YakYak: Parsing with Logical Side Constraints

Niels Damgaard *
damgaard@brics.dk

Nils Klarlund†
klarlund@research.att.com

Michael I. Schwartzbach*
mis@brics.dk

Abstract

Programming language syntax is often described by means of a context-free grammar, which is restricted by constraints programmed into the action code associated with productions. Without such code, the grammar would explode in size if it were to describe the same language.

We present the tool YakYak, which extends Yacc with first-order logic for specifying constraints that are regular tree languages. Concise formulas about the parse tree replace explicit programming, and they are turned into canonical attribute grammars through tree automata calculations. YakYak is implemented as a preprocessor for Yacc, in which the transitions of the calculated tree automata are merged into the action code. We provide both practical experience and theoretical evidence that the YakYak approach results in fast and concisely specified parsers.

1 Introduction

We introduce a declarative notation, a first-order logic on trees, to specify efficient parsers that require fewer attributes and less explicit programming than conventional parser generators. Our idea is to add concise specifications of regular tree language constraints to the popular parser generator Yacc. Such generators are usually supplied with a relatively simple context-free grammar that must be augmented with explicitly programmed attribute calculations in action code. These calculations often do nothing but calculate regular constraints.

In this paper, we show in theory and in practice how tree automata calculations make it possible to use a declarative first-order language to specify such constraints, in a complete analogy with how usual regular expressions are used to express finite-state automata on strings. Thus, the declarative component of Yacc, namely the context-free grammar, can be augmented with regular tree language constraints with little run time penalty.

Grammars and Side Constraints

Consider the following example grammar, which generates a tiny subset of the HTML notation that we shall call HTML0:

```
H : H E
  | E
;
E : <a href=url> H </a>
  | <b> H </b>
```

*BRICS, University of Aarhus, Ny Munkegade, DK-8000 Aarhus C, Denmark.
†AT&T Labs–Research, Shannon Labs, 180 Park Ave., Florham Park, NJ 07932.

```
   | <i> H </i>
   | <ul> L </ul>
   | word
;
L : /* empty */
   | L <li> H
;
```

Here, the set of tokens is $\{<, >,$ a, href, $=$, url, /, b, i, ul, word, li$\}$. There are many reasonable constraints on HTML0 syntax not captured by this grammar. For example, we should disallow nested anchor elements (an *element* is a named parenthetic structure delimited by a *begin tag* $< \cdots >$ and an *end tag* $< \backslash \cdots >$ (sometimes the end tag is omitted)). The resulting context-free language can now be described only by a grammar that doubles in size, since we have to introduce nonterminals H', E', and L' that cannot generate anchor elements:

```
H : H E                       H': H' E'
  | E                           | E'
;                             ;
E : <a href=url> H' </a>      E': <b> H' </b>
  | <b> H </b>                |  <i> H' </i>
  | <i> H </i>                |  <ul> L' </ul>
  | <ul> L </ul>              |  word
  | word                      ;
;
L : /* empty */               L': /* empty */
  | L <li> H                  |  L' <li> H'
;                             ;
```

As another example, the simple constraint "lists may be nested only to a depth of three" would require 12 nonterminals.

An even more interesting constraint is the stylistic requirement that if any part of some anchor text is in boldface then all anchor texts must be boldface in their entirety. A grammar capturing this language is quite complex, clearly something that is impractical to spell out. If we furthermore impose all three constraints simultaneously, then their respective nonterminals interfere and we are landed with a unwieldy grammar (with more than 100 nonterminals) that is almost impossible to maintain.

In such a situation, most language implementors would prefer to construct a parser for the simple base grammar and then to program the constraints by hand. But this approach is not declarative and concise; the code could not easily be part of the formal specification of the language; the code could be wrong; and it would have to be carefully rewritten if the representation of the parse trees changes.

