require you to read to the last byte before you can make sense of the first, so the typical parse time is closer to the lower bound than the upper bound.

Regular expressions are only able to describe regular languages, rendering them too limited for to be used without modification as the basis for our universal data description language. However, they do have their use in parsing, frequently being used as the basis for the lexical analyzer. Furthermore, their simplicity, utility, and popularity has spawned several extensions to the basic regular expression, giving them the power to describe an even broader class of languages.

3.3.2.2 Extended Regular Expressions. The ability of regular expressions to describe more complex languages is extended in some common usages. Most of these are simply syntactic sugar; they improve the usability of the grammar without increasing its power. However, many modern implementations of regular expressions also provide a back-referencing capability whereby an earlier substring matching part of a regular expression can be referenced later in the expression. This permits the grammar to describe languages of repeating substrings such as "`abcabc`" or, more usefully, "`begin abc ... end abc`". We see this construction in the X12 270 transaction format (Health Care Eligibility Benefit Inquiry), which requires an identical control number to be included in both the header and the trailer of the transaction.

Extended regular expressions are able to describe some context-sensitive languages, but they still lack the ability to describe recursive (nested) structures which, although not as common in data languages as in programming languages, do occasionally arise (for instance in a bill of materials). Therefore, in current use, extended regular expressions do not have the power required of our universal data description language. Nevertheless, the extension of basic regular expressions to permit them to describe a larger class of languages sets an interesting precedent, and it is quite possible that the path to a universal data description language begins with the regular expression. In fact, most grammar models begin with the phrase-structure grammar instead.

3.3.2.3 Phrase-Structure Grammars. Most generative grammars are based on the phrase-structure grammar introduced by Chomsky (1956). The grammar in Figure 2.1(a) is an example of a phrase-structure grammar, which consists of a set of terminal symbols (symbols in the language being described), a set of nonterminal symbols (symbols not in the language but which are used to describe the language), a set of production or substitution rules (such as those shown in Figure 2.1(a)), and a starting nonterminal symbol to be used as a beginning point in the language generation. `The`, `cat`, and `mat` are terminal symbols in our example grammar, while *sentence*, *noun-phrase*, and *verb* are nonterminals in the grammar. The starting symbol is the nonterminal *sentence*.

Definition 3.1. A *phrase-structure grammar* is a quadruple $G = [\Sigma_L, \Sigma_G, R, S]$ where:

1. Σ_L is a finite set of terminal symbols in the language.
2. Σ_G is a finite set of nonterminal symbols used by the grammar to describe the language.

$$\Sigma_L \cap \Sigma_G = \emptyset$$

3. R is a finite set of pairs representing production rules where

$$(\alpha, \beta) \in R \Rightarrow \alpha \in (\Sigma_L \cup \Sigma_G)^+ \ \wedge \ \beta \in (\Sigma_L \cup \Sigma_G)^*$$

   $(\alpha, \beta) \in R$ is normally written $\alpha \rightarrow \beta$.
4. $S \in \Sigma_G$.

A phrase-structure grammar G describes the language $L(G)$ as the set of all strings in Σ_L^* that can be derived from the start symbol S and a finite number of applications of the production rules in R.

Chomsky (1956) classified grammars based on their production rules, known as the Chomsky hierarchy. The broadest class of phrase-structure grammars in the Chomsky hierarchy is unrestricted grammars, which Chomsky called Type 0. Unrestricted grammars are capable of describing all recursively enumerable languages, which certainly includes all practical data languages, but there exist no practical algorithms for generalized parsing based on an unrestricted Type 0 phrase-structure grammar, so this is not a reasonable basis for a data description language.

3.3.2.4 Context-Sensitive Grammars. Context-sensitive grammars, which Chomsky called Type 1 grammars, adds the restriction

$$(\alpha, \beta) \in R \Rightarrow \text{length}(\alpha) \leq \text{length}(\beta)$$

to the production rule set R, restricting the descriptive power of phrase-structure grammars to recursive languages. Although context-sensitive languages, the set of languages that can be described by context-sensitive grammars, are only a subset of recursive languages (Sudkamp, 1997), it is possible (though by no means certain) that all practical data languages will be context-sensitive languages.

Unfortunately, these grammars are of little help, since, like Type 0 grammars, there are no known practical general parsing algorithms based on context-sensitive grammars (unlike Type 0 grammars, however, it has not been proven that no such parsing algorithm exists), so this, too, is unlikely to be a viable basis for a data description language. Even if a practical context-sensitive grammar parser could be found, there may still be grammar models preferred over context-sensitive grammars for describing data, as the expression of long-distance relationships (that is, relationships between two parts of a data stream separated by a large amount of unrelated information) in a context-sensitive grammar is not very intuitive, and the parse trees that would be generated would probably not properly describe the data.

3.3.2.5 Context-Free Grammars. Type 2 context-free grammars restrict the definition of an unrestricted phrase-structure grammar so that

$$(\alpha, \beta) \in R \Rightarrow \alpha \in \Sigma_G$$

For the most part, context-free languages (languages that can be described by a context-free grammar) are a proper subset of context-sensitive languages (the trivial exception being that context-sensitive grammars cannot describe languages that include the empty string, whereas context-free grammars can). However, data languages are frequently not context-free (it is quite common in data languages for messages to require that certain identifiers be repeated, as in a header and trailer, or that checksums be computed, and neither of these constructs are context-free), so we are below the required threshold for universality when considering context-free grammars.

Context-free grammars can be parsed in about $O(n^3)$ in general, though in practice, context-free grammars will parse in much less time. (Earley, 1970) There are some further restrictions that can be made to the context-free grammar definition to guarantee $O(n)$ parsing, making context-free grammars a possible basis for a data description language. However, the more restricted LL grammars described below, always parsable in $O(n)$ time, may be a better starting point, since we certainly want the parser to run at $O(n)$ time in general.

3.3.2.5.1 Regular Right Part Grammars. An extension to context-free grammars is to permit regular expressions of terminal and nonterminal symbols on the

right side of the production rules. Such grammars are called regular right part grammars.

This extension does not increase the power of the underlying grammar, but it may make the grammar easier to use, and it may permit a better parse tree to be created, particularly as it applies to repetition. The regular right part grammar rule:

$$A \rightarrow \alpha\beta^*\gamma$$

may be converted to a standard CF grammar describing the same language using the following recursive definition:

$$A \rightarrow \alpha B\gamma$$

$$B \rightarrow \varepsilon \mid \beta B$$

However, if we interpret the Kleene star operation iteratively, then it comes to stand for:

$$A \rightarrow \alpha\gamma \mid \alpha\beta\gamma \mid \alpha\beta\beta\gamma \mid \alpha\beta\beta\beta\gamma \mid \ldots$$

which would produce a better parse tree (as in Figure 2.5).

3.3.2.5.2 LL Grammars. LL grammars are a subset of context-free grammars which can be parsed using a top-down approach. Top-down parsing of data is preferred over bottom-up, since a process that consumes the output of the parser would prefer the indication of what the data is to precede the data rather than follow it. That is, top-down parsing provides the ability for a parser to output, "Here is the

Social Security number" followed by nine digits, whereas bottom-up parsing would have output the nine digits followed by a message like, "That was the Social Security number."

While top-down parsing may require back-tracking, LL(k) grammars limit the amount of back-tracking (or require a buffer of length k in order to make the correct decision), permitting parsing to proceed in $O(n)$ time. LL(k) grammars are a viable basis upon which to build a data description language. However, because they can describe only a subset of the context-free languages (and context-free languages are only a subset of the languages we wish to describe) extensions such as attributes and indexing would be required to strengthen LL(k) grammars for our data description language.

3.3.2.5.3 LR Grammars. The LR family of grammars, including LR(k) and LALR(k), are context-free grammars efficient for bottom-up parsing. Although stronger than the LL family of grammars, they are still weaker than unrestricted CF grammars, so suffer from the same limitations as LL grammars without the benefit of top-down parsing. LL grammars are preferred as a basis for a UDDL over LR grammars.

3.3.2.6 Regular Grammars. Type 3 regular grammars restrict the phrase-structure grammar definition even further so that the grammar can describe exactly the languages described by regular expressions. For the most part, regular grammars are equivalent to regular expressions, though the syntax of regular expressions is more natural and more familiar to most programmers. In contrast to regular expressions,

regular grammars can be parsed in $O(n)$ time, though the resulting parse tree is of less use to the user, due to the awkwardness of language description using regular grammars.

If a phrase-structure grammar is to be used as the basis for a data description language, one of the variations on context-free grammars is likely a better choice, since they permit efficient parsing, describe a larger class of languages, and impose fewer restrictions on the user.

3.3.2.7 Two-Level Grammars. Two-level grammars such as van Wijngaarden (van Wijngaarden et al., 1975) and environment (Ruschitzka & Clevenger, 1989) grammars are capable of describing all recursively enumerable languages, making them as powerful as unrestricted phrase-structure grammars. They often make grammar writing easier and the resulting grammar more intuitive. Unfortunately, there is no known practical algorithm for general parsing with a two-level grammar. Furthermore, enhancing a more restricted phrase-structure grammar model (e.g., LL grammar model) with attributes or indices is probably even more intuitive for the grammar writer.

3.3.2.8 Affix and Attribute Grammars. Affix (Koster, 1971) and attribute (Knuth, 1968) grammars are quite similar in their operation. Although, traditionally, attribute grammars have been used to enforce semantic rather than syntactic rules, there is no particular reason why they cannot be used to enforce syntactic rules as do affix grammars. (More on this later, in Section 4.2, but in fact attributes in grammars have typically not been used to *enforce* anything, but rather to mark up parse trees with semantic values.) Indeed, the boundary between syntax and semantics can be

blurry when dealing with data languages. To a large extent, affix grammars have been replaced by attribute grammar models (Grune & Jacobs, 1990), which is unsurprising, given their similarity. Not only are attributes easier to add on to a context-free grammar to provide a measure of context sensitivity, it does not seem impossible to add attributes to regular expressions—which are even easier for most grammar writers to use—and thus make them more powerful, though I have never seen anyone do this.

L-attributed grammars—attribute grammars that allow attributes to be evaluated left-to-right—are of particular interest in parsing. No attribute evaluation ever requires the value of an attribute assigned later in the parsing. This is very helpful in facilitating top-down parsing, which is preferred over bottom-up.

