Semantics and Algorithms for Data-dependent Grammars

Trevor Jim
AT&T Labs-Research
tjim@research.att.com

Yitzhak Mandelbaum
AT&T Labs-Research
yitzhak@research.att.com

David Walker
Princeton University
dpw@cs.princeton.edu

Abstract

Traditional parser generation technologies are incapable of handling the demands of modern programmers. In this paper, we present the design and theory of a new parsing engine, YAKKER, capable of handling the requirements of modern applications including (1) full scannerless context-free grammars with (2) regular expressions as right-hand sides for defining nonterminals. YAKKER also includes (3) facilities for binding variables to intermediate parse results and (4) using such bindings within arbitrary constraints to control parsing. These facilities allow the kind of data-dependent parsing commonly needed in systems applications, particularly those that operate over binary data. In addition, (5) nonterminals may be parameterized by arbitrary values, which gives the system good modularity and abstraction properties in the presence of data-dependent parsing. Finally, (6) legacy parsing libraries, such as sophisticated libraries for dates and times, may be directly incorporated into parser specifications. We illustrate the importance and utility of this rich format specification language by presenting its use on examples ranging from difficult programming language grammars to web server logs to binary data specification. We also show that our grammars have important compositionality properties and explain why such properties are important in modern applications such as automatic grammar induction.

In terms of technical contributions, we provide a traditional high-level semantics for our new grammar formalization and show how to compile these grammars into nondeterministic automata. These automata are stack-based, somewhat like conventional pushdown automata, but are also equipped with environments to track data-dependent parsing state. We prove the correctness of our translation of data-dependent grammars into these new automata and then show how to implement the automata efficiently using a variation of Earley's parsing algorithm.

1. Introduction

The study of parsing is one of the oldest and most intellectually satisfying areas of programming languages, stretching back decades to the dawn of computer science. The best work in this field has nearly universal practical application and yet is based around remarkably elegant and general automaton theories. More recently, however, the study of parsing has come to be viewed as a somewhat boring, largely solved problem. To be honest, using the word *parsing* in the introduction of a POPL paper is a bit of a dicey move.

Perhaps one reason the study of parsing in programming languages circles may have gone out of vogue is that widely-used tools such as YACC [16] hit a sweet spot in the expressiveness-performance tradeoff space in the 1970s. YACC and its relatives were based around the LR fragment of context-free grammars and hence were powerful enough to express the syntax of many programming languages (with only the occasional egregious hack), and yet also gave linear-time performance. Consequently, in an era when computational resources were less plentiful than in modern times, programmer convenience and expressiveness were sacrificed for performance. Since then, for the most part, PL researchers have hung on to legacy tools because they are well-known, well-supported, taught in school and universally available, rather than because they are optimally designed.

On the other hand, programmers *outside* the minuscule world of PL implementors almost never use parser generators. Despite the fact that they are constantly writing parsers—for data formats, networking protocols, configuration files, web scraping and small domain-specific languages—they do most of it by hand, often using regular expression libraries that support context-sensitive features like backreferencing and POSIX anchors. This is not because they are unaware of PL tools, but rather because these tools do not solve their problems. For example, when Rescorla implemented the SSL message data format, he "attempted to write a grammar-driven parser (using YACC) for the language with the aim of mechanically generating a decoder, but abandoned the project in frustration" [27, p. 68]. Another highly-visible project, HTML 5.0 [13], has abandoned the grammar formulation of previous versions and is essentially specifying the syntax by an imperative program!

Hence, in order to serve programming language researchers better, and, more importantly, the legions of other programmers who actually write most of the parser code in the world, we need substantial improvements in the power and flexibility of parser generation technology.

1.1 Towards full context-free grammars

There have been a number of recent efforts to build parser generators that support unrestricted context-free grammars [24, 32, 30, 6]. In particular, McPeak has made a number of convincing arguments for abandoning the constraints of LR [23]. From a theoretical perspective, one central problem with LR is that it has poor compositionality properties. One cannot necessarily define a first sublanguage A, then a second sublanguage B and take their union: $S = A \mid B$. If A and B overlap then S is ambiguous and is not LR. Moreover, in order to create a language S containing all of A and B, one must take the union of A with the asymmetric difference B/A (or vice versa: take B with the asymmetric difference A/B). Unfortunately, LR is not closed under set difference [24] so unifying the two languages by finding the difference could be impossible! Even when the set difference remains LR, finding the appropriate way to express it requires digging in to and adjusting the definitions of A and/or B—these sublanguages cannot be developed independently in a modular fashion.

In practice, working within the constraints of LR can be very difficult, as anyone who has attempted to debug a shift-reduce or shift-shift conflict can attest. Debugging such conflicts is not only difficult for the initial programmer, but often results in grammars that are harder to read, understand and maintain. Consequently, the grammars lose a great deal of their benefit as documentation. Moreover, not all practical grammars are amenable to such techniques. For instance, C and C++ contain a number of inherent ambiguities in their grammars. One troublesome example is that there is no way to determine whether `(a)&(b)` is a bit-wise conjunction or if it is a cast of `&(b)` to the type `(a)`, without knowing whether `a` is a variable name or a type name. Hence, attempting to parse C using an LR-based parser is only possible by stepping completely outside the grammar specification mechanism and having the parser communicate dynamic information about new type names back to the lexer. In a full context-free grammar specification mechanism, such hacks can elegantly be avoided simply by allowing ambiguities at parse time and disambiguating during semantic analysis [23].

1.2 Beyond context-free grammars

A robust parser generator for full context-free grammars may serve programming languages researchers well, but if we look beyond our community, there is a huge market for substantially more sophisticated parsing technology.

For example, in the web space, there is no more common task than matching identical XML or HTML tags, but, of course, this is not a context-free task. In binary formats, it is extremely common to use data fields that specify the lengths or sizes of other data fields, another non-context-free feature. In addition, the specifications of many systems data formats are made simpler by the presence of constraints, such as bounds on integer ranges, expressed as arbitrary predicates over parsed data. In summary, without moving aggressively beyond the bounds of context-free grammars, parser generators are nearly useless to systems developers, networking engineers and web programmers.

One final limiting factor of standard parser generators is the inability to incorporate useful legacy code and libraries into parser definitions. For instance, there are over a hundred commonly used date and time formats and consequently programmers have developed sophisticated libraries for dealing with them. It is extremely useful for programmers to be able to incorporate such libraries directly into their parser specifications—after all, handling dates and times correctly might well be the most difficult task in parsing some server log. If programmers cannot incorporate their favourite libraries, they may well just reject the parser specification mechanism all together.

1.3 YAKKER: A general solution to modern parsing problems

In this paper, we present the theory and design of a new, general-purpose parsing engine called YAKKER. YAKKER is designed to be general enough to solve the parsing problems of the modern world, both inside and outside the programming languages community. In particular, it combines all of the following key elements in one universal platform:

- *Full, scannerless context-free grammars.* We support unrestricted context-free grammars, making it easier to express the syntax of programming languages like C++. Scannerless parsers reduce the burden on both parser-generator users and implementors by avoiding the need for a separate lexing tool and description language [28], and they make it easier for users to combine sublanguages with different tokens or whitespace conventions [1].
- *Regular right sides.* YAKKER uses regular expressions over terminals and nonterminals for the right-hand sides of nonterminal definitions. Regular right sides allow for very concise definitions in many situations. YAKKER handles regular right sides directly, rather than desugaring them into a more restricted form (an inefficient translation).
- *Variable binding.* Our grammars allow variables to be bound to data read earlier in the parse and used to direct parsing that comes later.
- *Data-dependent constraints.* Our grammars can include arbitrary predicates over parsed data. When combined with our variable binding functionality, this feature allows for easy expression of length fields and other data-dependent parsing idioms.
- *Parameterized nonterminals.* Parameterized nonterminals allow users to create libraries of convenient abstractions that are easily customized and highly reusable.
- *Inclusion of parser libraries.* Arbitrary parser libraries can be included in grammar specifications as black boxes, so legacy parser libraries can be reused.

The central technical contribution of this paper is a comprehensive theory that explains how to combine this powerful set of features into a single grammar formalism and to implement that grammar formalism correctly. The implementation strategy is carried out by translating our grammars into a new kind of automaton, the *data-dependent automaton* (DDA). We give a direct, nondeterministic operational semantics to the data-dependent automaton and prove that it implements our grammars correctly. Like its brethren, the finite automaton and the pushdown automaton, our new automaton is an elegant, powerful abstraction in its own right and a useful implementation-level tool amenable to formal analysis, transformation and optimization. The last piece of our theory is to show how to implement the nondeterministic semantics efficiently by extending the traditional Earley parsing algorithm [5]. We prove our parsing algorithm correct with respect to the nondeterministic automaton semantics. By transitivity, we have a proof that it implements our high-level grammar specifications correctly as well.

In addition to these theoretical contributions, we illustrate the importance, utility and generality of our new design with example grammar fragments that specify a number of different sorts of language paradigms. These paradigms are drawn from a wide variety of domains ranging from complex programming language syntax to widely-used systems logs to networking protocols. We have also implemented a prototype of our system which is able to compile and run all of these examples.

The rest of the paper is structured as follows. In Section 2, we introduce the features of YAKKER by example. Then, in Sections 3 and 4, we present the syntax and semantics of grammars and data-dependent automata, respectively. Section 5 contains a proof that an automaton soundly and completely implements a grammar, assuming that a small set of conditions relating the grammar and automaton are met. The section concludes with a straightforward compilation of grammars into automata along with a proof that the resulting automata satisfy the specified conditions (thus guaranteeing the correctness of the compilation). Then, in Section 6, we present an Earley-style algorithm for parsing with data-dependent automata and prove its correctness. In Section 7 we discuss related work, and then conclude in Section 8.

2. YAKKER by example

We now illustrate the key features of YAKKER through examples.

Regular right sides In YAKKER, nonterminals are defined by regular expressions over the terminals and nonterminals of the grammar. For example, here we define two nonterminals:

$$\begin{aligned} \text{digit} &= \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9} \\ \text{number} &= \text{digit digit}^* \end{aligned}$$

The first nonterminal, digit, is defined as an alternation of terminals (in bold), and it denotes the set of ASCII numerical characters. The second, number, is defined in terms of digit using concatenation and Kleene-closure. This familiar notation is standard for lexer generators, but uncommon in parser generators. We allow it for all language constructs. Thus a list of expressions can be written

```
expr , expr (, expr)*
```

Parser generators that do not support regular right sides force programmers to define the list as a separate nonterminal, as in this excerpt from the OCAML implementation [20]:

```
expr_comma_list : expr_comma_list COMMA expr
                | expr COMMA expr
```

This is more verbose and less readable than the equivalent regular right side, a fact emphasized by the use of regular right sides to describe OCAML syntax in the user manual.