Parse Tree Logic

We propose an alternative approach where the underlying grammar is defined in the usual way, but the constraints are specified in a concise, formal logic on parse trees. In [7], we presented such a logic, called CDL, and demonstrated that it could be useful for capturing design constraints for object-oriented programs. We developed a prototype implementation that could transform constraints into simple attribute grammars that were intended to be included into syntax-directed editors.

In the present work, we extend Yacc with constraints written in a similar logic. A traditional parser is then automatically generated, where the grammar part is translated as usual into an LR-parser, while each constraint is translated into a deterministic, bottom-

up tree automaton. Our use of a formal logic with unrestricted quantification is much more succinct than the direct use of attribute grammars. In particular, the specification of a constraint does not involve a laborious encoding of the flow of attribute values up and down the tree. Rather, the formulas are translated into tree automata by the Mona tool according to a decision procedure recently implemented [8]. The tree automata represent low-level attribute grammars, which detail the minimum information flow implicit in the formulas across the set of nonterminals.

The use of automata is essential to good runtime performance. Once the automata have been calculated (a process that in our framework may take several minutes), the resulting parser calculates in (almost) linear time the behavior of the automata on the input of a tree. If there are n constraints, then the resulting parser makes moves in n automata, each of which can be calculated in a few microseconds.

Related Work

To our knowledge, the idea of using a first-order like parse tree logic to generate attribute grammars has been presented earlier only in our previous paper[7], which suggested how such a logic can be used to enforce design constraints or software architectures. There, we used an encoding of grammars that results in an inherent quadratic blowup, see Section 4. Similar ideas of using parse tree logic have been pursued in formal linguistics, see [11, 14] and in computer science logic [1], but no practical applications have been demonstrated.

Earlier work on the practical use of attribute grammars, like [13], tends to focus on minimizing calculations under more general circumstances. In contrast, our work deals with the generation of minimum grammars for the restricted class of synthesized attributes over finite domains.

Recently, work within the W3C promotes the use of simple side constraints on parse trees. For example, XML[3] offers a simple notation, based on regular expressions, for restrictions on the occurrence of subelements within an element. Our notation is strictly more expressive.

Extensions of Yacc-like parser generators are too numerous to mention—a quick search on the web found 69 different implementations. They focus on supporting different target languages, handling EBNF notation, coping with larger classes of grammars, adding attribute evaluations, or automatically building syntax trees. The ideas in YakYak could be incorporated into all such proposals. Note that the approach is not restricted to LALR parser generators even though our actual implementation is based on Bison.

Several other logic notations have been proposed for parsing. Definite Clause Grammars [12] elegantly express both synthesized and inherited attributes. The backtracking nature of the semantics may result in poor runtime performance, including lack of termination. ASTLOG[4] is another Prolog inspired programming notation, where for efficiency reasons the parse tree is handled as a separate semantic object. Despite its declarative look, this language is also Turing-complete, even if it in many cases result in reasonably efficient parsers. The GENOA system[5] provides a scripting language dedicated to the description of parse trees. A fragment of the notation expresses precisely PTIME parse tree analysis programs. The reference [5] also discusses many other similar systems.

It appears that all such systems are less declarative than ours in the sense that they explicitly model the information flow up and down the tree. Neither do they guarantee linear run time performance of the generated parser. On the other hand, they are far more expressive than our constraint formalism, which covers only the regular tree languages.

Plan

In Section 2, we discuss grammars and the parse tree logic; we formulate a simple type system for formulas; and we discuss the strength of our notation. Next, we show in Section 3 the concrete syntactic extensions that YakYak adds to Yacc. In Section 4, we provide an explanation of the WS2S logic that is at the heart of the parse tree formulation and our implementation; also, we formulate our main technical result that there is automaton representation of well-formed parse trees that is linear in the size of the grammar. In Section 5, we show how techniques are put together so that an ordinary Yacc specification can be generated from a YakYak specification. Finally in Section 6, we discuss several practical experiments, including compile time and run time statistics.