3.3.2.9 Parsing Expression Grammars. Most grammar models, being based on Chomsky's phrase structure grammar, are generative; that is, they define a language by describing how to generate every string in the language and only strings in the language. Parsing is the reverse process, of taking an input string and finding a way to generate the string using the generative process, or signaling an error if the string is not in the language. Since generative grammar models are not designed for parsing, there has been much research on the subject of creating (efficient) parsers given a generative grammar for a language.

An alternative type of grammar model is an analytic grammar. An analytic grammar, instead of describing how to generate the strings in a language, describes

how to determine whether a given input string is in a language. In this manner, it is describing how to analyze a string in terms of the grammar.

One promising analytic grammar model is parsing expression grammars (Ford, 2004). Parsing expression grammars bear a marked similarity to regular right part grammars. Although not original to parsing expression grammars, the model's most distinguishing feature is the ordering of alternatives. Whereas in a phrase-structure grammars, the rule $A \rightarrow \alpha \mid \beta$ is equivalent to $A \rightarrow \beta \mid \alpha$, this is not true in a parsing expression grammar. In parsing expression grammars, alternation is ordered (and written "/" to distinguish it from the unordered alternation " | ") so that $A \leftarrow \alpha / \beta$ means to only try matching the input to β if the input does not match α. Therefore, $A \leftarrow \alpha / \beta$ is clearly not equivalent to $A \leftarrow \beta / \alpha$. Parsing expressions can also permit look-ahead, allowing the parser to look ahead into the input stream to determine which choice to take.

Parsing expression grammars can describe a wide class of languages, including some context-free and context-sensitive languages. It is unproven whether parsing expression grammars can describe all context-free languages or not.

3.3.3 Research Scope

There being no existing data description languages suitable for universal data description, a new data description language must be developed. Of existing grammar models, only a few provide a promising basis for development of a universal data description language:

- regular expressions
- context-free grammars, particularly LL grammars, and possibly regular right part grammars
- attribute grammars, particularly L-attributed grammars
- parsing expression grammars

In addition to the existing grammar models, it may be necessary to invent new ones if these prove inadequate. Two ideas are (1) a model that more closely matches the way the human mind views and parses data, and (2) a model that treats syntactic elements in a more object-oriented way (e.g., with classes, inheritence, parsing rules, etc.).

While the idea of creating a wholly new grammar model is appealing (which, to satisfy my vanity, I would no doubt name *Mercer grammars*), it is premature at this time to do so without having thoroughly explored options with existing models. After all, insight gained by a closer look at existing grammars may prove helpful in two different ways:

1. Existing models may prove to be well-suited to our objective, obviating the need to develop a new model.
2. If adaptation of existing models proves not to be adequate, the insight gained by the investigation may be applied in the development of new grammar models.

Parsing expression grammars are attractive as a basis for our universal data description language because they easily permit description of both lexical and syntactic

structure of languages, and they do not permit ambiguous grammars to be written, making them safer than Chomsky's phrase-structure grammars. Legitimate data languages will never be ambiguous.

CHAPTER 4

ATTRIBUTED PARSING EXPRESSION GRAMMARS

In this chapter, attributed parsing expression grammars (APEGs) are formally defined. In Section 4.1, a formal definition of unattributed parsing expression grammars (PEGs) is shown along with an examples which will aid the reader in understanding PEGs. Section 4.2 describes the principle of enhancing existing grammar models with attributes. Section 4.3 combines the PEG grammar model with attributes to define the new APEG grammar model.

An attempt has been made to define our concepts in both prose and mathematical formalisms, the former being useful upon first reading to understand the concepts, the latter being useful once the concepts are properly understood for easy reference later on. We borrow some notation from the field of formal logic to assist us making our formalisms easy to read. Just as in mathematics we might write an equation vertically

$$\begin{array}{r} 12 \\ +\ 5 \\ \hline 17 \end{array}$$

where the problem is shown above the line and the solution below, formal logic uses similar notation, with premises written above a horizontal line and conclusions below.

$$\frac{\textit{premises}}{\textit{conclusions}}$$

For example, the transitive property of equality can be written

$$\frac{a = b \qquad b = c}{a = c}$$

and may be read, "Given $a = b$ and $b = c$, then it follows that $a = c$." This is decidedly easier to read than the more compact form of this statement:

$$a = b \ \wedge \ b = c \Rightarrow a = c$$

4.1 Parsing Expression Grammars

Parsing expression grammars (PEGs) were first defined by Ford (2004). The ideas behind PEGs were not new, but the parsing process had not previously been recognized nor formally defined as a grammar. They bear similarities (and are, in fact, proven to be equivalent in expressive power) to Top-Down Parsing Language (TDPL) and Generalized TDPL (GTDPL) (Aho & Ullman, 1972).

Like phrase-structure grammars, PEGs will be defined as a quadruple of terminal and nonterminal symbols, a rule set, and a start expression. Distinguishing PEGs from PS grammars, however, is that the rule-set is a function mapping nonterminal symbols to parsing expressions, and the start expression may, too, be a parsing expression, rather than being restricted to being a nonterminal. Therefore, we must first define parsing expressions.

Definition 4.1. *Parsing expressions* over a language alphabet Σ_L and a set of nonterminals Σ_G are defined inductively with the following base cases and recursive rules.

Empty String. The empty string is a parsing expression.

$$\frac{}{\varepsilon \text{ is a parsing expression}}$$

Terminal. Any terminal symbol in the language alphabet Σ_L is a parsing expression.

$$\frac{a \in \Sigma_L}{a \text{ is a parsing expression}}$$

Nonterminal. Any nonterminal symbol in Σ_G is a parsing expression.

$$\frac{A \in \Sigma_G}{A \text{ is a parsing expression}}$$

Concatenation. The concatenation of two parsing expressions is a parsing expression. If e_1 and e_2 are parsing expressions, we write this simply as "$e_1 e_2$".

$$\frac{e_1 \text{ is a parsing expression} \qquad e_2 \text{ is a parsing expression}}{e_1 e_2 \text{ is a parsing expression}}$$

Prioritized Choice. A valid parsing expression can be created by offering the choice between two parsing expressions. If e_1 and e_2 are parsing expressions, the choice is written "$e_1 \,/\, e_2$".

$$\frac{e_1 \text{ is a parsing expression} \qquad e_2 \text{ is a parsing expression}}{e_1 \,/\, e_2 \text{ is a parsing expression}}$$

Unlike nondeterministic alternation, alternation in parsing expressions is not always commutative, and permuting the expressions may produce semantically distinct parsing expressions. The second choice is tried only if the first choice fails immediately. If the first choice succeeds, the second choice is never attempted, even if the first choice results in a parsing error later on where the second choice would not have. This important distinction allows parsing based on PEGs to be performed in linear time.

Greedy Repetition. The repetition of a parse using a parsing expression as many times as possible is a parsing expression. If e is a parsing expression, we write this "e^*".

$$\frac{e \text{ is a parsing expression}}{e^* \text{ is a parsing expression}}$$

Greedy repetition is distinguished from the more familiar nondeterministic repetition of phrase-structure grammars by the fact that the greedy repetition will consume as much of the input as possible without regard for whether such greediness might later cause a parse error. In contrast, nondeterministic repetition omnisciently consumes precisely as much of the input needed to produce a successful parse.

Negative Predication. A parsing expression can be negated to form another parsing expression. If e is a parsing expression, we may write this "$!e$".

$$\frac{e \text{ is a parsing expression}}{!e \text{ is a parsing expression}}$$

Semantically, a negated parsing expression will succeed wherever the nonnegated parsing expression fails to parse, and will fail where the latter succeeds. The construct is a predicate because it consumes no input from the input stream, so the expression is simply making a statement—a predication—regarding the input stream.

Definition 4.2. A *parsing expression grammar* (PEG) is a quadruple $G = [\Sigma_L, \Sigma_G, R, e_s]$ where:

1. Σ_L is a finite set of terminal symbols in the language.
2. Σ_G is a finite set of nonterminal symbols used by the grammar to describe the language. Nonterminal symbols are not in the language being described.

$$\Sigma_L \cap \Sigma_G = \emptyset$$

3. R is a function mapping grammar nonterminals to parsing expressions over the terminal and nonterminal symbols, Σ_L and Σ_G respectively. We write $(A, e) \in R$ as "$A \leftarrow e$". In contrast to phrase-structure grammars, R is a function rather than just a relation; a nonterminal maps to at most one associated parsing expression.
4. e_s is the start expression that defines where parsing is to begin. Although not required by definition, the starting expression will often be a nonterminal symbol.

We may often abbreviate the representation of a PEG by listing only the rules in R, explicitly specifying the start expression when it is not obvious. Thus we can define a PEG as easily as we can a CFG, as in Example 4.3

Example 4.3. Consider the following PEG.

$$\begin{aligned}
\Sigma_L &= \{\mathtt{a}\} \\
\Sigma_G &= \{A\} \\
R &= \{A \leftarrow \mathtt{a} A \mathtt{a} \,/\, \mathtt{a}\} \\
e_s &= A\,!\mathtt{a}
\end{aligned}$$

We may abbreviate this grammar simply as:

$$A \leftarrow \mathtt{a} A \mathtt{a} \,/\, \mathtt{a}$$

where the other components of the grammar are implied. (The "!a" at the end of the start expression e_S is required to ensure that an input string is completely matched by the grammar rather than partially matched. When our language alphabet consists only of the symbol a, then !a will match only at the end of the string.)

The language accepted by this PEG is surprising. It is not the same language accepted by the corresponding CFG.

$$A \rightarrow \mathtt{a} A \mathtt{a} \mid \mathtt{a}$$

To demonstrate, consider the input string $s = \mathtt{a\,a\,a\,a\,a}$. This string may be generated by the CFG, as shown in Figure 4.1(a). On the other hand, attempting to parse this string with the PEG results in failure (Figure 4.1(b)).