When implemented properly, regular right sides are also efficient. Any grammar with regular right sides can be transformed into a grammar without them, however, this will add many nonterminals to the language, and thus a great deal of extra structure to parse trees. Regular expressions are highly ambiguous, e.g., the expression $(\mathbf{x}^*)(\mathbf{x}^*)$ can be matched against an input of length n in $n+1$ ways. Compiling away regular right sides therefore results in highly ambiguous grammars, which are more expensive to parse. YAKKER handles regular right sides directly and efficiently, ignoring their ambiguities via determinization, as is traditional with regular expressions (cf Section 4).

Full context-free grammars We support context-free grammars without restriction, including ambiguous grammars. For example, the problem we mentioned in the introduction involving C's syntax can be avoided simply by using an ambiguous grammar:

```
typename = identifier
    expr = & expr                  ; address-of
         | expr & expr             ; bitwise conjunction
         | ( typename ) expr       ; cast
         | ( expr )
         | identifier
         | ...
```

(The grammar is ambiguious because typenames and identifiers are identical.) Of course, ambiguities must be resolved at some point. One way would be to resolve them during the typechecking phase of the compiler, when all type names are available [23]; we give another way later in this section.

Outside of programming languages, ambiguous grammars are often used as documentation. For example, many network protocol message formats are specified using grammars (in IETF Request For Comments, or RFCs), and we have found that these grammars are almost invariably ambiguous.

Attribute-directed parsing Nonterminals can capture input substrings which the programmer can use to direct the parser's behaviour on the remainder of the input. This is commonly needed when parsing systems formats, such as network protocol messages. For example, protocols that need to transfer binary data often do so using messages consisting of the length of the data followed by the data. In the IMAP mail protocol [4] these are called *literals*, and their syntax is specified in the RFC [25] as follows:

```
literal8 = "~{" number ["+"] "}" CRLF *OCTET
       ;; A string that might contain NULs.
       ;; <number> represents the number of OCTETs
       ;; in the response string.
```

Here number is an ASCII representation of the length of the data, CRLF is a carriage return and linefeed, and OCTET is any 8-bit value. We can express the syntax of an IMAP literal directly using *bindings* and *constraints*:

```
literal8 = ~ { x=number (+ | ϵ) } {n=string2int(x)} CRLF
               ([n > 0] OCTET {n = n − 1})* [n = 0]
```

Here we bind the string parsed by number to a variable x, convert x to a number n and use constraints (expressions within square brackets) and value binding (assignments in braces) to express the RFC's side condition on the length directly in the grammar.

One way to implement this specification would be to evaluate these bindings and constraints in a semantic analysis following parsing, as we suggested for the C example, above. However, this would be very inefficient—our parser would have to treat the length abstractly, potentially promoting all possible lengths to its output. Instead, we evaluate constraints *during* parsing to prune ambiguities as soon as possible. We call this *attribute-directed parsing*, following Watt [33]. Notably, attribute-directed parsing subsumes backreferences, which are commonly used in hand-written parsers based on regular expression libraries.

Parameterized Nonterminals The length+data idiom of IMAP literals is used in many systems formats. We have parameterized nonterminals to support modular reuse of such idioms. For example, we could define a fixed-width string nonterminal:

```
stringFW(n) = ([n > 0] CHAR8 {n = n − 1})* [n = 0]
```

This is an imperative definition, relying on the assignment n $=$ n $-$ 1. We could alternatively use a functional style:

```
stringFW(n) = [n = 0] | [n > 0] CHAR8 stringFW(n − 1)
```

(The [n $=$ 0] case parses the empty string.) Our formalism, in principle, would allow an optimizing parser generator to convert tail calls of this form into Kleene-closures of the previous form.

Parameters can be used to thread parsing state through a parse to aid attribute-directed parsing. For example, ambiguities in the C grammar can be pruned at parse time by passing a table of type identifiers to the expr nonterminal:

```
    decls(s) = typedef type-expr x=identifier
                   decls(insert(s.types,x))
             | ...
 typename(s) = x=identifier [ member(s.types,x) ]
     expr(s) = & expr(s)                ; address-of
             | expr(s) & expr(s)        ; bitwise conjunction
             | ( typename(s) ) expr(s)  ; cast
             | ( expr(s) )
             | x=identifier [ not(member(s.types,x)) ]
             | ...
```

Scannerless parsing YAKKER does not divide parsing into separate scanning (lexing) and parsing phases. Many data formats use hand-written parsers and were not designed with a two-phase parsing strategy in mind. It would be difficult to write two-phase parsers for these formats. For example, we have found that many of the formats defined in RFCs using grammars require context-dependent tokenization.

Scannerless parsers are often also useful in parsing programming languages. One example is Javascript, which sometimes allows semicolons to be omitted at the end of a line, e.g., after a void **return** statement. Here is a simplification that illustrates how we can handle this in YAKKER:

```
statement = return (SP | TAB)* (LF | ; | identifier ;)
```

Another example is the use of indentation for block structure, as in Python and Haskell.[1] This is context-dependent whitespace: it must be handled specially by the lexer if it is at the beginning of a line within a block, and the block structure is only known by the parser. This forces the lexer and parser to be mutually recursive.

[1] This formatting is known as the "offside rule," and was first formulated by Peter Landin [19].

For a final example, consider a template language like PHP, which mixes the syntax of two languages, HTML and the PHP control language. The languages have different keywords and comment syntax, which again leads to context-dependent lexing.

Blackbox parsers Support for the full range of context free grammars (and beyond) allows us to seamlessly integrate support for foreign parsers, or *blackboxes*, into our formalism, which can be essential for parsing real-world languages and data formats. For example, consider these two sample lines from a web server log, which include a complex date field (the lines are broken with a "\" to fit).

```
207.136.97.49 - - [15/Oct/1997:18:46:51 -0700] \
  "GET /tk/p.txt HTTP/1.0" 200 30
tj62.aol.com - - [16/Oct/1997:14:32:22 -0700] \
  "POST /scpt/dd@grp.org/confirm HTTP/1.0" 200 941
```

Here is a grammar fragment describing this data, where the date field is described with a blackbox nonterminal, bbdate. SP is a nonterminal defined as the space character; we omit the definitions of the other nonterminals.

blackbox bbdate
entry = client SP auth-id SP auth-id SP **[** bbdate **]**
SP request SP response SP length

In our formalism, blackbox nonterminals come from a different namespace than other nonterminals. We indicate this here with the "**blackbox** bbdate" declaration.

Compositionality Our grammars are closed under union, concatenation, and Kleene closure. We learned about the importance of these compositionality properties from our experience with the PADS family of languages [7, 8, 22]. PADS is a domain-specific language that interprets specialized type declarations as grammars. These type declarations can be used to generate parsers and printers as well as a variety of useful end-to-end data processing tools. PADS grammars are superficially similar to the grammars presented in this paper—they contain regular expressions, variable binding, constraints, parameterized definitions and blackbox parsers. However, PADS grammars have a different semantics that makes them easy and efficient to implement using a top-down, recursive descent parsing algorithm with limited backtracking. In particular, if, when parsing the union $(A \mid B)$, PADS succeeds in parsing an A, it will commit to that choice and never backtrack, even when downstream errors arise that could be avoided if the input was interpreted as a B. PEGS, another related grammar specification language analyzed by Ford [10, 11], and implemented efficiently by Grimm [12], has similar expressive power and semantics.

Unfortunately, the PADS semantics has undesirable consequences in certain applications because it causes closure under union and concatenation to fail:[2]

- $L(A \mid B) \neq L(A) \cup L(B)$
- $L(AB) \neq L(A)\ L(B)$

Both of these principles are required for the correct functioning of divide-and-conquer grammar induction algorithms, including algorithms designed to infer PADS descriptions [9]. The PADS grammar induction algorithms attempt to avoid learning incorrect grammars by using various heuristics, but the heuristics are not always successful—the algorithms do fail occasionally on real data sources.

More generally, these principles are essential for modular grammar design. If they hold, programmers can develop sublanguages A and B in isolation and then, without further modification, perform natural operations such as union or concatenation and receive the expected resulting semantics.

3. Grammars and languages

We now specify the formal syntax and semantics of grammars. We assume a simple, untyped language of expressions (e) that includes variables (x, y, z, etc.), boolean values $true$ and $false$, the unit value $()$, and string values (sequences of terminals) w. An environment $E ::= \cdot \mid E[x = v]$ is a finite partial map from variables to values. We write E, E' for the concatenation of two environments. Bindings to the right take precedence over bindings to the left (an update semantics). We write $[\![e]\!]E$ to denote the value v that results from evaluating expression e in environment E.

Definition 1
A *grammar* G is a tuple $(\Sigma, \Delta, \Phi, A_0, \mathcal{R})$ where

- Σ is a finite set of terminals;
- Δ is a finite set of nonterminals;
- Φ is a finite set of blackboxes;
- $A_0 \in \Delta$ is the start nonterminal; and
- $\mathcal{R}$ maps nonterminals to regular right sides (defined below).

We use A, B, C to range over nonterminals, a, b, c to range over terminals, and ϕ to range over blackbox parsers. The empty sequence is written ϵ. We use r to range over the *(regular) right sides*, defined as follows:

r	$::=$	ϵ	the empty string
	$\mid$	$empty$	the empty language
	$\mid$	c	terminal
	$\mid$	$x = A(e)$	nonterminal
	$\mid$	$x = e$	binding
	$\mid$	$(r\ r)$	concatenation
	$\mid$	$(r \mid r)$	alternation
	$\mid$	(r^*)	Kleene-closure
	$\mid$	$[e]$	constraint
	$\mid$	$\phi(e)$	blackbox

We reserve a distinguished variable, y_0, for use as the formal parameter of nonterminals, and require that y_0 not appear as the x in $x = A(e)$ or in $x = e$ in a regular right side. If $\mathcal{R}(A)$ contains y_0 we say that A is *parameterized*, otherwise, A is *unparameterized*. We require the start nonterminal A_0 to be unparameterized.

For brevity, we may write $A(e)$ in place of $x = A(e)$ in a right side when this is the only occurrence of x in the right side. Similarly, we write A in place of $A(e)$ in right sides when A is unparameterized.

Figure 1 defines the judgment $E \vdash w \in r \Rightarrow E'$, which says that string w belongs to the language of a right side r in environment E, and produces new bindings E'. One of the simpler rules is GL-TERM, which states that the string c is in the language of the right side c in any environment E and produces no new bindings. A somewhat more interesting rule is GL-A, which states that w is in the language of $x = A(e)$ provided that e evaluates to v and w is in the right side of A evaluated with an environment mapping y_0 to v, and no other bindings. This rule also produces the binding $[x{=}w]$. Rule GL-SEQ shows how to process a concatenation expression—notice the way it threads environments through the rule from one part to the next.

The language of a nonterminal A is then defined as follows.

$$L_A = \lambda y.\{w \mid [y_0 = y] \vdash w \in r \Rightarrow E, \mathcal{R}(A) = r\}$$

The language of a grammar G is just the language of its start nonterminal: $L_G = L_{A_0}()$. Then our grammars have the desired compositionality properties:

[2] Ford [11] proves PEGS, which are related to PADS, are closed under union, but only under a non-standard definition of what it means for a string to be in a language (the PEG need only succeed on a *prefix* of the string).