2 Grammars and Constraints

In the following, we need to talk about the various components of a Yacc-like grammar: the root nonterminal is denoted root, the number of productions of the nonterminal N is denoted $|N|$, and the i'th symbol in the right-hand side of the j'th production of the nonterminal N is denoted $N(j,i)$. For the base HTML0 grammar above, we have that root=H, |E|=5, and L(2,5)=H.

Parse Tree Logic

The *parse tree logic* is a first-order logic that is interpreted over parse trees: first-order terms denote nodes in trees and a formula is either true or false for a given tree. The syntax involves *terms*, *term types*, and *formulas*. Terms denote nodes in a parse tree:

t : \$\$	the root
$\mid t.i$	the i'th child node of t
$\mid \alpha$	a first-order variable

A term type describes a set of nodes in a tree:

$\tau : N$	any production of nonterminal N
$\mid N[j]$	the j'th production of nonterminal N

A formula assigns a truth-value to a given tree:

$\phi : t_1 < t_2$	ancestor relation
$\mid t_1 = t_2$	equality
$\mid \neg\phi$	negation
$\mid \phi_1 \Rightarrow \phi_2$	implication
$\mid \phi_1 \wedge \phi_2$	conjunction
$\mid \phi_1 \vee \phi_2$	disjunction
$\mid \exists\alpha{:}\tau.\phi$	existential quantification
$\mid \forall\alpha{:}\tau.\phi$	universal quantification

The constraint that a parse tree in the HTML0 grammar does not have nested anchors is expressed as:

$$\forall \mathsf{a}{:}\mathsf{E}[1].\neg\exists \mathsf{b}{:}\mathsf{E}[1].\mathsf{a} < \mathsf{b}$$

$$\frac{}{\Gamma \vdash \mathsf{\$\$} : \mathsf{root}} \qquad \frac{}{\ldots,\alpha{:}\tau,\ldots \vdash \alpha{:}\tau} \qquad \frac{\Gamma \vdash t : N[j]}{\Gamma \vdash t.i : M} \; \textit{if } N(j,i) = M$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1 < t_2} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1 = t_2} \qquad \frac{\Gamma \vdash t : N}{\Gamma \vdash t.i : M} \; \textit{if } \forall j \in 1..|N| : N(j,i) = M$$

$$\frac{\Gamma \vdash \phi}{\Gamma \vdash \neg\phi} \qquad \frac{\Gamma \vdash \phi_1 \quad \Gamma \vdash \phi_2}{\Gamma \vdash \phi_1 \Rightarrow \phi_2} \qquad \frac{\Gamma \vdash \phi_1 \quad \Gamma \vdash \phi_2}{\Gamma \vdash \phi_1 \wedge \phi_2}$$

$$\frac{\Gamma \vdash \phi_1 \quad \Gamma \vdash \phi_2}{\Gamma \vdash \phi_1 \vee \phi_2} \qquad \frac{\Gamma, \alpha : \tau \vdash \phi}{\Gamma \vdash \exists \alpha : \tau.\phi} \qquad \frac{\Gamma, \alpha : \tau \vdash \phi}{\Gamma \vdash \forall \alpha : \tau.\phi}$$

Figure 1: Type Inference Rules

Nodes of type E[1] are anchors, and the formula simply states that no such node can appear below another of the same kind.

The constraint that lists are nested to at most depth three is expressed as:

$$\neg\exists\, \mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d} : \mathsf{E[4]}.\mathsf{a} < \mathsf{b} \wedge \mathsf{b} < \mathsf{c} \wedge \mathsf{c} < \mathsf{d}$$

Here, nodes of type E[4] are lists, and we forbid chains of length four.