$A \rightarrow \mathrm{a}A\mathrm{a} \mid \mathrm{a}$

A	start expression
$\mathrm{a}A\mathrm{a}$	substitute $A \rightarrow \mathrm{a}A\mathrm{a}$
$\mathrm{aa}A\mathrm{aa}$	substitute $A \rightarrow \mathrm{a}A\mathrm{a}$
aaaaa	substitute $A \rightarrow \mathrm{a}$

(a)

$A \leftarrow \mathrm{a}A\mathrm{a} \,/\, \mathrm{a}$

1. A !a
2. (aAa / a) !a
3. (a(aAa / a)a / a) !a
4. (a(a(aAa / a)a / a)a / a) !a
5. (a(a(a(aAa / a)a / a)a / a)a / a) !a
6. (a(a(a(a(aAa / a)a / a)a / a)a / a)a / a) !a
7. (a(a(a(a(a(aAa / a)a / a)a / a)a / a)a / a)a / a) !a
8. (a(a(a(a(aaa / a)a / a)a / a)a / a)a / a) !a
9. (a(a(a(aaa / a)a / a)a / a)a / a) !a
10. (a(a(aaa / a)a / a)a / a) !a
11. (a(aaaaa / a)a / a) !a
12. (aaa / a) !a
13. aaa !a

(b)

Figure 4.1. A Parsing Expression Grammar Produces an Unexpected Result. Although the CFG and PEG look similar, *(a)* the CFG is shown to generate a string of five a's. *(b)* However, when the PEG grammar analyzes the string, the "greediness" of the PEG model's alternation causes the analysis to reject the string. The PEG parser tries the first choice (aAa) until it results in a sequence of six a's. Since there is no a in the sixth position (underlined), the choice fails, and the second choice (a) is used. However, in step 10, the first choice results in a sequence of only five a's, so it is the second choice that is discarded. This decision results in too many a's in step 11, but discarding the first choice in that step results in too few a's at the end, and the final predicate in step 13 will fail, there being an a in the fourth position. There being no remaining alternative, the entire parse will fail.

This result of Example 4.3 does not mean that PEGs are in any way "weaker" than CFGs. On the contrary, it is the PEG grammar model's ability to describe this language (which, by the way, is $\texttt{a}^n$ where $n = 2^m - 1 = 1,3,7,15,\ldots$ for $m = 1,2,3,4,\ldots$) that might make PEGs actually stronger than CFGs. It has been proven that PEGs can describe languages that CFGs cannot (Ford, 2004). The opposite has not been proven.

A further item of interest is that parsers based on a PEG are able to parse an input string in time proportional to the length of the input, $O(n)$. This can be demonstrated by again using the PEG of Example 4.3. We define several functions based on parsing expressions in the grammar which take a natural number as their argument and return the length of a match for that parsing expression from that position in the input. For instance, the function $A(n)$ may be defined to return the length of the match of nonterminal A at position n in the input. If A does not match the input at position n, then $A(n)$ is not defined.

$$
\begin{aligned}
a(n) &= 1 \quad \text{if } s(n) = \texttt{a} \\
aAa(n) &= a(n) + A(n + a(n)) + a(n + a(n) + A(n + a(n))) \\
A(n) &= \begin{cases} aAa(n) & \text{if } aAa(n) \text{ is defined} \\ a(n) & \text{otherwise} \end{cases} \\
!a(n) &= 0 \quad \text{if } a(n) \text{ is not defined} \\
e_s(n) &= A(n) + !a(n + A(n))
\end{aligned}
$$

	n					
	0	1	2	3	4	>4
$s(n)$	a	a	a	a	a	X
$a(n)$	1	1	1	1	1	X
$aAa(n)$	3	X	3	X	X	X
$A(n)$	3	1	3	1	1	X
$!a(n)$	X	X	X	X	X	0
$e_s(n)$	X	X	3	X	1	X

(a)

	n							
	0	1	2	3	4	5	6	>6
$s(n)$	a	a	a	a	a	a	a	X
$a(n)$	1	1	1	1	1	1	1	X
$aAa(n)$	7	5	3	X	3	X	X	X
$A(n)$	7	5	3	1	3	1	1	X
$!a(n)$	X	X	X	X	X	X	X	0
$e_s(n)$	7	X	X	X	3	X	1	X

(b)

Table 4.1. PEG Parsing Function Examples. Values of the helper functions for the grammar of Example 4.3 for two input strings *(a)* a^5 and *(b)* a^7. Cells marked with "X" are undefined. The values for the cells can be easily calculated by hand in order top-to-bottom, right-to-left. An input string is in the language defined by the PEG if and only if $e_s(0)$ is defined. Thus it can be seen that a^5 is not in the language, whereas a^7 is.

In the formulae, s is the input string. (Refer to Definition 2.2 for the definition of a string as a function.) These functions are better illustrated with some concrete examples. Table 4.1 shows the values of these functions for the strings a^5 and a^7. A PEG parser does not necessarily have to calculate the value of each function for every n ($n < |s|$), more likely calculating the values only as required and memoizing the value in case it is needed later. However, even in the worst case where every value must be calculated, the number of calculations is still proportional to the length of the input.

When we compare the performance of PEG parsing with that of the best general CFG parsing algorithms (which run in polynomial time), and note that PEGs can describe languages that CFGs cannot, we might reasonably conclude that even unattributed PEGs are superior to the more common CFGs. However, the PEG

grammar model is still unable to adequately describe all data languages (it is unable to describe a language with repetition of substrings), so some enhancement is required.

4.2 Attributed Grammars

Attributes were first added to existing grammar models in an effort to describe the semantics of context-free languages (Knuth, 1968). In the initial definition, attributes did not modify the parse tree at all, but rather attempted to provide an interpretation of it. For example, a parser based on a standard, unattributed grammar may recognize the string "`13.25`" as a number. However, by enhancing the grammar with attributes, we can add an attribute to the branch of the parse tree containing the number actually providing its value: thirteen and a quarter.

Knuth classified attributes into two groups: synthesized attributes, which are produced by a production rule, and inherited attributes, which are passed into the production rule. Although the same semantics can be described by using only synthesized attributes, inherited attributes make the grammar model easier to use. Other authors (Vogt, Swierstra, & Kuiper, 1989) have added a third class of attributes, local attributes, which are neither inherited nor synthesized, but which are used within a production rule to simplify the grammar.

When studying the semantics of context-free languages, Knuth assigned values to attributes based on the string being parsed, but the attributes did not affect the success or failure of the parse, so the resulting parse tree had the same shape as the equivalent unattributed grammar. Value attributes were assigned to nodes of the tree

only to describe the semantics. It is no great leap, however, to use the attribute values to validate the parsing; after all, some understanding of the semantics of a string is sometimes necessary in order to correctly parse it. In the example where the string "`13.25`" evaluated to the value 13.25, if the language required that the value of that string be no greater than 12, then we could reject the parse tree in favor of one that does not violate our constraint.

Knuth put attribute valuation at the end of each production rule. This was appropriate because his attributes were only intended to describe semantics, and the grammar model to which he was adding attributes, vanilla PS grammars, do not support union nor repetition in a production rule. (Union in a PS grammar is represented by multiple production rules for the same nonterminal, and repetition is represented recursively.) Thus, the only operation that appeared in a production was concatenation. In PEGs, however, union (in the form of prioritized choice) and repetition are built into the language of a parsing expression, so different attribute valuation rules may be required depending on the alternative chosen in a prioritized choice. An intuitive way to implement this is to embed attribute valuation instructions at the appropriate point in the parsing expression rather than appending them to the end. Further clarification is required to define how to handle attributes under the parsing expression repetition operator.

It is natural when considering attribute models for PEGs to consider only L-attributed models: models in which attributes can always be evaluated left to right. Indeed, it would be counterproductive to introduce a nondeterministic attribute

model onto deterministic PEGs. L-attributed attribute models require specific ordering to the attribute rules to ensure that they can always be evaluated when encountered, and since we do not (only) intend for attributes to add semantic values to the information model, but instead to use the semantic values in the validation process, it is natural for the attributes to be valued and validated within the parsing expression.

4.3 APEGs Defined

In this section, we define attributed parsing expression grammars (APEGs) formally. In order to define APEGs, we shall first formally define attributes, and then define attribute contexts in which attribute expressions are evaluated. Attributed parsing expressions are parsing expressions that have embedded in them these attribute contexts and functions to manipulate the contexts. We shall define attributed parsing expressions, and then use them to define our grammar based upon them. We shall also define what it means to parse a string with an APEG.

4.3.1 Attributes

An attribute is a named value, like variables in mathematics. When we write "$x = 18$", we are assigning the name x to the value 18. Later, we can refer to the value 18 by its name, x. Attributes play the same role in attributed grammars as variables do in mathematics. We can assign an attribute named x the value 18. In the case of a formal definition, we must define these names and values. Attribute names are taken from an alphabet which we shall designate Σ_α, and they may be assigned any value

in the set $\mathbb{T}$. Set $\mathbb{T}$ may be (and is very likely) an infinite set, though given that the purpose of APEGs is to facilitate automatic parsing, it is reasonable to restrict the set to values that can practically be represented and manipulated by a computer. At any rate, we do not define this set within our grammar model, leaving that to a concrete implementation. The set of attribute names Σ_α, like all alphabets, is finite.

Definition 4.4. An *attribute* is a pair $[\alpha, v]$ where $\alpha \in \Sigma_\alpha$ is the name of the attribute and $v \in \mathbb{T}$ is its value.

Definition 4.5. An *attribute context* is a collection of attributes such that an attribute name is assigned at most one value. We use the letter m to denote an attribute context, which we model as a function $m : \Sigma_\alpha \to \mathbb{T}$ mapping attribute names to their values. We denote the set of all possible attribute contexts with the capital letter M. Thus, $M = (\Sigma_\alpha \to \mathbb{T})$ and $m \in M$.

4.3.2 Attributed Parsing Expressions

Definition 4.6. *Attributed parsing expressions* over a language alphabet Σ_L, a set of non-terminals Σ_G, and attribute contexts M are defined inductively with the following base cases and recursive rules.