$\boxed{E \vdash w \in r \Rightarrow E'}$

$$\textsc{GL-Eps}\ \frac{}{E \vdash \epsilon \in \epsilon \Rightarrow \cdot}$$

$$\textsc{GL-Term}\ \frac{}{E \vdash c \in c \Rightarrow \cdot}$$

$$\textsc{GL-Pred}\ \frac{\llbracket e \rrbracket E = \mathbf{true}}{E \vdash \epsilon \in [e] \Rightarrow \cdot}$$

$$\textsc{GL-Bind}\ \frac{\llbracket e \rrbracket E = v}{E \vdash \epsilon \in x{=}e \Rightarrow [x{=}v]}$$

$$\textsc{GL-}\phi\ \frac{\llbracket e \rrbracket E = v \quad w \in \Phi(\phi)(v)}{E \vdash w \in \phi(e) \Rightarrow \cdot}$$

$$\textsc{GL-A}\ \frac{\llbracket e \rrbracket E = v \quad \mathcal{R}(A) = r \quad [y_0{=}v] \vdash w \in r \Rightarrow E'}{E \vdash w \in x{=}A(e) \Rightarrow [x{=}w]}$$

$$\textsc{GL-Seq}\ \frac{E \vdash w_1 \in r_1 \Rightarrow E_1 \quad E, E_1 \vdash w_2 \in r_2 \Rightarrow E_2}{E \vdash w_1 w_2 \in (r_1\ r_2) \Rightarrow E_1, E_2}$$

$$\textsc{GL-AltL}\ \frac{E \vdash w_1 \in r_1 \Rightarrow E_1}{E \vdash w_1 \in (r_1 \mid r_2) \Rightarrow E_1}$$

$$\textsc{GL-AltR}\ \frac{E \vdash w_2 \in r_2 \Rightarrow E_2}{E \vdash w_2 \in (r_1 \mid r_2) \Rightarrow E_2}$$

$$\textsc{GL-}*\ \frac{E, E_1, \ldots, E_{i-1} \vdash w_i \in r \Rightarrow E_i, \text{ for } i = 1 \text{ to } k}{E \vdash w_1 \ldots w_k \in (r^*) \Rightarrow E_1, \ldots, E_k}$$

Figure 1. The string-inclusion judgment for rules.

Theorem 1 (Compositionality Properties)
Let G_1 and G_2 be grammars with disjoint variables and nonterminals and let $A_{0,1}$ and $A_{0,2}$ be the start nonterminals of G_1 and G_2. It is possible to construct three new grammars with the following properties.

i. If G' is the union of G_1 and G_2 with a new start nonterminal defined as $(A_{0,1} \mid A_{0,2})$ then $L(G') = L(G_1) \cup L(G_2)$.

ii. If G' is the union of G_1 and G_2 with a new start nonterminal defined as $(A_{0,1}\ A_{0,2})$ then $L(G') = (L(G_1)\ L(G_2))$.

iii. If G' is G_1 with a new start nonterminal defined as $A_{0,1}^*$ then $L(G') = L(G_1)^*$.

The compositionality properties follow immediately from the rules of Figure 1. The following corollary is a direct consequence.

Corollary 2 (Closure Properties)
If L_1 and L_2 are languages of grammars, then $(L_1 \cup L_2)$, $(L_1\ L_2)$, and L_1^* are languages of grammars.

4. Data-dependent Automata

One of the beauties of parsing theory is that high-level grammatical concepts can often be implemented in terms of lower-level automata, which are themselves useful abstractions for programmers and compiler implementers. In this section, we present a new kind of automaton that can implement our data-dependent grammars.

Our new automata are technically transducers—that is, they do not only recognize an input, but also produce outputs, although only from final states. They extend the transducers of Jim and Mandelbaum [15], which were created to parse context-free grammars with regular right-hand sides. Jim and Mandelbaum's idea was to encode regular right sides as subautomata within the transducer, to explicitly label calls from one nonterminal's automaton to another's, and to explicitly label final states with the name of the nonterminal being completed. Constructing transducers in this way gives them three essential characteristics. First, they allow for left-factoring of grammar alternatives, which makes it possible to reduce the nondeterminism that will arise during parsing. Second, they allow for direct implementation of regular expressions as traditional automata, rather than through desugaring into a more restricted form of context-free grammar. Such a direct implementation has a significant impact on parsing efficiency by reducing stack activity. Third, the transducers support left-recursion—programmers are not forced to write their grammars with repetition operators, as is the case for PEGs [11] and many parser combinators [14].

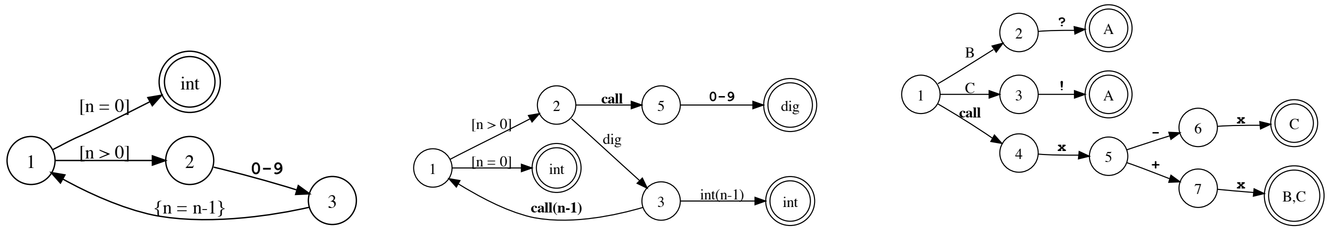
In this section, we extend the core ideas found in Jim and Mandelbaum's work by showing how to add support for binding, dependency, constraints, parameters and black boxes. We give a nondeterministic operational semantics for transducers containing this rich set of features. While this semantics could be used to implement the transducer directly with a backtracking, depth-first parser, we do not do so—Section 6 presents a breadth-first algorithm based on ideas drawn from Earley's parsers for context-free grammars [5]. However, before presenting any of these technical details, we explain the high-level ideas through the use of several simple examples.

4.1 Transducers by example

Figure 4.1 shows three example transducers along with their source grammars. Figure 4.1 (a) defines a fixed-width integer, with the width specified as a parameter to the nonterminal named "int." In this picture, some edges (such as the edge between states 2 and 3) are labeled with terminal symbols. These edges can be interpreted in the ordinary way: a transition is enabled when the current input symbol matches the label of the edge. There are also predicate-labeled edges (with square brackets enclosing the predicates)—the edge between states 1 and 2 is a predicate edge. A transition is enabled along a predicate edge when the predicate evaluates to $true$ in the current environment. The edges with labels enclosed in curly braces are assignment edges—traversing the edge from 3 to 1 assigns $n - 1$ to n. Final states are marked by double-circles and are labeled with a set of nonterminals; in this case, the single nonterminal "int." When evaluation reaches a final state, it indicates completion of all the nonterminals labeling the state.

Figure 4.1 (b) contains a grammar and transducer for the same language as Figure 4.1 (a), expressed in a functional style. Whereas the previous example made no use of the stack, this example will build a stack of depth equal to the value of the argument of "int." Stack frames are pushed (saving the current environment and calling state) at each **call** transition. Stack frames are popped upon arrival at final states. For instance, imagine the transducer takes the call transition between states 2 and 5 and then reads a digit to arrive at the state labeled "dig." At this point, control will *return* to state 2, pop the stack (reinstalling the saved environment) and take the transition between states 2 and 3. The 2–3 transition is taken because it is labeled "dig"—the same nonterminal found in the final state the transducer is returning from. A slightly more sophisticated call pattern occurs between states 3 and 1. In this case a parameter is passed to the callee. Notice that the 3–1 edge is matched by another edge leading from 3 labeled "int(n-1)." The "int" part of the label indicates this edge supports returns from states labeled with nonterminal "int." The "n-1" part of the label indicates it also requires that the parameter passed to the call from which a return is made is equal to n-1.

Finally, Figure 4.1 (c) demonstrates the ability to represent left factoring efficiently and the utility of final states labeled with mul-



(a) int(n) = ([n $>$ 0] (**0** | . . . | **9**) {n = n − 1})* [n = 0]

(b) dig = **0** | **1** | . . . | **9**
int(n) = [n = 0] | [n $>$ 0] dig int(n − 1)

(c) A = (B **?**) | (C **!**) B = **x + x**
C = (**x + x**) | (**x − x**)

Figure 2. Transducers for (a) an imperative specification of a fixed-width integer, (b) a functional specification of a fixed-width integer, and (c) a (very) simple expression language demonstrating left-factoring. Final states are labeled with nonterminals, rather than state numbers.

tiple nonterminals. In this transducer, the call transition from states 1 to 4 is coupled with two return transitions from 1. One return transition is labeled with nonterminal B and one with nonterminal C. Intuitively, the single call along the 1–4 edge implements a parser for both B and C. Now, execution of the transducer can, if the input matches **x−x**, proceed from state 4 to states 5 then 6 and finally to the state labeled C. In this case, we have found a parse for C only and upon return can transition only from state 1 to state 3. Alternatively, if the input matches **x+x**, execution will proceed from 4 to 5 to 7 to the B,C state. In this case both B and C nonterminals have been parsed simultaneously and transitions to either 2 or 3 may be taken from state 1 upon return. The ability to optimize automata by merging states and to parse multiple nonterminals simultaneously results in substantial practical performance gains [15]. We retain this important feature despite the extensions required by data-dependent grammars.

4.2 Trees

Our transducer semantics builds *parse trees*.

Definition 2 (Parse Trees)
A tree T is a sequence of

- terminals c,
- bindings $\{x = v\}$,
- blackbox strings $\langle w \rangle$, or
- four-tuples $x{:}A(v)\langle T' \rangle$ representing a parse of a nonterminal A applied to value v and with leaves bound to x.

Definition 3 (Subtrees)
T_1 is a subtree of T at depth n iff

1. $T = T_0\ T_1\ T_2$ and $n = 1$, or
2. $T = T_0\ x{:}A(v)\langle T' \rangle\ T_2$, and T_1 is a subtree of T' at depth $n - 1$.

We use W to range over *abstract strings*: elements of the set

$$(c \mid \{x{=}v\} \mid \langle w \rangle \mid x{:}A(v){=}w)^*.$$

We will discuss abstract strings further in Section 5. We use m range over abstract strings that are nonterminal-free:

$$(c \mid \{x = v\} \mid \langle w \rangle)^*.$$

We define an *erasure* function, $|\cdot|$, from abstract strings W to strings w as follows:

$$\begin{array}{ll} |\epsilon| = \epsilon & |cW| = c|W| \\ |\{x = v\}W| = |W| & |\langle w \rangle W| = w|W| \\ |x{:}A(v){=}w\,W| = w|W| & \end{array}$$

The following definition provides a way to refer to the concrete strings embedded in abstract strings.

Definition 4 (String members of an abstract string)
$\mathsf{Strings}(W, A, v) = \{w \mid x{:}A(v){=}w \text{ is a substring of } W\}$.