Empty String. The empty string is an attributed parsing expression.

$$\frac{}{\varepsilon \text{ is an attributed parsing expression}}$$

Terminal. Any terminal symbol in the language alphabet Σ_L is an attributed parsing expression.

$$\frac{a \in \Sigma_L}{a \text{ is an attributed parsing expression}}$$

Nonterminal. In contrast to unattributed parsing expressions, nonterminal grammar symbols cannot stand alone as attributed parsing expressions. Two functions are required to provide attribute context: one inheritence function f_{inh} to map the attribute context in which the nonterminal appears to an inherited attribute context in which the nonterminal will be parsed; and a synthesis function f_{synth} to map the attributes synthesized by the nonterminal back into the original attribute context.

$$\frac{A \in \Sigma_G \qquad f_{\text{inh}} : M \to M \qquad f_{\text{synth}} : M \times M \to M}{[A, f_{\text{inh}}, f_{\text{synth}}] \text{ is an attributed parsing expression}}$$

Concatenation. The concatenation of two attributed parsing expressions is an attributed parsing expression. If e_1 and e_2 are parsing expressions, we write this simply as "$e_1 e_2$".

$$\frac{e_1 \text{ is an attributed parsing expression} \qquad e_2 \text{ is an attributed parsing expression}}{e_1 e_2 \text{ is an attributed parsing expression}}$$

Prioritized Choice. A valid attributed parsing expression can be created by offering the choice between two attributed parsing expressions. If e_1 and e_2 are attributed parsing expressions, the choice is written "$e_1 \,/\, e_2$".

$$\frac{e_1 \text{ is an attributed parsing expression} \qquad e_2 \text{ is an attributed parsing expression}}{e_1 \,/\, e_2 \text{ is an attributed parsing expression}}$$

Greedy Repetition. An attributed parsing expression repeated as many times as possible is an attributed parsing expression. As with unattributed parsing expressions, the repetition is greedy; the resulting attributed parsing expression will consume input until it can consume no more, even if in so doing it results in a failed parse where a more abstemious repetition might have succeeded.

$$\frac{e \text{ is an attributed parsing expression}}{e^{*} \text{ is an attributed parsing expression}}$$

Negative Predication. An attributed parsing expression can be negated to form another attributed parsing expression. As with parsing expressions, a predicate consumes no input, regardless of outcome. (However, as we shall see, it may modify the

attribute context.)

$$\frac{e \text{ is an attributed parsing expression}}{!e \text{ is an attributed parsing expression}}$$

Attribute Usage. Attributes may be used as either predicates or to perform operations that modify the context (or both). As a predicate, the usage function f is defined only for attribute contexts in which the predicate is true.

$$\frac{f : M \to M}{f \text{ is an attributed parsing expression}}$$

4.3.3 Attributed Parsing Expression Grammars

Definition 4.7. An *attributed parsing expression grammar* (APEG) is defined as the 5-tuple $G = [\Sigma_L, \Sigma_G, M, R, e_s]$ where:

1. Σ_L is a finite set of terminal symbols in the language.
2. Σ_G is a finite set of nonterminal symbols used by the grammar to describe the language. Nonterminal symbols are not in the language being described.

$$\Sigma_L \cap \Sigma_G = \emptyset$$

3. M is the set of attribute contexts mapping attribute names to values, as defined by the attribute names, Σ_α, and values, $\mathbb{T}$.

4. R is a function mapping grammar nonterminals to attributed parsing expressions over the terminal and nonterminal symbols Σ_L and Σ_G, and attribute contexts M. Because there are a finite number of nonterminal symbols, there are a finite number of rules in R.
5. e_s is the start expression that defines where parsing is to begin.

4.3.4 The Semantics of Attributed Parsing Expression Grammars

We shall define parsing as a function π where the inputs are an attribute context, an attributed parsing expression, and a string to parse, and the outputs are an attribute context and either the parsed portion of the string or the symbol $\texttt{fail} \notin \Sigma_L^*$ indicating a failure to parse. A string $x_1 x_2 \in \Sigma_L^*$ is in the language described by an APEG if $\pi(\emptyset, e_s, x_1 x_2) = [m, x_1]$ for the grammar's start expression e_s and some attribute context $m \in M$. As with PEGs—but in contrast to phrase-structure grammars—we do not require the entire string to match in order to have successful parse.

Definition 4.8. The parse function $\pi : M \times E \times \Sigma_L^* \to M \times (\Sigma_L^* \cup \{\texttt{fail}\})$ (where E is the set of attributed parsing expressions over Σ_L, Σ_G, and M) for the grammar $G = [\Sigma_L, \Sigma_G, M, R, e_s]$ is defined inductively with the following base cases and recursive rules.

Empty. The empty attributed parsing expression always successfully parses, consuming no input and making no modifications to the attribute context.

$$\frac{}{\pi(m, \varepsilon, x) = [m, \varepsilon]}$$

Terminal. An attributed parsing expression consisting of a single terminal symbol succeeds if and only if the first symbol in the string to parse is the same symbol. If they match, the symbol is consumed from the input string. This attributed parsing expression will fail when the input is the empty string. In any case, no changes are made to the attribute context.

Success Case.

$$\frac{a \in \Sigma_L}{\pi(m, a, ax) = [m, a]}$$

Failure Case 1.

$$\frac{a \in \Sigma_L \quad b \in \Sigma_L \quad a \neq b}{\pi(m, a, bx) = [m, \texttt{fail}]}$$

Failure Case 2.

$$\frac{a \in \Sigma_L}{\pi(m, a, \varepsilon) = [m, \texttt{fail}]}$$

Nonterminal. Nonterminal attributed parsing expressions require the use of the attribute inheritence and synthesis functions to parse correctly. When parsing fails, the attribute context is not changed.

Success Case.

$$\frac{[A, e] \in R \qquad \pi(f_{\text{inh}}(m), e, xy) = [m', x]}{\pi(m, [A, f_{\text{inh}}, f_{\text{synth}}], xy) = [f_{\text{synth}}(m, m'), x]}$$

Failure Case.

$$\frac{[A, e] \in R \qquad \pi(f_{\text{inh}}(m), e, x) = [m', \texttt{fail}]}{\pi(m, [A, f_{\text{inh}}, f_{\text{synth}}], x) = [m, \texttt{fail}]}$$

Note that the result of the parse is not defined when either $f_{\text{inh}}(m)$ or $f_{\text{synth}}(m, m')$ is undefined.

Sequence. A sequence attributed parsing expression parses both component attributed parsing expressions in turn, succeeding only when both succeed. When parsing fails, the attribute context is not changed.

Success Case.

$$\frac{\pi(m, e_1, xyz) = [m', x] \qquad \pi(m', e_2, yz) = [m'', y]}{\pi(m, e_1 e_2, xyz) = [m'', xy]}$$

Failure Case 1.

$$\frac{\pi(m, e_1, x) = [m', \texttt{fail}]}{\pi(m, e_1 e_2, x) = [m, \texttt{fail}]}$$

Failure Case 2.

$$\frac{\pi(m, e_1, xy) = [m', x] \qquad \pi(m', e_2, y) = [m'', \texttt{fail}]}{\pi(m, e_1 e_2, xy) = [m, \texttt{fail}]}$$

Alternation. The first alternative is always parsed. If it succeeds, its result is the result of the alternation. If it fails, then the second alternative is parsed with the original attribute context. If it succeeds, its result is the result of the alternation. If both parses fail, that alternation parse fails and the attribute context is unchanged.

Success Case 1.

$$\frac{\pi(m, e_1, xy) = [m', x]}{\pi(m, e_1 \,/\, e_2, xy) = [m', x]}$$

Success Case 2.

$$\frac{\pi(m, e_1, xy) = [m', \texttt{fail}] \qquad \pi(m, e_2, xy) = [m'', x]}{\pi(m, e_1 \,/\, e_2, x) = [m'', x]}$$

Failure Case.

$$\frac{\pi(m, e_1, x) = [m', \texttt{fail}] \qquad \pi(m, e_2, x) = [m'', \texttt{fail}]}{\pi(m, e_1 \,/\, e_2, x) = \pi(m, \texttt{fail})}$$

Repetition. Parsing of a repetition attributed parsing expression always succeeds.

Termination Case.

$$\frac{\pi(m, e, x) = [m', \texttt{fail}]}{\pi(m, e^*, x) = [m, \varepsilon]}$$

Repetition Case.

$$\frac{\pi(m, e, xyz) = [m', x] \qquad \pi(m', e^*, yz) = [m'', y]}{\pi(m, e^*, xyz) = [m'', xy]}$$

Negative Predication. A negative predication attributed parsing expression $!e$ succeeds where e fails and fails where e succeeds. As a predicate, it never consumes any input.

Success Case.

$$\frac{\pi(m, e, x) = [m', \texttt{fail}]}{\pi(m, !e, x) = [m', \varepsilon]}$$

Failure Case.

$$\frac{\pi(m, e, xy) = [m', x]}{\pi(m, !e, xy) = [m', \texttt{fail}]}$$

Note the failure case, which is the only rule that can fail and change its attribute context. This enables an affirmative predicate composed of the negation of a negative predicate to succeed and change the attribute context. Based on the above rules, it follows that:

$$\frac{\pi(m, e, xy) = [m', x]}{\pi(m, !!e, xy) = [m', \varepsilon]}$$

The negative predication attributed parsing expression is also the only attributed parsing expression that makes use of a context change on failure, so a context change on failure will never propagate beyond an immediate reapplication of the negative predicate.

Attribute Usage. Attribute usage is a function on the attribute context. It does not read the input string at all, and no input is consumed. A parse of an attribute usage attributed parsing expression will fail, however, if the function is not defined for the input attribute context. In this way, an attribute usage attributed parsing expression can be used as a predicate. It is a pure predicate (i.e., it makes no changes) if $f(m) = m$ for all m where $f(m)$ is defined.

Success Case.

$$\frac{f(m) \text{ is defined}}{\pi(m, f, x) = [f(m), \varepsilon]}$$

Failure Case.

$$\frac{f(m) \text{ is not defined}}{\pi(m, f, x) = [m, \mathtt{fail}]}$$

Example 4.9. The following APEG describes the language of all strings over $\Sigma_L = \{\mathtt{a}, \mathtt{b}\}$ in which the first half of the string equals the second half. That is, xx is in the language for all strings $x \in \Sigma_L^*$. This is an important language, since it cannot be de-

scribed by either a CFG or a PEG.