Two additional concepts, the *leaves* of a tree and the *roots* of a tree are central in the following technical development. Intuitively, the *leaves* of a tree is the string of terminals that were parsed. The *roots* of a tree is the top-level, abstract string of symbols in the tree. A more precise definition follows.

Definition 5 (Leaves and Roots)
The leaves and roots of a tree that consists of a single non-terminal-free abstract string are:

- $leaves(m) = |m|$
- $roots(m) = m$

The leaves and roots of a tree that consists of a single non-terminal element are:

- $leaves(x{:}A(v)\langle T \rangle) = leaves(T)$.
- $roots(x{:}A(v)\langle T \rangle) = x{:}A(v){=}leaves(T)$.

The leaves and roots of an arbitrary tree are computed by concatenating the leaves and roots of its sequence of elements.

We now verify some simple properties of roots and leaves that follow from these definitions.

Lemma 1 (The Leaf Lemma)
If $T = m_0\, x_1{:}A_1(v_1)\langle T_1 \rangle\, m_1 \ldots x_n{:}A_n(v_n)\langle T_n \rangle\, m_n$ *then* $leaves(T) = w_0\, w'_1\, w_1 \ldots w'_n w_n$*, where* $w_i = |m_i|$ *and* $w'_i = leaves(T_i)$*, for* $0 \leq i \leq n$.

Proof: from the definition of the $leaves$ function.

Lemma 2 (The Root-Leaf Lemma)
$|roots(T)| = leaves(T)$.

Proof: from the definition of $roots(T)$, T must have the form $m_0\, x_1{:}A_1(v_1)\langle T_1 \rangle\, m_1 \ldots x_n{:}A_n(v_n)\langle T_n \rangle\, m_n$, with $leaves(T_i) = w'_i$. The result then follows from the definition of the erasure function and the Leaf Lemma.

4.3 Semantics

A *parsing transducer* T is a tuple $(\Sigma, \Delta, Q, \Phi, A_0, q_0, \xrightarrow{g}, \xrightarrow{\mathbf{call}(e)}, \mapsto)$ where

- Σ is a finite set of terminals;
- Δ is a finite set of nonterminals;

$$\boxed{(q,E,T,r){::}tl \Rightarrow (q',E',T',r'){::}tl'}$$

$$\textsc{S-Term}\ \frac{r \xrightarrow{c} s}{(q,E,T,r){::}tl \Rightarrow (q,E,Tc,s){::}tl}$$

$$\textsc{S-Pred}\ \frac{r \xrightarrow{\ e\ } s \quad \llbracket e \rrbracket E = \mathbf{true}}{(q,E,T,r){::}tl \Rightarrow (q,E,T,s){::}tl}$$

$$\textsc{S-Bind}\ \frac{r \xrightarrow{x=e} s \quad \llbracket e \rrbracket E = v \quad (x \neq y_0)}{(q,E,T,r){::}tl \Rightarrow (q,E[x{=}v],T\{x{=}v\},s){::}tl}$$

$$\textsc{S-}\phi\ \frac{r \xrightarrow{\phi(e)} s \quad \llbracket e \rrbracket E = v \quad w \in \Phi(\phi)(v)}{(q,E,T,r){::}tl \Rightarrow (q,E,T\langle w\rangle,s){::}tl}$$

$$\textsc{S-Call}\ \frac{r \xrightarrow{\mathbf{call}(e)} s \quad \llbracket e \rrbracket E = v}{(q,E,T,r){::}tl \Rightarrow (s,[y_0{=}v],\epsilon,s){::}(q,E,T,r){::}tl}$$

$$\textsc{S-Return}\ \frac{\begin{array}{c} r \mapsto A \quad r' \xrightarrow{x=A(e)} s \\ \llbracket e \rrbracket E' = v = E(y_0) \quad (x \neq y_0) \end{array}}{\begin{array}{c}(q,E,T,r){::}(q',E',T',r'){::}tl \Rightarrow \\ (q',E'[x{=}leaves(T)],T'\,x{:}A(v)\langle T\rangle,s){::}tl\end{array}}$$

Figure 3. The stack evaluation relation.

- Q is a finite set of states;
- Φ is a a finite set of blackboxes;
- $A_0 \in \Delta$ is the start nonterminal;
- $q_0 \in Q$ is the initial state;
- $\xrightarrow{g}$ is the transition relation, where g has one of the following forms: c (terminal), e (constraint), $x{=}e$ (binding), $\phi(e)$ (blackbox), $x{=}A(e)$ (nonterminal);
- $\xrightarrow{\mathbf{call}(e)} \subseteq Q \times \Omega \times Q$ is the call relation; and
- $\mapsto\ \subseteq Q \times \Delta$ is the output relation from final states to nonterminals.

We write the reflexive and transitve closure of the transition relation as $r {\rightarrow}^* s$. When we wish to refer to the labels of the constituent transitions, we write $\xrightarrow{P}{}^*$, where $P \in g^*$. We use q, r, s, t, u to range over states. A *configuration* is a 5-tuple $(q,E,T,r) :: tl$ where tl acts as a stack that grows to the left. The first element of the tuple (q) is the *callee* at which the parse of the current nonterminal(s) began. While the callee does not influence the parsing process, its inclusion simplifies the task of proving that the Earley-style parsing algorithm in Section 6 preserves the semantics of transducers. The next element of the tuple (E) is the current environment. It is followed by the parse tree (T) under construction. The last element of the tuple (r) is the current state.

Figure 3 defines $\Rightarrow$, a single-step evaluation relation for configurations. One of the simpler rules is S-INPUT, which extends the current tree with a terminal based on a transition on that terminal appearing in the transducer. Of greater complexity are rules S-CALL and S-RETURN, which manage the stack. Rule S-CALL transitions to another state much like a function call. A new stack tuple is pushed with the callee as the current state, an empty tree, and an environment that contains only the call argument bound to y_0. To guarantee access to the call argument at any point during evaluation, we ensure that y_0 is not updated with the side condition $x \neq y_0$ on rules S-BINDING and S-RETURN.

Rule S-RETURN is invoked whenever a final state is reached. The stack is popped and any transition $x = A(e)$ from the previous current-state r' can be followed, as long as e evaluates to the call argument of the current tuple. Also, the string parsed by $A(e)$, $leaves(T)$, is bound to x in the environment.

The multi-step evaluation relations, $\Rightarrow^*$ and $\Rightarrow^+$, are defined as usual. Notice that these relations define a nondeterministic algorithm for parsing with the transducer.

We complete this section by noting that just as we can talk about nonterminal languages in the grammar, we can describe nonterminal languages in the transducer. However, given that the transducer might have multiple callee states for any given nonterminal, we describe the nonterminal languages with respect to a particular callee.

Definition 6
We characterize $L_A(q)$, the language of A at q, as

$$\begin{aligned} L_A(q) = &\lambda\, y.\{w \mid (q,[y_0{=}y],\epsilon,q) \Rightarrow^* (q,E,T,r), \\ &r \mapsto A,\ w = leaves(T)\} \end{aligned}$$

and the language of the transducer $L_{\mathcal{T}} = L_{A_0}(q_0)()$.

5. Grammars and Transducers

Before we present our Earley-style parsing algorithm, we will prove that data-dependent automata are powerful enough to parse the languages of data-dependent grammars. We do so in two steps. First, we identify sufficient conditions for a given transducer to parse the language of a given grammar. Then, we present a translation from grammars to transducers that satisfies the conditions.

5.1 Abstract Languages

Our strategy for relating grammars to transducers is to reduce the problem to one of independently comparing subautomata within the transducer with right-hand sides in the grammar, without reference to the remainder of the grammar and transducer. In Section 4.2, we defined abstract strings W, which contain symbolic elements, like nonterminals, in addition to terminals. Here, we define *abstract languages*—sets of abstract strings—for grammar nonterminals and transducer callees. We then compare subautomata with right-hand sides via their respective abstract languages and show how equivalence of these abstract languages implies equivalence between a grammar and transducer.

In essence, abstract languages are defined as the abstract strings that can be read directly from right-hand sides and transducer subautomata, without recurring on nonterminals. For example, in the following grammar,

$$A = B \mid C \qquad B = b \qquad C = c$$

the (concrete) language of A is the set $\{b, c\}$ whereas the abstract language is the set $\{B, C\}$. The difference is that the latter makes no mention of the languages of B and C. For grammars with bound nonterminals, the abstract language is a little more involved. Specifically, each bound nonterminal in an abstract string W is represented as $x{:}A(v){=}w$—that is, it carries an associated concrete string along with it. The strings w must be included because evaluating W's membership requires a value to be bound to x for use in evaluating the remainder of W. However, the essential abstraction is that we do not force that value to be a valid member of the language of $A(v)$. Although constraints in the language can constrain the range of possible associated strings, they do not necessarily constrain them to be drawn strictly from the language of the respective associated nonterminal.

In Figure 4, we define the abstract-string inclusion judgment for grammars, which we can then use to define the *abstract language* of a nonterminal.

Definition 7 (Grammar Abstract Language)
The abstract language of nonterminal A is

$$G_A = \lambda\, y.\{W \mid [y_0{=}y] \vdash W \in r \Rightarrow E, \mathcal{R}(A) = r\}$$

$\boxed{E \vdash W \in r \Rightarrow E'}$

$$\textrm{AGL-E}{\small\textrm{PS}}\ \frac{}{E \vdash \epsilon \in \epsilon \Rightarrow \cdot}$$

$$\textrm{AGL-T}{\small\textrm{ERM}}\ \frac{}{E \vdash c \in c \Rightarrow \cdot}$$

$$\textrm{AGL-P}{\small\textrm{RED}}\ \frac{\llbracket e \rrbracket E = \mathbf{true}}{E \vdash \epsilon \in [e] \Rightarrow \cdot}$$

$$\textrm{AGL-B}{\small\textrm{IND}}\ \frac{\llbracket e \rrbracket E = v}{E \vdash \{x{=}v\} \in x{=}e \Rightarrow [x{=}v]}$$

$$\textrm{AGL-}\phi\ \frac{\llbracket e \rrbracket E = v \quad w \in \Phi(\phi)(v)}{E \vdash \langle w \rangle \in \phi(e) \Rightarrow \cdot}$$

$$\textrm{AGL-A}\ \frac{\llbracket e \rrbracket E = v}{E \vdash x{:}A(v) = w \in x{=}A(e) \Rightarrow [x{=}w]}$$

$$\textrm{AGL-S}{\small\textrm{EQ}}\ \frac{E \vdash W_1 \in r_1 \Rightarrow E_1 \quad E, E_1 \vdash W_2 \in r_2 \Rightarrow E_2}{E \vdash W_1 W_2 \in (r_1\ r_2) \Rightarrow E_1, E_2}$$

$$\textrm{AGL-A}{\small\textrm{LT}}\textrm{L}\ \frac{E \vdash W \in r_1 \Rightarrow E_1}{E \vdash W \in (r_1 \mid r_2) \Rightarrow E_1}$$

$$\textrm{AGL-A}{\small\textrm{LT}}\textrm{R}\ \frac{E \vdash W \in r_2 \Rightarrow E_2}{E \vdash W \in (r_1 \mid r_2) \Rightarrow E_2}$$

$$\textrm{AGL-*}\ \frac{E, E_1, \ldots, E_{i-1} \vdash W_i \in r \Rightarrow E_i, \ \text{for } i = 1 \text{ to } k}{E \vdash W_1 \ldots W_k \in (r^*) \Rightarrow E_1, \ldots, E_k}$$

Figure 4. String-inclusion rules for abstract languages of nonterminals.