$$
\begin{aligned}
\Sigma_G &= \{A, B, C, D, E\} \\
M &= (\{\mathtt{c}\} \times \Sigma_L) \cup (\{\mathtt{s}, \mathtt{t}\} \times \Sigma_L^*) \cup (\{\mathtt{u}\} \times (\Sigma_L \cup \varepsilon)) \\
R &= \left\{ \begin{array}{lll}
A & \leftarrow & \mathtt{a}\,(\lambda m\alpha\,.\,\text{if } \alpha = \mathtt{c} \text{ then } \mathtt{a} \text{ else } m(\alpha)) \\
B & \leftarrow & \mathtt{b}\,(\lambda m\alpha\,.\,\text{if } \alpha = \mathtt{c} \text{ then } \mathtt{b} \text{ else } m(\alpha)) \\
C & \leftarrow & [A, \emptyset, \text{synth}(\{\mathtt{c}\})] \,/\, [B, \emptyset, \text{synth}(\{\mathtt{c}\})] \\
D & \leftarrow & [C, \emptyset, \text{synth}(\{\mathtt{c}\})][E, \text{inh}(\{\mathtt{s}, \mathtt{t}, \mathtt{u}\}), \text{synth}(\{\mathtt{s}, \mathtt{t}, \mathtt{u}\})] \\
E & \leftarrow & ((\lambda m\,.\,\text{if } m(\mathtt{u}) = \varepsilon \text{ then } m) \\
 & & \quad (\lambda m\alpha\,.\,\text{if } \alpha = \mathtt{u} \text{ then } \text{part}_L(m(\mathtt{t}), 1) \\
 & & \qquad \text{else if } \alpha = \mathtt{t} \text{ then } \text{part}_R(m(\mathtt{t}), 1) \\
 & & \qquad \text{else } m(\alpha) \quad) \\
 & & \quad /\,(\lambda m\alpha\,.\,\text{if } \alpha = \mathtt{s} \text{ then } \text{concat}(m(\mathtt{s}), m(\mathtt{u})) \\
 & & \qquad \text{else if } \alpha = \mathtt{u} \text{ then } \varepsilon \\
 & & \qquad \text{else } m(\alpha) \quad)
\end{array} \right\} \\
e_s &= (\lambda m\alpha\,.\,\text{if } \alpha \in \{\mathtt{s}, \mathtt{t}, \mathtt{u}\} \text{ then } \varepsilon) \\
 &\quad\ [D, \text{inh}(\{\mathtt{s}, \mathtt{t}, \mathtt{u}\}), \text{synth}(\{\mathtt{s}, \mathtt{t}, \mathtt{u}\})]^* \\
 &\quad\ (\lambda m\,.\,\text{if } m(\mathtt{s}) = m(\mathtt{t}) \text{ then } m)
\end{aligned}
$$

where inh and synth are functions that produce inheritence and synthesis functions respectively for the attributes named in the argument set.

$$
\begin{aligned}
\text{inh}(V) &= \lambda m\alpha\,.\,\text{if } \alpha \in V \text{ then } m(\alpha) \\
\text{synth}(V) &= \lambda m m'\alpha\,.\,\text{if } \alpha \in V \text{ then } m'(\alpha) \text{ else } m(\alpha)
\end{aligned}
$$

The attributes used by our grammar are defined by the parameter M. Attribute c is used to hold the last symbol read by the parser, so it has a type of Σ_L, any symbol

in the language alphabet. Attributes s and t are used to hold the first and second halves of the string respectively, and so have a type of Σ_L^*, any string in the language alphabet. When an odd number of symbols have been read from the input by the parser, the attribute u is used to hold the middle symbol, which is neither in the first or second half. It has type $\Sigma_L \cup \varepsilon$, since it can be any symbol in the language alphabet or empty.

CHAPTER 5

EVALUATION OF APEGS

We shall here evaluate attributed parsing expression grammars on the basis of the criteria enumerated in 3.2.

5.1 Generality

It can be shown that APEGs are truly universal: they are capable of describing any recursive language.

Theorem 5.1. *Any recursive language L can be described by an attributed parsing expression grammar.*

Proof. If a language L is recursive, then there exists a function $f_L : \Sigma_L^* \rightarrow \{\texttt{true}, \texttt{false}\}$ which decides whether an input string is in the language or not. We can therefore construct the APEG $G = [\Sigma_L, \Sigma_G, \Sigma_\alpha, \mathbb{T}, R, e_s]$ to describe L.

1. Define the specifications of the language to be described:

   a) Let L be the recursive language to be described.

   b) Let Σ_L be the finite alphabet of L.

   c) Let $n = |\Sigma_L|$.

   d) Let $a_1, a_2, \ldots, a_n$ be the distinct elements of Σ_L.

2. Define the attributes to be used by the APEG. We shall read each input character into the attribute c, and form the string s from the concatenation of the characters. When the input has been completely read, s will equal the input string.

   a) Let $\Sigma_\alpha = \{\mathtt{c}, \mathtt{s}\}$, the attribute names to be used by the APEG.

   b) Let $\mathbb{T} = \Sigma_L^*$, the possible values of attributes.

3. Define APEG rules to read the next character from the input string and put it in attribute c:

   a) For $1 \leq i \leq n$, define C_i to be a nonterminal symbol corresponding to the terminal symbol a_i in Σ_L.

   b) For $1 \leq i \leq n$, define a function to set the attribute c to the symbol a_i.

$$\text{set}_{\mathtt{c}:=a_i}(m) = \lambda\alpha\,.\,\text{if } \alpha = \mathtt{c} \text{ then } a_i \text{ else } m(\alpha)$$

   c) For $1 \leq i \leq n$, define r_{C_i} to be an APEG rule for each nonterminal C_i to read symbol a_i in Σ_L and put the symbol read into attribute c. r_{C_i} is defined as follows:

$$C_i \leftarrow a_i \ \text{set}_{\mathtt{c}:=a_i}$$

   d) Let $\text{inh}_\emptyset : M \to M$ be an inheritence function that inherits no attributes from the parent context.

$$\text{inh}_\emptyset(m) = \emptyset$$

Note that the $\emptyset$ on the right side of the equality is typed as an attribute context, $M = (\Sigma_\alpha \to \mathbb{T})$, so $\forall \alpha \in \Sigma_\alpha, (\text{inh}_\emptyset(m))(\alpha) \uparrow$.

e) Let $\text{synth}_c : M \times M \to M$ be a synthesis function that synthesizes the attribute c from the nonterminal attribute context to the parent context.

$$\text{synth}_c(m_1, m_2) = \lambda\alpha \,.\, \text{if } \alpha \in \{\text{c}\} \text{ then } m_1(\alpha) \text{ else } m_2(\alpha)$$

f) Let r_C be a nonterminal rule to read the next symbol from the input and put the read symbol into attribute c. r_C is defined as follows:

$$\begin{aligned} C \leftarrow \quad & [C_1, \text{inh}_\emptyset, \text{synth}_c] \\ & /\, [C_2, \text{inh}_\emptyset, \text{synth}_c] \\ & \quad \vdots \\ & /\, [C_n, \text{inh}_\emptyset, \text{synth}_c] \end{aligned}$$

4. Define the start expression for the APEG, which reads the input into an attribute and passes the input string to the function f_L.

a) Let $\text{set}_{\text{s}:=\varepsilon} : M \to M$ be an attribute context modification function which initializes the s attribute's value to the empty string.

$$\text{set}_{\text{s}:=\varepsilon}(m) = \lambda\alpha \,.\, \text{if } \alpha = \text{s} \text{ then } \varepsilon \text{ else } m(\alpha)$$

b) Let $\text{set}_{\texttt{s}:=sc} : M \rightarrow M$ be an attribute context modification function which concatenates the value of attribute c to s.

$$\text{set}_{\texttt{s}:=sc}(m) = \lambda\alpha\,.\,\text{if } \alpha = \texttt{s} \text{ then } \text{concat}(m(s), m(c)) \text{ else } m(\alpha)$$

c) Let $\text{pred}_{f_L(s)} : M \rightarrow M$ be a predicate function which tests a string with the function f_L to determine whether the string is in L.

$$\text{pred}_{f_L(s)}(m) = \text{if } f_L(s) = \texttt{true} \text{ then } m$$

Note that there is no *else* clause in the function, thus when $f_L(s) \neq \texttt{true}$, $\text{pred}_{f_L(s)}(m)\uparrow$.

d) Our start expression is thus:

$$\begin{aligned} e_s = \; & \text{set}_{\texttt{s}:=\varepsilon} \\ & ([C, \text{inh}_{\emptyset}, \text{synth}_c]\, \text{set}_{\texttt{s}:=sc})^* \\ & \text{pred}_{f_L(s)} \end{aligned}$$

5. Define the nonterminal symbols used by the grammar.

$$\Sigma_G = \{C, C_1, \ldots, C_n\}$$

6. Define the rules of the grammar.

$$R = \{r_C, r_{C_1}, \ldots, r_{C_n}\}$$

7. The APEG $G = [\Sigma_L, \Sigma_G, \Sigma_\alpha, \mathbb{T}, R, e_s]$ thus describes language L.

□

This proof is somewhat unhelpful, since it wraps all the language description into the function f_L, which is not described as an APEG, so we have shown little more than an interface between APEGs and other language description mechanisms. While this may be quite helpful, if the only feature of APEGs were that they enable language description in non-APEG forms, then there would appear to be little use for APEGs in the first place. However, we find that even a restricted APEG—one which disallows functions that magically do all the work—is Turing-complete, and thus deserving of the *universal* label.

Theorem 5.2. *Any recursive language can be described by an attributed parsing expression grammar that uses only the concatenate and partition functions.*

Proof. We can construct a restricted APEG that simulates the processing of a Turing machine. The APEG workes by maintaining the state of the Turing machine in four attributes: q, the Turing machine's state; h, the symbol under the machine's head; t, the symbols to the right of the head for the portion of the input that has been read; and p, the symbols to the left of the head (in reverse order). This is similar to the way one might simulate a Turing machine using a two-stack push-down automaton.

1. Define the specifications of the language to be described:

    a) Let L be the recursive language to be described.

b) Let Σ_L be the finite alphabet of L.

c) Let $n = |\Sigma_L|$.

d) Let $a_1, a_2, \ldots, a_n$ be the distinct elements of Σ_L.

2. Define the specifications of a Turing machine that accepts L. We know such a machine exists because L is recursive.

   a) Let M be a Turing machine which accepts by final state recursive language L. $M = [Q, \Sigma_L, \Sigma_M, \delta, q_0, Q_f]$ where:

      - Q is the set of states in M.
      - Σ_M is the tape alphabet of M. $\Sigma_M \supseteq \Sigma_L \cup \{\not{b}\}$ where $\not{b}$ is the tape's blank symbol. $\not{b} \notin \Sigma_L$.
      - $\delta : Q \times \Sigma_M \rightarrow Q \times \Sigma_M \times \{L, R\}$ is the transition function of M. The domain of δ is finite.
      - $q_0 : Q$ is the initial state of M.
      - Q_f is the set of states, $Q_f \subseteq Q$ that are final accepting states. $x \in L$ if and only if M completes its processing in state $q \in Q_f$.

   b) Let $m = |\delta|$, the number of rules in δ.

   c) Let $\rho_1, \rho_2, \ldots, \rho_m$ be the distinct elements of δ.

$$\rho_i = [[q_{\rho_i}, \gamma_{\rho_i}], [q'_{\rho_i}, \gamma'_{\rho_i}, d_{\rho_i}]]$$

where q_{ρ_i} and γ_{ρ_i} are the machine's state and the symbol under the tape head, q'_{ρ_i} is the next state of the machine, γ'_{ρ_i} is the symbol written to the

tape, and $d_{\rho_i} : \{L, R\}$ indicates the direction in which the tape head moves after writing the symbol γ'_{ρ_i}.

3. Define the attributes to be used by the APEG.

   a) Let $\Sigma_\alpha = \{\mathtt{c}, \mathtt{h}, \mathtt{p}, \mathtt{q}, \mathtt{t}\}$, the attribute names to be used by the APEG.

   b) Let $\mathbb{T} = \Sigma_M^* \cup Q$, the possible values of attributes.

4. Define APEG rules to read the next character from the input string and put it in attribute c:

   a) For $1 \leq i \leq n$, define C_i to be a nonterminal symbol corresponding to the terminal symbol a_i in Σ_L.

   b) For $1 \leq i \leq n$, define a function to set the attribute c to the symbol a_i.

$$\text{set}_{\mathtt{c}:=a_i}(m) = \lambda\alpha \,.\, \text{if } \alpha = \mathtt{c} \text{ then } a_i \text{ else } m(\alpha)$$

   c) For $1 \leq i \leq n$, define r_{C_i} to be an APEG rule for each nonterminal C_i to read symbol a_i in Σ_L and put the symbol read into attribute c. r_{C_i} is defined as follows:

$$C_i \leftarrow a_i \ \text{set}_{\mathtt{c}:=a_i}$$

   d) Let $\text{inh}_\emptyset : M \to M$ be an inheritence function that inherits no attributes from the parent context.

$$\text{inh}_\emptyset(m) = \emptyset$$

Note that the $\emptyset$ on the right side of the equality is typed as an attribute context, $M = (\Sigma_\alpha \to \mathbb{T})$, so $\forall \alpha \in \Sigma_\alpha, (\text{inh}_\emptyset(m))(\alpha) \uparrow$.

e) Let $\text{synth}_c : M \times M \to M$ be a synthesis function that synthesizes the attribute c from the nonterminal attribute context to the parent context.

$$\text{synth}_c(m_1, m_2) = \lambda\alpha\,.\,\text{if } \alpha \in \{\texttt{c}\} \text{ then } m_1(\alpha) \text{ else } m_2(\alpha)$$

f) Let r_C be a nonterminal rule to read the next symbol from the input and put the read symbol into attribute c. r_C is defined as follows:

$$\begin{aligned} C \leftarrow \quad & [C_1, \text{inh}_\emptyset, \text{synth}_c] \\ / & [C_2, \text{inh}_\emptyset, \text{synth}_c] \\ & \vdots \\ / & [C_n, \text{inh}_\emptyset, \text{synth}_c] \end{aligned}$$

5. Define APEG rules to modify the attributes containing the Turing machine state, $[q, h, p, t]$, so as to simulate the Turing machine according to its definition:

   a) Define an APEG rule to modify the Turing machine state to effect a move to the right.

      i. The symbol under the tape head (h) is moved to the list of symbols preceding the head (p). Let $\text{set}_{p:=hp} : M \to M$ be an attribute context modification function which prepends the value of attribute h to p.

$$\text{set}_{p:=hp}(m) = \lambda\alpha\,.\,\text{if } \alpha = \texttt{p} \text{ then } \text{concat}(m(\texttt{h}), m(\texttt{p})) \text{ else } m(\alpha)$$

ii. The new symbol under the tape head might already have been read, in which case it is stored in the list of symbols following the head, t. We need a function to test for this. Let $\text{pred}_{t=\varepsilon} : M \to M$ be a predicate function which tests whether attribute t is the empty string.

$$\text{pred}_{t=\varepsilon}(m) = \text{if } m(\mathtt{t}) = \varepsilon \text{ then } m$$

Note that there is no *else* clause in the function, thus when $t \neq \varepsilon$, $\text{pred}_{t=\varepsilon}(m)\uparrow$.

iii. If the Turing machine state contains the symbol to the right of the tape head (t), then the symbol is popped into our tape head attribute (h). Let $\text{set}_{ht:=t} : M \to M$ be an attribute context modification function which pops the first symbol of t into attribute h.

$$\text{set}_{ht:=t}(m) = \lambda\alpha.\; \begin{array}{llll} \text{if} & \alpha = \mathtt{h} & \text{then} & \text{part}_L(m(\mathtt{t}), 1) \\ \text{else if} & \alpha = \mathtt{t} & \text{then} & \text{part}_R(m(\mathtt{t}), 1) \\ \text{else} & & & m(\alpha) \end{array}$$

iv. To read the next symbol from the input and put it into h, we will use nonterminal C. However, C puts the read character into attribute c, so we need a function to synthesize C's c to h. Let $\text{synth}_{h:=c} : M \times M \to M$ be a synthesis function that synthesizes the attribute h into the parent

context from the value of nonterminal attribute c.

$$\text{synth}_{h:=c}(m_1, m_2) = \lambda\alpha\,.\,\text{if } \alpha = \mathtt{h} \text{ then } m_1(\mathtt{c}) \text{ else } m_2(\alpha)$$

v. The Turing machine tape is infinite. If there is no more input to be read, then the next symbol must be a blank. Let $\text{set}_{\mathtt{h}:=\not{b}} : M \rightarrow M$ be an attribute context modification function which sets the value of attribute h to the tape's blank symbol.

$$\text{set}_{\mathtt{h}:=\not{b}}(m) = \lambda\alpha\,.\,\text{if } \alpha = \mathtt{h} \text{ then } \not{b} \text{ else } m(\alpha)$$

vi. Let r_{D_R} be a nonterminal rule to modify the Turing machine state to effect a move to the right. r_{D_R} is defined as follows:

$$\begin{array}{ll} D_R \leftarrow & \text{set}_{p:=hp} \\ & (\text{pred}_{t=\varepsilon} \\ & \quad ([C, \text{inh}_{\emptyset}, \text{synth}_{\mathtt{h}:=c}] \,/\, \text{set}_{\mathtt{h}:=\not{b}})) \\ & \ / \\ & \quad \text{set}_{ht:=t}) \end{array}$$

(To explain, the first line always pushes the current tape symbol onto p, since after a rightward move, the symbol will precede the head. The second line checks to see whether the the next symbol to the right is held in t or not. If so, the *if* will fail that choice, and pop the first symbol of t off to h on line 5. If the next rightward symbol is not in t, then it is

read from the input on line 3, or set to the blank symbol, $\not b$, if there is no more input.)

b) Define an APEG rule to modify the Turing machine state to effect a move to the left.

   i. The symbol under the tape head (h) is moved to the list of symbols following the head (t). Let $\text{set}_{t:=ht} : M \to M$ be an attribute context modification function which prepends the value of attribute h to t.

$$\text{set}_{t:=ht}(m) = \lambda\alpha\,.\,\text{if } \alpha = \texttt{t} \text{ then } \text{concat}(m(\texttt{h}), m(\texttt{t})) \text{ else } m(\alpha)$$

   ii. When the tape head is already at the leftmost position on the tape, $p = \varepsilon$. Attempting to move left from this position is an error, so we need a function to test for this. Let $\text{pred}_{p=\varepsilon} : M \to M$ be a predicate function which tests whether attribute p is the empty string.

$$\text{pred}_{p=\varepsilon}(m) = \text{if } m(\texttt{p}) = \varepsilon \text{ then } m$$

   Note that there is no *else* clause in the function, thus when $p \neq \varepsilon$, $\text{pred}_{p=\varepsilon}(m)\uparrow$.

   iii. The symbols to the left of the tape head are in attribute p. Let $\text{set}_{hp:=p} : M \to M$ be an attribute context modification function which pops the first symbol of p into attribute h.

$$\text{set}_{hp:=p}(m) = \lambda\alpha.\ \begin{array}{lll} \text{if} & \alpha = \mathtt{h} & \text{then} \ \ \text{part}_L(m(\mathtt{p}), 1) \\ \text{else if} & \alpha = \mathtt{p} & \text{then} \ \ \text{part}_R(m(\mathtt{p}), 1) \\ \text{else} & & \qquad\quad\ m(\alpha) \end{array}$$

iv. When the tape head is already at the leftmost position on the tape, the head will run off the left edge of the tape, causing an abnormal failure, which we signal by clearing q. $q = \varepsilon$ will never match a rule in δ, so the parse will complete, but since $\varepsilon \notin Q_f$, it will be a failed parse. We therefore require a function to set q to the empty string.

$$\text{set}_{q:=\varepsilon}(m) = \lambda\alpha\,.\,\text{if } \alpha = \mathtt{q} \text{ then } \varepsilon \text{ else } m(\alpha)$$

v. Let r_{D_L} be a nonterminal rule to modify the Turing machine state to effect a move to the left. r_{D_L} is defined as follows:

$$\begin{array}{ll} D_L \leftarrow & \text{pred}_{p=\varepsilon} \\ & \quad \text{set}_{q:=\varepsilon} \\ & / \\ & \quad \text{set}_{t:=ht}\ \text{set}_{hp:=p} \end{array}$$

c) For each rule in the Turing machine's transition function δ, define an APEG rule to apply it.

i. Define predicate functions for each rule ρ_i in δ which checks the APEG attributes q and h against the conditions for that rule.

$$\text{pred}_{\rho_i} = \text{if } m(\mathtt{q}) = q_{\rho_i} \ \wedge \ m(\mathtt{h}) = \gamma_{\rho_i} \text{ then } m$$

ii. When a Turing machine rule ρ_i is applied, the symbol γ'_{ρ_i} is written to the tape and the machine's state is updated to the new state, q'_{ρ_i}. For each rule ρ_i in δ, define attribute context modification functions which set the values of attributes h and q to γ'_{ρ_i} and q'_{ρ_i} respectively.

$$\begin{aligned} \text{set}_{h:=\gamma'_{\rho_i}}(m) &= \lambda\alpha\,.\,\text{if } \alpha = \texttt{h} \text{ then } \gamma'_{\rho_i} \text{ else } m(\alpha) \\ \text{set}_{q:=q'_{\rho_i}}(m) &= \lambda\alpha\,.\,\text{if } \alpha = \texttt{q} \text{ then } q'_{\rho_i} \text{ else } m(\alpha) \end{aligned}$$

iii. When the Turing machine's head is moved left or right, we shall utilize the nonterminals D_L and D_R defined previously. When we shall therefore require inheritence and synthesis functions to handle the affected attributes, h, p, and t.