In Figure 5, we present a judgment relating abstract strings W to paths through the transducer. We also define its reflexive and transitive closure in the expected way. We can now define the *abstract language* of a nonterminal at a callee.

Definition 8 (Transducer Abstract Language)
The abstract language of nonterminal A, at callee q is

$$\mathcal{T}_A(q) = \lambda\, y.\{W \mid [y_0{=}y] \vdash W : q \rightarrow^* s;\ E \text{ and } s \mapsto A\}$$

5.2 Transducer Conditions

Perhaps *the key element* of the proof that transducers implement grammars properly is the definition of the set of conditions **(T0)**, **(T1)** and **(T2)** that relates the language of a grammar G to the language implemented by a transducer $\mathcal{T}$. The intuition for the conditions is as follows.

- **(T0)** states that the language of the transducer starting at q_0 (the start state) and finishing at a final state for A_0 (the start nonterminal) must be the same as the language of A_0 in the grammar. In a nutshell, this condition states that the transducer implements the start nonterminal correctly.
- An $A(e)$*-edge* is any edge with the form $r \xrightarrow{x=A(e)} s$. With this in mind, **(T1)** states that any $A(e)$-edge from a state r must be coupled with a call edge from r with parameter e. Moreover, that call edge must transition to a state q that implements the language of non-terminal A. In a nutshell, this condition states that the transducer contains *some call* that implements each $A(e)$-edge correctly.
- A *callee* is either the start state q_0 or any other state that has a call edge leading to it. Hence, **(T2)** states that for each non-terminal A, the language of A at every callee is a subset of the language of A in the grammar. In a nutshell, the transducer contains *no calls* that put extra garbage into the language of any non-terminal.

$\boxed{E \vdash W : q \rightarrow r;\ E'}$

$$\textrm{ATL-T}{\small\textrm{ERM}}\ \frac{q \xrightarrow{c} r}{E \vdash c : q \rightarrow r;\ \cdot}$$

$$\textrm{ATL-P}{\small\textrm{RED}}\ \frac{q \xrightarrow{e} r \quad \llbracket e \rrbracket E = \mathbf{true}}{E \vdash \epsilon : q \rightarrow r;\ \cdot}$$

$$\textrm{ATL-B}{\small\textrm{IND}}\ \frac{q \xrightarrow{x=e} r \quad \llbracket e \rrbracket E = v \quad (x \neq y_0)}{E \vdash \{x{=}v\} : q \rightarrow r;\ [x{=}v]}$$

$$\textrm{ATL-}\phi\ \frac{q \xrightarrow{\phi(e)} r \quad \llbracket e \rrbracket E = v \quad w \in \Phi(\phi)(v)}{E \vdash \langle w \rangle : q \rightarrow r;\ \cdot}$$

$$\textrm{ATL-A}\ \frac{q \xrightarrow{x=A(e)} r \quad \llbracket e \rrbracket E = v \quad (x \neq y_0)}{E \vdash x{:}A(v) = w : q \rightarrow r;\ [x{=}w]}$$

$\boxed{E \vdash W : q \rightarrow^* r;\ E'}$

$$\frac{}{E \vdash \epsilon : q \rightarrow^* q;\ \cdot}$$

$$\frac{E \vdash W_1 : q \rightarrow^* r';\ E_1 \quad E, E_1 \vdash W_2 : r' \rightarrow r;\ E_2}{E \vdash W_1 W_2 : q \rightarrow^* r;\ E_1, E_2}$$

Figure 5. Abstract string inclusion for transducers and its reflexive and transitive closure.

Formally, these conditions are specified as follows.

Definition 9
$\mathcal{T}$ is a transducer for G iff

(T0) $\mathcal{T}_{A_0}(q_0)() = G_{A_0}()$.

(T1) If $r \xrightarrow{x=A(e)} s$ then $r \xrightarrow{\mathbf{call}(e)} q$, for some q with $\mathcal{T}_A(q)(v) = G_A(v)$, for all v.

(T2) If q is a callee, then $\mathcal{T}_A(q)(v) \subseteq G_A(v)$, for all A, v.

5.3 Grammar-Transducer Correspondence

In our first few lemmas, we investigate some properties of stack executions. Our first result, the Backup Lemma, shows that if there is an execution path resulting in a particular stack, then there is a (shorter) execution path to any of the substacks within it. It is used frequently in the S-RETURN case of inductive proofs before application of the induction hypothesis. We use the notation $\Rightarrow^+$ to indicate that at least one step was taken.

Lemma 3 (The Backup Lemma)
If $tl_1 \Rightarrow^+ (r_1, E_1, T_1, s_1){::}(r_2, E_2, T_2, s_2){::}tl$ *then*

a. $tl_1 \Rightarrow^* (r_2, E_2, T_2, s_2){::}tl$ *in fewer steps, and*

b. $s_2 \xrightarrow{\mathbf{call}(e)} r_1$ *and* $\llbracket e \rrbracket E_2 = E_1(y_0)$*, and*

c. $(r_1, [y_0{=}E_1(y_0)], \epsilon, r_1){::}(r_2, E_2, T_2, s_2){::}tl \Rightarrow^*$ $(r_1, E_1, T_1, s_1){::}(r_2, E_2, T_2, s_2){::}tl$*, in equal or fewer steps.*

Proof: by induction on the number steps taken. Rule S-RETURN is the interesting case, where we apply the induction hypothesis twice, each time removing the top stack element.

Corollary 4
If $tl_1 \Rightarrow^{+} (r_1, E_1, T_1, s_1)::(r_2, E_2, T_2, s_2)::tl$ *then* $(r_2, E_2, T_2, s_2)::tl \Rightarrow^{*} (r_1, E_1, T_1, s_1)::(r_2, E_2, T_2, s_2)::tl$*, in equal or fewer steps.*

Proof: by parts b. and c. of Backup Lemma and rule S-CALL.

Our next lemma generalizes the third part of the Backup Lemma to show that for any execution there is a subexecution starting at the callee field of the top stack element, with an environment containing only y_0 and an empty tree.

Corollary 5 (Start-at-callee)
If $(q, E_0, \epsilon, q) \Rightarrow^{*} (q_1, E_1, T_1, s_1)::tl$ *and* $y_0 \in \mathsf{dom}(E_0)$ *then* $(q_1, [y_0{=}E_1(y_0)], \epsilon, q_1)::tl \Rightarrow^{*} (q_1, E_1, T_1, s_1)::tl.$

Proof: By induction on the number of steps taken, using the Backup Lemma for the case of rule S-RETURN.

Next, show that the callee field is true to its name—that is, it is always populated by a transducer callee.

Lemma 6
If q *is a callee and* $(q, E_0, \epsilon, q) \Rightarrow^{*} (q_1, E_1, T_1, s_1)::tl$ *then* q_1 *is a callee.*

Proof: by induction on number of steps taken.

We follow with simple result showing that y_0 is immutable, as desired:

Lemma 7 (y_0 Immutability)
If $(q, E, T, r)::tl \Rightarrow^{*} (q, E', T', r')::tl$ *then* $E(y_0) = E'(y_0)$.

Proof: by induction on the number of steps taken.

The Prefix Lemma, below, is helpful for reasoning about a subcomponent of a tree.

Lemma 8 (The Prefix Lemma)
If $(q, E_0, \epsilon, q) \Rightarrow^{*} (q, E, T_1 T_2, r)$*, then*
$\exists\, s, E'.\, (q, E_0, \epsilon, q) \Rightarrow^{*} (q, E', T_1, s)$ *and* $\mathsf{dom}(E') \subseteq \mathsf{dom}(E)$.

Proof: by induction on height $\Rightarrow^{*}$ derivation.

We will now present a theorem showing that results about stack evaluations in one context to be applied in other contexts. We single it out as a theorem because it also provides intuition as to the extent to which parsing is context sensitive. Specifically, it shows that the context-sensitivity of a parse is strictly limited to the environment in the top stack tuple. Anything outside of that environment, located in the stack below, cannot affect parsing at higher stack levels. This theorem plays an important role in the proofs of nearly all of the lemmas and theorems that follow.

However, its correctness is far from obvious and relies on the fact that if one were to attempt to pop a series of stack frames and then push them back on, one would not arrive back in exactly the same state, because the embedded parse trees would grow. We prove this property first, before stating the theorem. Note that the tricky aspect of this lemma was finding an appropriate statement of the desired property. We use the notation @ to represent the append function on stacks.

Lemma 9 (No Stack Repeat)
If $tl_1 @ (q_1, E_1, T_1, r_1)::tl \Rightarrow^{*} (q_2, E_2, T_2, r_2)::tl$*, where* tl_1 *is not the empty stack, then* $length(T_2) > length(T_1)$.

Proof: by induction on height $\Rightarrow^{*}$ derivation. Notice that the case of rule S-CALL is impossible, because application of the induction hypothesis leads to a contradiction. For rule S-RETURN, it either follows immediately from the premises, or from Backup Lemma and the induction hypothesis.

Theorem 10 (Context Independence)
If $(q, E, T, r)::tl \Rightarrow^{*} (q, E_1, T_1, r_1)::tl$ *then*
$\forall\, tl'.\, (q, E, T, r)::tl' \Rightarrow^{*} (q, E_1, T_1, r_1)::tl'$

Proof: by induction on $\Rightarrow^{*}$ derivation. We use Lemma 9 to show that the case of rule S-CALL is inapplicable.

The following lemmas complete the statement of correspondence between trees and stack evaluations. In essence, they state that every subtree within the tree produced by an evaluation has some corresponding sub-evaluation.

Lemma 11 (The Direct-Subtree Lemma)
If $(q, E_0, \epsilon, q) \Rightarrow^{*} (q, E, T_0\, x{:}A(v)\langle T\rangle, r)$ *and* $y_0 \in \mathsf{dom}(E_0)$ *then there exist* $e_1, e_2, E_1, r', s, t, u$ *such that*

- $s \xrightarrow{\mathbf{call}(e_1)} t$, $s \xrightarrow{x=A(e_2)} r'$, $u \mapsto A$*, and*
- $[\![e_1]\!]E_1 = [\![e_2]\!]E_1 = v$*, and*
- $(q, E_0, \epsilon, q) \Rightarrow^{*} (q, E_1, T_0, s)$*, and*
- $(q, [y_0{=}v], \epsilon, t)::(q, E_1, T_0, s) \Rightarrow^{*} (t, E_2, T, u)::(q, E_1, T_0, s)$*, and*
- $r' \rightarrow^{*} r.$

Proof: by induction on $\Rightarrow^{*}$ relation, using Backup Lemma and Lemma 5 for case of rule S-RETURN. Only other applicable case is rule S-PRED, which follows directly from the induction hypothesis.

Lemma 12 (The Subtree Lemma)
If q *is a callee,* $y_0 \in \mathsf{dom}(E_0)$*,* $(q, E_0, \epsilon, q) \Rightarrow^{*} (q, E_1, T, r)$*, and* $x{:}A(v_1)\langle T_1\rangle$ *is a subtree of* T *at depth* n*, then there exists a stack* tl*, callee* s *and state* $t \mapsto A$ *such that* $(s, [y_0{=}v_1], \epsilon, s)::tl \Rightarrow^{*} (s, E_2, T_1, t)::tl$.

Proof: by induction on the depth of T_1. For a tree of depth 1, where $T = T_0\, x{:}A(v)\langle T_1\rangle\, T_2$ we use the Prefix Lemma to "remove" T_2 and then apply the Direct-Subtree Lemma. The inductive case follows a similar approach. T_1 must be a subtree of some direct-subtree T'. We use the Prefix and Direct-Subtree Lemmas on T' to satisfy the premises of the induction hypothesis. The only twist is that we use the Context-Independence Lemma to "remove" the stack before applying the induction hypothesis.

Next, we seek to establish the relationships between our concrete and abstract languages in a series of lemmas. These lemmas are essential to our end goal because we are seeking to show that correspondence between a grammar and transducer at the abstract level is enough to ensure correspondence at the concrete level. To do so, we need to understand the relationship between concrete and abstract languages for grammars and transducers, respectively.

First, we show that the presence of a concrete string in the concrete language of a nonterminal implies the existence of some corresponding abstract string in the abstract language of that same nonterminal. Then, we'll show that the opposite holds, under the important constraint that every concrete string within the abstract string actually belongs to language of the nonterminal with which it is associated.

Lemma 13 (Grammar Concrete-Abstract Correspondence)
If $E \vdash w \in r \Rightarrow E'$ *then there exists* W *such that* $w = |W|$, $E \vdash W \in r \Rightarrow E'$*, and, for all* $w_1 \in \mathsf{Strings}(W, A, v)$, $[y_0{=}v] \vdash w_1 \in r_1 \Rightarrow E_1$ *with a lower derivation height (where* $\mathcal{R}(A) = r_1$*).*

Proof: By induction on premise derivation.

Lemma 14 (Grammar Abstract-Concrete Correspondence)
If $E \vdash W \in r \Rightarrow E'$ and $\mathsf{Strings}(W, A, v) \subseteq L_A(v)$ then $E \vdash |W| \in r \Rightarrow E'$.

Proof: By induction on height of first derivation. The case of rule AGL-A uses the second premise.

Lemma 15 (Transducer Concrete-Abstract Correspondence)
If $(q, E_0, \epsilon, q) \Rightarrow^* (q, E, T, r)$ then $E = E_0, E_1$ and $E_0 \vdash roots(T) : q \rightarrow^* r;\, E_1$.

Proof: by induction on the $\Rightarrow^*$ relation, using the Backup Lemma (Lemma 3) in the S-RETURN case.

Below, we define a predicate on transducers and abstract strings which corresponds to the constraint $\mathsf{Strings}(W, A, v) \subseteq L_A(v)$ from Lemma 14. This definition is used in the statement of Lemma 17.

Definition 10
$\mathsf{Ok}(\mathcal{T}, W)$ iff $r \xrightarrow{x=A(e)} s$ implies that there exists a callee t such that $r \xrightarrow{\mathbf{call}(e)} s$ and $\mathsf{Strings}(W, A, v) \subseteq L_A(t)(v)$.

Lemma 16 (Ok W-Subcomponents)
If $\mathsf{Ok}(\mathcal{T}, W_1 W_2)$ then $\mathsf{Ok}(\mathcal{T}, W_1)$ and $\mathsf{Ok}(\mathcal{T}, W_2)$.

Proof: By inspection of structure of W and definition of Ok.

Lemma 17 (Transducer Abstract-Concrete Correspondence)
If q is a state, $E \vdash W : s \rightarrow^* r;\, E'$, and $\mathsf{Ok}(\mathcal{T}, W)$, then exists T' such that $W = roots(T')$, $(q, E, T, s) \Rightarrow^* (q, [E, E'], T\,T', r)$.

Proof: By induction on the height of the derivation of the third premise. For inductive case, use Lemma 16 to help satisfy the induction hypothesis. Then, proceed by case analysis of ATL rules. The case of rule ATL-A uses the fourth premise, rule S-CALL, Lemma 10, and, finally, rule S-RETURN. For the latter, we use Lemma 7 to help satisfy the third premise.

Now, we may begin the task of relating the languages of grammars and transducers.

Corollary 18 (The Roots Lemma)
If q is a callee, $(q, [y_0{=}v], \epsilon, q) \Rightarrow^* (q, E, T, r)$ and $r \mapsto A$, then $roots(T) \in \mathcal{T}_A(q)(v)$.

Proof: This Corollary follows directly from Lemma 15 and the definition of $\mathcal{T}_A(q)(v)$.

With this result, we can now relate trees constructed in a stack evaluation directly to a grammar.

Lemma 19 (The Roots-Grammar Lemma)
If $\mathcal{T}$ is a transducer for G, q is a callee, $(q, [y_0{=}v], \epsilon, q) \Rightarrow^* (q, E, T, r)$, and $r \mapsto A$, then $roots(T)$ in $G_A(v)$.

Proof: The proof of the Roots-Grammar Lemma follows directly from Lemma 18 together with condition **(T2)** that is required for $\mathcal{T}$ to be a transducer for G.

At this point, we are ready to prove our final result, namely, that the language of a transducer will match the language of a grammar, assuming our conditions are met. We prove language equality in two steps, by proving mutual inclusion of the two languages. The Leaves Lemma will show that the transducer's language is included in the grammar's, and the Callee Correctness Lemma will show the reverse.

Lemma 20 (The Leaves Lemma)
If $\mathcal{T}$ is a transducer for G, q is a callee, $(q, [y_0{=}v], \epsilon, q) \Rightarrow^* (q, E, T, r)$, and $r \mapsto A$, then $leaves(T) \in L_A(v)$.

Proof: by induction on height of the tree T. For all cases, the essential lemmas are the Root-Grammar lemma, and Lemma 14. Together, they take a string from a transducer language to an abstract transducer language to an abstract grammar language and then to a concrete grammar language. In the inductive case, for every direct subtree of T, we use the Prefix lemma and the Direct-Subtree lemma to satisfy the induction hypothesis for that subtree. Then, Lemma 14 and Lemma 1 give us our result.

Lemma 21 (Callee Correctness)
If $\mathcal{T}$ is a transducer for G, q is a callee, $G_A(v) = \mathcal{T}_A(q)(v)$, $\mathcal{R}(A) = r$, and $[y_0{=}v] \vdash w \in r \Rightarrow E$, then $w \in L_A(q)(v)$.

Proof: by induction on the height of the derivation of the last premise. However, in place of case analysis on the GL rules, we perform cases analysis on the structure of w. Specifically, we designate base cases as those for which no portion of w corresponds to a nonterminal, and those for which some do.

In the proof of this lemma, we take the opposite path as the previous lemma, taking a string from a grammar language to an abstract grammar language to an abstract transducer language and then to a concrete transducer language. For all cases, we start with Lemma 13, use the third premise to cross over to the abstract grammar language, and then finish up with Lemma 17. For the base cases, we satisfy $\mathsf{Ok}(\mathcal{T}, W)$ trivially (the relevant W is just an m), while for the inductive cases we make use of condition **(T1)** and then apply the induction hypothesis. In the inductive case, we tie together our results using Lemma 2.

Theorem 22 (Transducer for Grammar Correctness)
If $\mathcal{T}$ is a transducer for G then $L_{\mathcal{T}} = L_G$.

Proof: with the above lemmas and the definitions of $L_{\mathcal{T}}$ and L_G, we can directly show that $L_{\mathcal{T}} \in L_G$ and $L_G \in L_{\mathcal{T}}$, respectively, with the latter result relying on condition **(T0)**.

5.4 Translation from Grammars to Transducers

The translation from data dependent grammars to transducers, presented in Figure 6, is an extension of the Thompson translation of regular expressions into automata. The first judgment, $S \vdash r \rightsquigarrow (s, F, \mathcal{T})$, states that right-hand side r is translated into transducer graph $\mathcal{T}^3$ with start state s and final state F. S is a finite partial map from nonterminals to the start states for the automata implementing them. The second judgment, $G \rightsquigarrow \mathcal{T}$, uses the first judgment to build transducer graphs for all right-hand sides of nonterminals in grammar G and puts the results together to construct a complete transducer $\mathcal{T}$. These judgments use the following notation.

- $[s \xrightarrow{z} t]$ is a transducer graph with a single arc of sort z from states s to t.
- $[s \longrightarrow t]$ is a transducer graph with a single epsilon transition from states s to t. An epsilon transition is an abbreviation for the predicate transition $s \xrightarrow{\mathbf{true}} t$.
- $\mathcal{T}_1; \mathcal{T}_2$ is the transducer graph built by taking the union of nodes and edges from graphs $\mathcal{T}_1$ and $\mathcal{T}_2$.
- $[s \not\rightarrow t]$ is the transducer graph with disconnected states s and t.

[3] In a slight abuse of notation, we have overloaded meta-variable $\mathcal{T}$ so it represents both transducer graphs (the set of nodes and edges making up the transducer relations) and full transducers (the graphs plus auxiliary information such as the black boxes, terminal alphabet, etc.).

- Given a transducer graph $\mathcal{T}$, $[\Sigma, \Delta, \Phi, A_0, s_{A_0}, \mathcal{T}]$ builds the transducer with graph $\mathcal{T}$ and other components specified by $\Sigma, \Delta, \Phi, A_0$ and s_{A_0}.

We are able to prove the following lemma establishing the correctness of the right-hand side translation.

Lemma 23 (Rule Translation Correctness)

i. *If* $S \vdash r \rightsquigarrow (s, F, \mathcal{T})$ *then the following are true of transducer graph* $\mathcal{T}$*:*

a. There are no final states in $\mathcal{T}$*.*

b. If $r \xrightarrow{x=A(e)} t$ *in* $\mathcal{T}$ *then* $r \xrightarrow{\mathbf{call}(e)} S(A)$ *in* $\mathcal{T}$*.*

c. If $r \xrightarrow{\mathbf{call}(e)} s$ *in* $\mathcal{T}$ *then* $s = S(A)$ *for some* A*.*

ii. *If* $E \vdash W \in r \Rightarrow E'$ *and* $S \vdash r \rightsquigarrow (s, F, \mathcal{T})$ *then* $E \vdash W : s \to^{*} F;\, E'$*.*

iii. *If* $S \vdash r \rightsquigarrow (s, F, \mathcal{T})$ *and* $E \vdash W : s \to^{*} F;\, E'$ *then* $E \vdash W \in r \Rightarrow E'$*.*

Proof: Each part is proven independently, by induction on height of the first derivation. Intuitively, parts (*ia*) and (*ic*) help establish **(T2)** (no extra garbage in the language). Part (*ib*) helps establish **(T1)** (all $A(e)$ edges are implemented). Parts (*ii*) and (*iii*) help establish the language equivalence conditions specified in **(T0)**, **(T1)** and **(T2)**.