$$\begin{aligned} \text{inh}_{hpt}(m) &= \lambda\alpha\,.\,\text{if } \alpha \in \{\texttt{h},\texttt{p},\texttt{t}\} \text{ then } m(\alpha) \\ \text{synth}_{hpt}(m_1, m_2) &= \lambda\alpha\,.\,\text{if } \alpha \in \{\texttt{h},\texttt{p},\texttt{t}\} \text{ then } m_1(\alpha) \text{ else } m_2(\alpha) \end{aligned}$$

iv. For $1 \leq i \leq m$, let B_i be a nonterminal symbol corresponding to the transition rule ρ_i in δ; let r_{B_i} be an APEG rule for each nonterminal in Σ_B to apply a rule in δ. r_{B_i} is defined as follows:

$$B_i \leftarrow \text{pred}_{\rho_i}\ \text{set}_{h:=\gamma'_{\rho_i}}\ \text{set}_{q:=q'_{\rho_i}}\,[D_{d_{\rho_i}}, \text{inh}_{hpt}, \text{synth}_{hpt}]$$

d) Let r_A be a nonterminal rule to apply the correct rule in δ. Only one B_i nonterminal (corresponding to rule $\rho_i \in \delta$) will match. If no nonterminal matches, parsing is complete. r_A is defined as follows:

$$\begin{aligned} A \leftarrow \quad & [B_1, \text{inh}_{hpt}, \text{synth}_{hpt}] \\ & / [B_2, \text{inh}_{hpt}, \text{synth}_{hpt}] \\ & \quad \vdots \\ & / [B_m, \text{inh}_{hpt}, \text{synth}_{hpt}] \end{aligned}$$

6. Define the start expression for the APEG, which both begins the parse and evaluates the final state to determine whether an input string is in L.

   a) Define attribute context modification functions to initialize the state variables q, h, p, and t.

$$\begin{aligned} \text{set}_{q:=q_0}(m) &= \lambda\alpha\,.\,\text{if } \alpha = \mathtt{q} \text{ then } q_0 \text{ else } m(\alpha) \\ \text{set}_{h:=\not{b}}(m) &= \lambda\alpha\,.\,\text{if } \alpha = \mathtt{h} \text{ then } \not{b} \text{ else } m(\alpha) \\ \text{set}_{p:=\varepsilon}(m) &= \lambda\alpha\,.\,\text{if } \alpha = \mathtt{p} \text{ then } \varepsilon \text{ else } m(\alpha) \\ \text{set}_{t:=\varepsilon}(m) &= \lambda\alpha\,.\,\text{if } \alpha = \mathtt{t} \text{ then } \varepsilon \text{ else } m(\alpha) \end{aligned}$$

   b) Define inheritence and synthesis functions for the attributes q, h, p, and t.

$$\begin{aligned} \text{inh}_{qhpt}(m) &= \lambda\alpha\,.\,\text{if } \alpha \in \{\mathtt{q}, \mathtt{h}, \mathtt{p}, \mathtt{t}\} \text{ then } m(\alpha) \\ \text{synth}_{qhpt}(m_1, m_2) &= \lambda\alpha\,.\,\text{if } \alpha \in \{\mathtt{q}, \mathtt{h}, \mathtt{p}, \mathtt{t}\} \text{ then } m_1(\alpha) \text{ else } m_2(\alpha) \end{aligned}$$

c) Define a predicate function to determine whether the state in attribute q is in the set of Turing machine final accepting states, Q_f.

$$\text{pred}_{q \in Q_f} = \text{if } m(\text{q}) \in Q_f \text{ then } m$$

d) Our start expression is thus:

$$e_s = \text{set}_{q:=q_0}\, \text{set}_{h:=\not{b}}\, \text{set}_{p:=\varepsilon}\, \text{set}_{t:=\varepsilon}\, [A, \text{inh}_{qhpt}, \text{synth}_{qhpt}]^* \, \text{pred}_{q \in Q_f}$$

7. Define the nonterminal symbols used by the grammar.

$$\Sigma_G = \{C, D_R, D_L, A, C_1, \ldots, C_n, B_1, \ldots, B_m\}$$

8. Define the rules of the grammar.

$$R = \{r_C, r_{D_R}, r_{D_L}, r_A, r_{C_1}, \ldots, r_{C_n}, r_{B_1}, \ldots, r_{B_m}\}$$

9. The APEG $G = [\Sigma_L, \Sigma_G, \Sigma_\alpha, \mathbb{T}, R, e_s]$ thus describes language L.

□

5.2 Form of Output

How are attributes recorded in the output stream and parse tree? We have several choices:

1. The entire attribute context (all attributes and their values) are output into the parse tree after every step which might modify the attribute context: at the begining or end of any nonterminal, or any attribute usage function.
2. Only attributes which might have changed in a nonterminal or attribute usage function are output to the parse tree.
3. In a concrete APEG syntax, provide some sort of output clause which when encountered during a parse will cause the atribute context (or a subset of the attribute context) to be output to the parse tree. This has the advantage of giving the grammar writer more control over the insertion of attributes into the parse tree, but it also may cause certain parse information to be lost if the grammar never outputs some attributes. That is not necessarily a bad thing.

This decision is best left to later, when the details of a concrete APEG syntax are developed.

5.3 Time and Efficiency

An APEG can be constructed to simulate any Turing machine. The APEG runs under the same time bounds as the Turing machine, and since it is believed that a Turing machine can be constructed to simulate any computation, it follows that an APEG can be constructed for any language that runs with the best possible time bounds.

In practice, however, it is unlikely that an APEG would be constructed to simulate a Turing machine, but rather that the expressiveness of APEGs would be used

to define the language independent of the design of a Turing machine intended to parse the language. It is difficult to generalize in such cases, however, as to how much time the APEG might take to parse an input string in a language, since it depends on the approach taken. It is possible that a grammar designer might make trade-offs in time efficiency in favor of clarity of the grammar, but it is comforting to know that if time efficiency is of paramount importance, an APEG can be constructed with the best possible characteristics.

This kind of analysis does not lend itself nicely to general statements like, “APEGs can be parsed in linear time,” as unattributed PEGs can be, nor statements like, “APEGs can be parsed in polynomial time,” as can CFG’s. On the other hand, APEGs are much more expressive than either of those grammar models, capable of describing infinitely many more languages. Furthermore, it is correct to make general statements like, “any language that can be described by an unattributed PEG can be described by an APEG, and a parser for such an APEG can be built that works in linear time.” A similar statement can be made for context-free languages described by an APEG (i.e., that an APEG-based parser can be built which parses in polynomial time.

5.4 Memory Usage

Memory usage is more difficult to analyze, and no completely rigorous analysis is really possible, since there are multiple ways to describe a language in an APEG. Turing machines do not have memory limitations, yet an APEG-based Turing machine

simulation does consume an amount of memory in proportion to its furthest position from the tape start. A poorly designed APEG, therefore, could use an infinite amount of memory and never complete. However, since the simulated Turing machine tape head can move only at a finite speed, the time efficiency bounds should also describe an upper limit on memory usage.

5.5 Fault Tolerance

Attributes can be used to record error conditions in the parsing (for instance, if a value is out of range, but the parse is continued). There is no standard way to propagate a log of such errors, however, so such handling is up to the grammar writer, or a standard might be constructed at a future time when a more concrete syntax and implementation is developed for APEGs.

5.6 Ease of Use

This is the most difficult quality to measure. The ease of use of the grammar is highly dependent on the concrete syntax of the APEG model used. For instance, an APEG syntax which provides attribute set, predicate, inherit, and synthesize functions is going to be much easier to use than one which does not, which requires the use of lambda expressions everywhere such a function is required. We examine in this section a possible concrete syntax.

5.6.1 Toward a Concrete Syntax

One possible way to make the APEG grammar model into a usable language is to do away with the explicit inheritence, synthesis, and attribute modification functions, and imply them with constructs that perform the same operations. For instance, setting an attribute s to a value which is a function of the attribute context, $f(m)$, may be written simply $\{s := f\}$. For example, we might write

$$\{s := \text{concat}(s, u)\}$$

instead of the more cumbersome

$$\lambda m\alpha\,.\,\text{if } \alpha = \texttt{s} \text{ then } \text{concat}(m(\texttt{s}), m(\texttt{u})) \text{ else } m(\alpha)$$

The former expression is likely easier to use than the latter. Predicates may likewise be simplified thus:

$$\{\text{check } s = t\}$$

instead of

$$\lambda m\,.\,\text{if } m(\texttt{s}) = m(\texttt{t}) \text{ then } m$$

Inheritence functions may be replaced by adding arguments to our nonterminals. The definition of a nonterminal, then, will specify its interface, the attributes it expects to be inherited. For example, if our nonterminal D requires inherited values

for the attributes s, t, and u, then we would write our rule for D as

$$D(s,t,u) \leftarrow \cdots$$

When an APEG uses this nonterminal, instead of referencing it with the triple

$$[D, \text{inheritence function}, \text{synthesis function}]$$

we need only the pair

$$[D(f_s, f_t, f_u), \text{synthesis function}]$$

where the arguments to the nonterminal are functions of the attribute context, as we saw on the right-hand side of our improved set syntax. We might therefore invoke nonterminal D with syntax like the following:

$$[D(\text{concat}(s,u), \varepsilon, \mathsf{a}), f_{\text{synth}}]$$

and the parser would understand that the inheritence function is:

$$\{s := \text{concat}(s,u); t := \varepsilon; u := \mathsf{a}\}$$

As we see, inheritence fills the role of the arguments in a function call. In comparison, synthesis fills the role of function return values. We might therefore also add

the attributes we expect to synthesize into our nonterminal "signature", so that it now consists not only of the inherited attributes but also the synthesized attributes. Let us add the synthesized attributes to the nonterminal definition. For example, the nonterminal defined as

$$A(a,b)(c,d) \leftarrow \cdots$$

inherits values for its attributes a and b, and synthesizes two values, which will be the values of its attributes c and d after completing. This nonterminal could thus be invoked

$$A(\text{concat}(x,y),\varepsilon)(s,t)$$

This is only one possible way to handle attribute synthesis (and inheritence), but we shall use it in our example. Future work should be directed at improving upon this rough outline of a concrete syntax.

Example 5.3. We shall attempt to see how this syntax will work with a real example. We shall use the example of the X12 interchange header segment. The X12 interchange header record is both a fixed-width and delimited record: it is delimited, but all fields have a fixed width. It is also self-defining in terms of the delimiters it uses; the characters in positions 4, 105, and 106 determine the delimiters for the entire interchange (although only the element separator at position 4 is used subsequently in the interchange header itself). The rules for the interchange segment are as follows.