The Rule Translation Correctness Lemma, together with the definition of the grammar translation and Definition 9 is sufficient to prove that a transducer produced by the translation implements its grammar correctly:

Theorem 24 (Grammar Translation Correctness)
If $G \rightsquigarrow \mathcal{T}$ *then* $\mathcal{T}$ *is a transducer for* G*.*

6. An Earley-style Parsing Algorithm

The stack evaluation relation, while informative as a semantics of the transducer, does not lend itself to efficient direct implementation. The nondeterministic nature of the relation could result in an exponential time bounds for parsing even relatively simple grammars, and nontermination for grammars with left-recursion. Therefore, in this section, we provide an alternative, Earley-style parsing algorithm that matches the transducer semantics, while (often) improving execution behavior.

6.1 The Algorithm

The traditional Earley algorithm proceeds by computing a set of *Earley items* for each position in the input. These items are computed from left to right: First the Earley set for input position 1 is computed, then the set for input position 2, *etc.*. Each item contains information about what grammar rule is being parsed (and how much of that grammar rule has been parsed) as well as the position in the input where the parse for that rule started. A key aspect of the algorithm is that the Earley sets act like a memoization table—rather than re-parsing portions of the input multiple times like an exponential-time back-tracking algorithm would do, Earley saves work by reusing items from Earley sets. In Earley's case, the algorithm works because context-free grammars are, well, *context-free*. Intuitively, in our case, an extension of the algorithm will work because we include the local context E in our modified Earley sets and, crucially, because, as stated by the Context Independence Lemma (Lemma 10), parsing a particular grammar rule only depends upon that local context, not on the tail of the stack.

With that background in mind, we present our modified Earley algorithm. The Earley sets involved in our algorithm are indexed sets of parse trees (forests). A tree T belongs to the set

$\boxed{S \vdash r \rightsquigarrow (s, F, \mathcal{T})}$

$$\text{T-EPS}\ \frac{s, F \text{ fresh}}{S \vdash \epsilon \rightsquigarrow (s, F, [s \longrightarrow F])}$$

$$\text{T-TERM}\ \frac{s, F \text{ fresh}}{S \vdash c \rightsquigarrow (s, F, [s \xrightarrow{c} F])}$$

$$\text{T-PRED}\ \frac{s, F \text{ fresh}}{S \vdash [e] \rightsquigarrow (s, F, [s \xrightarrow{e} F])}$$

$$\text{T-BINDING}\ \frac{s, F \text{ fresh} \qquad x \neq y_0}{S \vdash x = e \rightsquigarrow (s, F, [s \xrightarrow{x=e} F])}$$

$$\text{T-BLACKBOX}\ \frac{s, F \text{ fresh}}{S \vdash \phi(e) \rightsquigarrow (s, F, [s \xrightarrow{\phi(e)} F])}$$

$$\text{T-A}\ \frac{s, F \text{ fresh} \qquad x \neq y_0}{S \vdash x{=}A(e) \rightsquigarrow (s, F, [s \xrightarrow{x=A(e)} F]; [s \xrightarrow{\mathbf{call}(e)} S(A)])}$$

$$\text{T-SEQ}\ \frac{\begin{array}{c} s, F \text{ fresh} \\ S \vdash r_1 \rightsquigarrow (s_1, F_1, \mathcal{T}_1) \quad S \vdash r_2 \rightsquigarrow (s_2, F_2, \mathcal{T}_2) \\ \mathcal{T}_3 = [s \longrightarrow s_1]; [F_1 \longrightarrow s_2]; [F_2 \longrightarrow F] \end{array}}{S \vdash r_1.r_2 \rightsquigarrow (s, F, \mathcal{T}_1; \mathcal{T}_2; \mathcal{T}_3)}$$

$$\text{T-ALT}\ \frac{\begin{array}{c} s, F \text{ fresh} \\ S \vdash r_1 \rightsquigarrow (s_1, F_1, \mathcal{T}_1) \quad S \vdash r_2 \rightsquigarrow (s_2, F_2, \mathcal{T}_2) \\ \mathcal{T}_3 = [s \longrightarrow s_1]; [s \longrightarrow s_2]; [F_1 \longrightarrow F]; [F_2 \longrightarrow F] \end{array}}{S \vdash r_1 + r_2 \rightsquigarrow (s, F, \mathcal{T}_1; \mathcal{T}_2; \mathcal{T}_3)}$$

$$\text{T-}*\ \frac{\begin{array}{c} s, F \text{ fresh} \qquad S \vdash r_1 \rightsquigarrow (s_1, F_1, \mathcal{T}_1) \\ \mathcal{T}_2 = [s \longrightarrow s_1]; [F_1 \longrightarrow F]; [s \longrightarrow F]; [F_1 \longrightarrow s_1] \end{array}}{S \vdash r_1{*} \rightsquigarrow (s, F, \mathcal{T}_1; \mathcal{T}_2)}$$

$$\text{T-EMPTY}\ \frac{s, F \text{ fresh}}{S \vdash \mathit{empty} \rightsquigarrow (s, F, [s \not\longrightarrow F])}$$

$\boxed{G \rightsquigarrow \mathcal{T}}$

$$\text{T-G}\ \frac{\begin{array}{c} \mathcal{R} = [A_0 = \lambda y_0.r_{A_0}, \ldots, A_k = \lambda y_0.r_{A_k}] \\ s_{A_0}, \ldots, s_{A_k} \text{ fresh} \\ S = [A_0 = s_{A_0}, \ldots, A_k = s_{A_k}] \\ S \vdash r_{A_i} \rightsquigarrow (s_i, F_i, \mathcal{T}_i) \quad (\text{for } i = 0, \ldots, k) \\ \mathcal{T}_{init} = [s_{A_0} \longrightarrow s_0]; \cdots; [s_{A_k} \longrightarrow s_k] \\ \mathcal{T}_{final} = [F_0 \mapsto A_0]; \cdots; [F_k \mapsto A_k] \\ \mathcal{T} = \mathcal{T}_0; \cdots; \mathcal{T}_k; \mathcal{T}_{init}; \mathcal{T}_{final} \end{array}}{(\Sigma, \Delta, \Phi, A_0, \mathcal{R}) \rightsquigarrow [\Sigma, \Delta, \Phi, A_0, s_{A_0}, \mathcal{T}]}$$

Figure 6. Translation from grammars to transducers.

$$\boxed{T \in tree(i,j,q,E,s)}$$

$$\text{ET-INIT}\ \frac{}{\epsilon \in tree(0,0,q_0,[y_0{=}()],q_0)}$$

$$\text{ET-TERM}\ \frac{T \in tree(i,j{-}1,q,E,r) \quad r \xrightarrow{c_j} s}{Tc_j \in tree(i,j,q,E,s)}$$

$$\text{ET-PRED}\ \frac{T \in tree(i,j,q,E,r) \quad r \xrightarrow{e} s \quad [\![e]\!]E = \mathbf{true}}{T \in tree(i,j,q,E,s)}$$

$$\text{ET-BIND}\ \frac{\begin{array}{ll} T \in tree(i,j,q,E,r) & r \xrightarrow{x=e} s \\ [\![e]\!]E = v & (x \neq y_0) \end{array}}{T\{x{=}v\} \in tree(i,j,q,E[x{=}v],s)}$$

$$\text{ET-BLACKBOX}\ \frac{\begin{array}{ll} T \in tree(i,k{-}1,q,E,r) & r \xrightarrow{\phi(e)} s \\ [\![e]\!]E = v & c_k..c_j \in \Phi(\phi)(v) \end{array}}{T\langle c_k..c_j\rangle \in tree(i,j,q,E,s)}$$

$$\text{ET-CALL}\ \frac{T \in tree(i,j,q,E,r) \quad r \xrightarrow{\mathbf{call}(e)} s \quad [\![e]\!]E = v}{\epsilon \in tree(j,j,s,[y_0{=}v],s)}$$

$$\text{ET-RETURN}\ \frac{\begin{array}{l} [\![e_1]\!]E_2 = [\![e_2]\!]E_2 = E_1(y_0) = v \quad (x \neq y_0) \\ r \mapsto A \quad t \xrightarrow{\mathbf{call}(e_1)} q \quad t \xrightarrow{x=A(e_2)} u \\ T_1 \in tree(k,j,q,E_1,r) \quad T_2 \in tree(i,k,s,E_2,t) \end{array}}{T_2\, x{:}A(v)\langle T_1\rangle \in tree(i,j,s,E_2[x{=}leaves(T_1)],u)}$$

Figure 7. The Earley Sets

$tree(i,j,q,E,r)$ when that tree is constructed by parsing the input from position $i+1$ to position j. The parse of this subsequence must have begun with the transducer in callee state q and ended with the transducer in state r. Environment E is the environment that was built during the course of the parse.

Figure 7 gives a declarative presentation of our earley algorithm by specifying the trees that belong to each earley set. The first five rules define tree construction when no subtrees are involved and are quite similar to their counterparts (by name) in the definition of the stack evaluation relation. Rule ET-TERM refers to character c_j—the j^{th} symbol in the input string.

Rules ET-CALL and ET-RETURN control the construction of subtrees. ET-CALL adds an empty tree to the forest whose start index matches the current index of the caller and whose callee is related to the current state via a call edge.[4] ET-RETURN finds a parse tree for some nonterminal A (that is, the tree is a member of a forest whose state is a final state for A) and looks for all the potential parents of that tree. They are found via the following criteria: their current position is k, their current state calls the callee state, they transition on A, and the value of A's argument in the context of the parent tree's forest must match the value of the calling argument recorded in the context of the subtree's forest.

Based on the tree sets defined in Figure 7, we define *Earley parsing* as follows:

Definition 11 (Earley Parsing)
If $T \in tree(0,j,q_0,E,r)$ and $r \mapsto A_0$ then $\mathsf{Earley}(c_1 \ldots c_j) = tree(0,j,q_0,E,r)$.

Therefore, we can say that an entire string $w = c_1 \ldots c_n$ is succesfully parsed if $\mathsf{Earley}(w) = S \neq \emptyset$. Moreover, S contains all possible parse trees for w.

We note that our declarative rules do not specify the order in which to construct the Earley sets, and many different orders are possible. The simplest order to use is to build the parse trees in a breadth-first fashion, moving left to right through the input: initialize $tree(0,0,q_0,[y_0{=}()],q_0)$ to ϵ (as specified by rule ET-INIT) and then, for each index j from 0 to the size of the input, apply all rules which add a tree to some forest whose second index is j, until those forests stop changing. There are a number of potential optimizations one could apply to this algorithm, but exploring them is beyond the scope of this paper.

6.2 Correctness

We would like to be sure that our algorithm matches the transducer semantics defined earlier. We therefore show that for every tree derivable in one schema, a corresponding tree is derivable in the other schema. We first show that the Earley algorithm is sound with respect to the stack semantics.

Theorem 25 (Earley Soundness)
If $T \in tree(i,j,q,E,r)$ then there exists tl such that $(q_0,[y_0{=}()],\epsilon,q_0) \Rightarrow^* (q,E,T,r){::}tl$

Proof: by induction on the derivation that $T \in tree(i,j,q,E,r)$.

Next, we show that the Earley algorithm is complete. First, though, we extend the definition of $leaves(\cdot)$ to stacks:

$$leaves((q,E,T,r){::}tl) = leaves(tl)\, leaves(T)$$

Theorem 26 (Earley Completeness)
If $(q_0,[y_0{=}()],\epsilon,q_0) \Rightarrow^* (q,E,T,r){::}tl$, $leaves(tl) = c_1..c_i$, $leaves(T) = c_{i+1}..c_j$, then $T \in tree(i,j,q,E,r)$.

Proof: by induction on height of $\Rightarrow^*$ derivation.

Corollary 27 (Earley Parsing Simulates Transducer Execution)
$T \in \mathsf{Earley}(w)$ iff $(q_0,[y_0{=}()],\epsilon,q_0) \Rightarrow^* (q_0,E,T,r)$ and $r \mapsto A_0$.

6.3 Running Time

We now turn to the issue of the running time and termination of an algorithm implementing the rules of Figure 7, assuming that such an algorithm does not needlessly revisit elements for which all possible rules have already been applied. Given an input string of length n, Earley showed a time bound of $O(n^3)$ for his original algorithm[5] [5]. Our extension enjoys a *pay-as-you-go* property, with the following consequences:

1. In the case that a context-free grammar is specified, we retain the $O(n^3)$ bound, given an efficient representation of tree sets (for example, the binarised Shared Packed Parse Forests (SPPFs) used by Scott [29]).
2. In the case that the full features of our system are used, the algorithm is guaranteed to terminate on all inputs, if (a) all expressions within the grammar terminate, (b) the size of values in environments is bounded, (c) all blackboxes terminate, (d) the size of tree attributes is bounded and (e) tree sets have a finite representation.

[4] The latter criterion is characteristic of Earley's algorithm in that it ensures that a subparse is only attempted if it is "predicted" by the grammar (transducer).

[5] Following Earley, we consider the set insertion step in each rule as a primitive operation, whose complexity is independent of the input [5].

3. In the case that one or more of the above conditions are violated, no guarantees can be made. Note, though, that those are sufficient, but not necessary conditions, because exact behaviour of the algorithm will usually depend upon the particular input.

7. Related Work

Throughout the paper, we have mentioned a number of important related systems—we will not reiterate all of the points of comparison with those systems here. However, please recall the major differences between our system and systems for Generalized LR (GLR) include our support for direct compilation of regular-right sides, attribute-directed parsing and blackboxes. Regarding Parsing Expression Grammars (PEGs), we are additionally distinguished by the compositionality properties of our formalism. These compositionality properties also distinguish us from the various data description languages such as PADS [7, 22] and the Data Description Calculus formalism [8].

Attribute grammars (AGs) are a very powerful extension of context-free grammars originally proposed by Knuth for defining the semantics of programming languages [18]. Much work in AGs has been devoted to finding tractable and efficient restrictions, such as those based on LR or LL grammars [17, 26]. Within attribute grammars, our calculus corresponds most closely to the L-attributed grammars [21]; our nonterminal parameters correspond to inherited attributes, and our environments and bindings overlap with synthesized attributes. Watt introduced the idea of directed parsing [33], and applied it to the LR fragment of context-free languages. Correa [3] and Tokuda and Watanabe [31] have extended Earley's algorithm to L-attributed grammars, though omitting features such as our environments, regular right sides, and blackboxes; Correa implemented attribute-directed parsing.

Woods' augmented-transition networks (ATNs) [34] are an automaton formalism closely related to our data-dependent automata. They support all context-free languages, regular right-hand sides and attribute-directed parsing. Moreover, Chou and Fu describe an Earley-style algorithm capable of parsing with ATNs [2]. However, they differ in a number of subtle, yet important, details. First, ATNs are lower-level than our transducers (for example, requiring explicit stack manipulation to handle call arguments and return values) and are specified directly, rather than with a grammar which can be compiled into an ATN. Woods does not present any such high-level grammar formalism, nor state or prove any correspondence to some existing grammar formalism, as we have. In addition, ATNs do not support merging the automata of multiple nonterminals, because final states are not labeled with their corresponding nonterminal. ATNs do not support blackboxes, although they could be extended to do so in the same way as we have done in our formalism. Finally, to the best of our knowledge, the literature on ATNs does not include proofs of correctness of the Earley algorithm with respect to a transducer semantics.

Finally, as we have mentioned before in the paper, this paper builds on our previous work on Earley parsing for context-free grammars with regular right sides [15], extending it to handle attribute-directed parsing and blackboxes. Also new in this paper is the presentation of a comprehensive meta-theoretic framework in which we show how to prove the correspondence between grammars, transducers and Earley parsing.

8. Conclusion

Modern programmers require modern parser generators. Parsing is still very much an essential element of software systems in nearly every area of software development, yet the technology underlying the most common tools is outdated and the tools, therefore, largely irrelevant. Promising advances are still being made in support of full context-free grammars, most notably surrounding the GLR algorithm. Yet, we believe, and have attempted to demonstrate with a number of examples, that even support for all context-free grammars is not enough for many mundane parsing tasks, particularly in the area of systems programming. Features like scannerless parsing, data-dependence, and blackbox support are crucial to meet the many and varied demands of modern programmers.

We have presented a concise formalism which incorporates all of these features into one framework. We have demonstrated the utility and necessity of its features with a variety of examples and formalized its syntax and semantics. We have also presented and formalized the novel *data-dependent automata*, which are capable of parsing the languages of data-dependent grammars. We have specified sufficient conditions under which an automaton can be said to implement a grammar, proven that under those conditions the language of the automaton matches the language of the grammar, and presented an example compilation from grammars to automata that satisfies the sufficient conditions. Finally, we have presented and proven correct an (often) efficient algorithm for parsing with data-dependent automata based on Earley's classic algorithm for parsing the full range of context-free grammars.

Acknowledgments

This material is based upon work supported by the NSF under grants 0612147 and 0615062 and by a gift from Google. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or Google.

References

[1] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2004.

[2] S. M. Chou and K. S. Fu. Transition networks for pattern recognition. Technical Report TR–EE–75–39, School for Electrical Engineering, Purdue University, West Lafayette, IN, 1975.

[3] N. Correa. An extension of Earley's algorithm for S- and L-attributed grammars. In *Proc. IntlĊonf. on Current Issues in Computational Linguistics*, Penang, Malaysia, 1991.

[4] M. Crispin. Internet Message Access Protocol — Version 4rev1. `http://www.ietf.org/rfc/rfc3501.txt`, March 2003.

[5] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[6] Giorgios Economopoulos, Paul Klint, and Jurgen Vinju. Faster scannerless GLR parsing. In *Proceedings of the 18th International Conference on Compiler Construction (CC)*. Springer-Verlag, 2009.

[7] Kathleen Fisher and Robert Gruber. PADS: A domain specific language for processing ad hoc data. In *PLDI*, pages 295–304, 2005.

[8] Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. The next 700 data description languages. In *ACM Symposium on Principles of Programming Languages*, 2006.

[9] Kathleen Fisher, David Walker, Kenny Q. Zhu, and Peter White. From dirt to shovels: Fully automatic tool generation from ad hoc data. In *ACM Symposium on Principles of Programming Languages*, pages 421–434, January 2008.

[10] Bryan Ford. Packrat parsing: simple, powerful, lazy, linear time. In *ACM International Conference on Functional Programming*, pages 36–47. ACM Press, October 2002.

[11] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM Symposium on Principles of Programming Languages*, pages 111–122. ACM Press, January 2004.

[12] Robert Grimm. Practical packrat parsing. Technical Report TR2004–854, New York University, March 2004.

[13] Ian Hickson and David Hyatt. HTML 5: A vocabulary and associated APIs for HTML and XHTML. `http://dev.w3.org/html5/spec/Overview.html#parsing`.

[14] R. John M. Hughes and S. Doaitse Swierstra. Polish parsers, step by step. In *ACM International Conference on Functional Programming*, pages 239–248, New York, NY, USA, 2003. ACM.

[15] Trevor Jim and Yitzhak Mandelbaum. Efficient earley parsing with regular right-hand sides. In *Workshop on Language Descriptions Tools and Applications*, 2009.

[16] S. C. Johnson. Yacc: Yet another compiler compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[17] Neil Jones and Michael Madsen. Attribute-influenced LR parsing. In *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 393–407. Springer Berlin, 1980.

[18] Donald E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, June 1968.

[19] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[20] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jéróme Vouillon. The Objective Caml system release 3.10: Documentation and user's manual, 2007.

[21] P.M. Lewis, D.J. Rosenkrantz, and R.E. Stearns. Attributed translations. *Journal of Computer and System Sciences*, 9(3):279–307, December 1974.

[22] Yitzhak Mandelbaum, Kathleen Fisher, David Walker, Mary Fernandez, and Artem Gleyzer. PADS/ML: A functional data description language. In *ACM Symposium on Principles of Programming Languages*, 2007.

[23] Scott McPeak. Elkhound: A fast, practical GLR parser generator. Technical Report UCS/CSD–2–1214, University of California, Berkeley, 2002.

[24] Scott McPeak and George C. Necula. Elkhound: A fast, practical GLR parser generator. In *Proceedings of Conference on Compiler Constructor*, April 2004.

[25] A. Melnikov. Collected extensions to IMAP4 ABNF. `http://www.ietf.org/rfc/rfc4466.txt`, April 2006.

[26] Karel Müller. Attribute-directed top-down parsing. In *Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*, pages 37–43. Springer Berlin, 1992.

[27] Eric Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley Professional, October 2000.

[28] D. J. Salomon and G. V. Cormack. Scannerless NSLR(1) parsing of programming languages. In *PLDI*, pages 170–178, 1989.

[29] Elizabeth Scott. SPPF-style parsing from Earley recognisers. In *Proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA)*, March 2007.

[30] Elizabeth Scott and Adrian Johnstone. Right nulled GLR parsers. *ACM Trans. Program. Lang. Syst.*, 28(4):577–618, 2006.

[31] Takehiro Tokuda and Yoshimichi Watanabe. An attribute evaluation of context-free languages. *Information Processing Letters*, 52(2):91–98, October 1994.

[32] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[33] David Watt. Rule splitting and attribute-directed parsing. In *Semantics-Directed Compiler Generation*, Lecture Notes in Computer Science, pages 363–392. Springer Berlin, 1980.

[34] W. A. Woods. Transition network grammars for natural language analysis. *Communications of the ACM*, 13(10):591–606, 1970.