1. The first three characters must be the string "ISA".

2. The fourth character defines the element separator, which must also appear at positions 7, 18, 21, 32, 35, 51, 54, 70, 77, 82, 84, 90, 100, 102, and 104.
3. The character appearing at position 105 defines the component element separator and may not appear anywhere else in the interchange header.
4. The character appearing at position 106 defines the component segment terminator and may not appear anywhere else in the interchange header.

Our syntax for the interchange header segment is shown in Figure 5.1. There are several ways this could have been written; in this example, the segment is read with each element placed into its corresponding attribute. After the segment has been read, checking is performed to ensure that the delimiters only appear where they are supposed to.

Admittedly, it is the author's opinion that the syntax is not ideal, and that it could use some improvement. However, the example does describe the X12 interchange header segment, and the example serves to show that an APEG-based language description need not look like the construction in Example 4.9. The design of a good DDL based on attributed parsing expression grammars is an open question left for future research.

$$
\begin{array}{lll}
S(l)(s) & \leftarrow & \{\ \text{check } l > 0\ \} \\
 & & \quad S(l-1)(s) \\
 & & \quad \{\ s := \text{concat}(s, \cdot)\ \} \qquad \text{“$\cdot$” means read the next character from input.} \\
 & & /\ \{\ s := \varepsilon\ \}
\end{array}
$$

$$
\begin{array}{lll}
T(s)(R) & \leftarrow & \{\ \text{check } |s| > 0\ \} \\
 & & \quad T(\text{part}_R(s,1))(R) \\
 & & \quad \{\ R := R \cup \text{part}_L(s,1)\ \} \\
 & & /\ \{\ R := \emptyset\ \}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{G_{ISA}()(e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}, e_{13}, e_{14}, e_{15}, e_{16}, d_1, d_2, d_3)} \\
 & \leftarrow & \texttt{I S A} \\
 & & S(1)(d_1) \qquad d_1 \text{ is element separator.} \\
 & & S(2)(e_1)\ d_1\ S(10)(e_2)\ d_1 \\
 & & S(2)(e_3)\ d_1\ S(10)(e_4)\ d_1 \\
 & & S(2)(e_5)\ d_1\ S(15)(e_6)\ d_1 \\
 & & S(2)(e_7)\ d_1\ S(15)(e_8)\ d_1 \\
 & & S(6)(e_9)\ d_1\ S(4)(e_{10})\ d_1 \\
 & & S(1)(e_{11})\ d_1\ S(5)(e_{12})\ d_1 \\
 & & S(9)(e_{13})\ d_1\ S(1)(e_{14})\ d_1 \\
 & & S(1)(e_{15})\ d_1\ S(1)(e_{16}) \\
 & & \{\ d_2 := e_{16}\ \} \qquad d_2 \text{ is component element separator.} \\
 & & S(1)(d_3) \qquad d_3 \text{ is segment terminator.} \\
 & & \{\ \text{check } d_1 \neq d_2\ \} \\
 & & \{\ \text{check } d_1 \neq d_3\ \} \\
 & & \{\ \text{check } d_2 \neq d_3\ \} \\
 & & T(\text{concat}(\text{“ISA”}, e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}, e_{13}, e_{14}, e_{15}))(R) \\
 & & \{\ \text{check } d_1 \notin R\ \} \\
 & & \{\ \text{check } d_2 \notin R\ \} \\
 & & \{\ \text{check } d_3 \notin R\ \}
\end{array}
$$

Figure 5.1. A Possible Concrete Syntax for the X12 Interchange Header Segment. The nonterminal S reads a string of length l from the input and returns it as its attribute. Nonterminal T is used to create a set consisting of the characters in its inherited attribute s. Nonterminal G_{ISA} parses the X12 interchange header segment from the input, returning the values of the sixteen elements and three delimiters as synthesized attributes.

CHAPTER 6

CONCLUSIONS

The goal is to develop a data description language that is general enough to describe all data languages in an intuitive way and is practical. A practical data description language is one that permits a parser to be built that works, producing timely output in a usable form. The specification of a universal data description language would likely not only specify the language of the actual data description, but also provide languages for addressing and transforming the models of UDDL-parsed documents. Such a universal data description language would be a boon to electronic data interchange and commerce, allowing for the exchange of data formats in a parser-ready format, and allowing standard tools to be developed to work with the resulting document models.

The basis of any data description language is a grammar model. Once a suitable grammar model has been developed, a data description language is formed by putting a concrete syntax to the theory. A given grammar model may have multiple concrete syntax implementations. We have defined a new grammar model, combining parsing expression grammars, which are efficient to parse, with attributes, which give greater defining "power" to the grammars. The resulting grammar model is Turing-complete,

and thus is capable of describing any data language. It is therefore a viable candidate to be the basis for a universal data description language.

6.1 Suggestions For Further Research

The attributed parsing expression grammar model is not itself a data description language, but we can use it as the theoretical basis for a UDDL. The next step in this research is the creation of a concrete syntax for the attributed parsing expression grammar model. The main challenge in developing APEG-based DDLs is finding an intuitive way to handle the functions that manipulate attributes and attribute contexts. One possibility is proposed, which treats inheritence functions as function inputs, synthesis functions as return values, and provides a syntax for predicates and the modification of the attribute context by setting attribute values. Several candidates may be explored before settling on a "best" DDL implementation. It might, for instance, be more intuitive to have multiple parsing expressions for a nonterminal, with the choice of which one to use determined (deterministically) by pattern matching on the inherited context. Finding a usable concrete syntax is the next problem for further research into APEGs.

6.2 Suggestions For Related Research

Finding a practical universal data description language is a problem of such great importance, that we ought not restrict ourselves only to the APEG branch of research. At the end of Chapter 3, several other possibilities were suggested. Three other gram-

mar models were recommended as promising bases for a UDDL, and different ways of extending those grammar models might prove as successful (or more successful) that attributed PEGs. In addition, wholly new grammar models might be created, and some ideas for starting points for those are also suggested in Chapter 3.

REFERENCES

REFERENCES

Aho, A. V., Sethi, R., & Ullman, J. D. (1986). *Compilers: Principles, techniques, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Aho, A. V., & Ullman, J. D. (1972). *The theory of parsing, translation, and compiling* (Vol. 1: Parsing). Englewood Cliffs, NJ, USA: Prentice Hall.

Back, G. (2002). Datascript—a specification and scripting language for binary data. In *GPCE '02: The ACM SIGPLAN/SIGSOFT conference on generative programming and component engineering* (pp. 66–77). London, UK: Springer-Verlag.

Chomsky, N. (1956, September). Three models for the description of language. *Institute of Radio Engineers Transactions on Information Theory*, *2*(3), 113–124.

Earley, J. (1970, February). An efficient context-free parsing algorithm. *Communications of the ACM*, *13*(2), 94–102.

Fisher, K., & Gruber, R. (2005). PADS: A domain-specific language for processing ad hoc data. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation, Chicago, IL, USA* (pp. 295–304). New York, NY, USA: ACM Press.

Ford, B. (2004). Parsing expression grammars: A recognition-based syntactic foundation. In *Symposium on principles of programming languages.* January 14–16, Venice, Italy: ACM.

Grune, D., & Jacobs, C. (1990). *Parsing techniques: A practical guide*. Chichester, West Sussex, England: Ellis Horwood Limited.

Johnson, S. C. (1975). *Yacc: Yet another compiler compiler* (Computing Science Technical Report No. 32). Murray Hill, NJ, USA: AT&T Bell Laboratories.

Klint, P., Lämmel, R., & Verhoef, C. (2005, July). Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology*, *14*(3), 331–380.

Knuth, D. E. (1968, June). Semantics of context-free languages. *Theory of Computing Systems*, *2*(2), 127–145.

Koster, C. H. A. (1971). Affix grammars. In J. E. L. Peck (Ed.), *ALGOL 68 implementation* (pp. 95–109). Amsterdam: North-Holland Publ. Co.

McCann, P. J., & Chandra, S. (2000). Packet types: abstract specification of network protocol messages. In *SIGCOMM '00: Proceedings of the conference on applications, technologies, architectures, and protocols for computer communication, Stockholm, Sweden* (pp. 321–333). New York, NY, USA: ACM Press.

McGee, W. C. (1972). Some current issues in data description. In *Proceedings of 1972 ACM-SIGFIDET workshop on data description, access and control* (pp. 1–12). New York, NY, USA: Association of Computing Machinery.

Ruschitzka, M., & Clevenger, J. L. (1989). Heterogeneous data translations based on environment grammars. *IEEE Trans. Softw. Eng.*, *15*(10), 1236–1251.

Sibley, E. H., & Taylor, R. W. (1973). A data definition and mapping language. *Communications of the ACM*, *16*(12), 750–759.

Sudkamp, T. A. (1997). *Languages and machines: An introduction to the theory of computer science* (2nd ed.). Reading, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

van Wijngaarden, A., Mailloux, B., Peck, J. E. L., Koster, C. H. A., Sintzoff, M., Lindsey, C. H., et al. (1975, March). Revised report on the algorithmic language algol 68. *Acta Informatica*, *5*(1–3), 1–236.

Vogt, H. H., Swierstra, S. D., & Kuiper, M. F. (1989). Higher order attribute grammars. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 conference on programming language design and implementation* (pp. 131–145). New York, NY, USA: ACM Press.

Wall, L. (1998). Foreword. In T. Christiansen & N. Torkington (Eds.), *Perl cookbook* (pp. xxi–xxii). Sebastopol, CA, USA: O'Reilly & Associates, Inc.

BIOGRAPHICAL SKETCH

BIOGRAPHICAL SKETCH

David Boyd Mercer was born in Melbourne, Australia on November 19, 1970. He graduated from the Massachusetts Institute of Technology, Cambridge, Massachusetts, with an S.B. (Bachelor of Science) in Electrical Engineering and Computer Science in 1992. In 2005, he was awarded the University of South Alabama School of Computer and Information Sciences Outstanding Graduate Fellowship.

David has sixteen years professional experience working with data. He is Certified in Production and Inventory Management (CPIM), and has worked as a Senior Software Engineer for Oracle Corporation in Redwood Shores, California, and Director of Data Management Services for Digital Impact, in San Mateo, California. He is currently the Manager for Research and Development at The SSI Group in Mobile, Alabama.

David is married with two children. He and his family reside in Spanish Fort, Alabama, where they attend Spanish Fort Presbyterian Church.