Finally, the more intricate constraint that "if any part of an anchor text is in boldface then all anchor texts must be entirely in boldface;" is expressed as:

$$\begin{array}{l} \quad \exists \mathsf{a} : \mathsf{E[1]}.\exists \mathsf{b} : \mathsf{E[2]}.\exists \mathsf{w} : \mathsf{E[5]}.\mathsf{a} < \mathsf{w} \wedge \mathsf{b} < \mathsf{w} \\ \Downarrow \\ \quad \forall \mathsf{w} : \mathsf{E[5]}.(\exists \mathsf{a} : \mathsf{E[1]}.\mathsf{a} < \mathsf{w}) \Rightarrow (\exists \mathsf{b} : \mathsf{E[2]}.\mathsf{b} < \mathsf{w}) \end{array}$$

The antecedent of the implication states that there is some anchor node a and a word w contained in a, and there is a boldface element b containing w. Notice that it may be either the case that the boldface element encloses the anchor element or the case that the anchor element encloses the boldface element. The consequent of the implication is that any word that is enclosed in an anchor element is also enclosed in a boldface element.

Type System

A formula is required to be *well-typed*, which is determined by the inference rules in Figure 1. The notation $\Gamma \vdash \phi$ means that ϕ is well-typed in the environment Γ which assigns types to free variables. The notation $\Gamma \vdash t : \tau$ means that t is well-typed in Γ and has type τ. A well-typed formula has the property that all term expressions perform only sensible navigations in trees.

An example of an ill-typed term over the basic HTML0 grammar that is caught by the type checker is $$.1, since the two H-productions have different first symbols.

Expressive Power and External Predicates

For reason of simplicity, we omitted *monadic second-order variables* from the logic presented above, even though they are more general than first-order variables, as discussed in Section 4. With these variables, it can be shown that the parse tree logic exactly captures the class of all regular tree languages. This class is usually defined as the set of languages that are accepted by a finite-state tree automaton. It follows that a grammar restricted

by parse tree formulas still expresses a context-free language, albeit a language that may require an explosive amount of nonterminals if expressed alone by a context-free grammar.

So although parse tree constraints extend the class of languages that may be conveniently expressed in a declarative manner, it is often desirable to impose constraints that go beyond context-freeness. A telling, if contrived, example is to require that all integers appearing in boldface must be prime numbers. For that purpose, we could introduce two *external* predicates Num and Prime that decide if a word is a numeral and if a numeral is a prime number. The required formula is then:

$$\forall \mathsf{b}:\mathsf{E}[2].\forall \mathsf{w}:\mathsf{E}[5].(\mathsf{b}<\mathsf{w} \wedge \mathsf{Num}(\mathsf{w}.1)) \Rightarrow \mathsf{Prime}(\mathsf{w}.1)$$

The semantics of external predicates must be implemented in conventional C-code.

3 The YakYak Language

YakYak is an extension of Yacc. An example is:

```
%left '+' '-'
%left '*' '/'

%constraint all p,q:exp[1].p==q

exp : exp '+' exp
    | exp '-' exp
    | exp '*' exp
    | exp '/' exp
    | identifier
    | intconst
;
```

The %constraint is a global requirement. If it is violated, then yyerror() is invoked with a suitable error message. The above constraint forbids the occurrence of more than one plus operator. For some applications, we may want to act on the truth-value of a formula rather than to generate a parsing error. For this use, one can write:

```
exp : exp '+' exp
    | exp '-' exp   [ex p:exp[6].Prime(p.1)]
    | exp '*' exp
    | exp '/' exp
    | identifier
    | intconst
;
```

The formula is written in square brackets and is placed immediately before the action code. The requirement is imposed only for the subtrees rooted by that particular production. The truth-value (0 or 1) of the formula for the current subtree is available inside the action code as the value of the variable $4—corresponding to the index of the formula in the right-hand side of the rule.

The above formula decides if a prime number constant appears anywhere inside the arguments of the minus operator. The predicate Prime is external, and to make it known to YakYak we must declare:

```
%predicate Prime(int i);
```

since the %type of intconst is declared to be int. The implementation of the Prime predicate must be available at link time.

```
term : "$$"                     formula : term relop term                logop : && | || | => | <=>
     | term "." number                  | id "(" term ")"                ;
     | id                               | "!" formula                    relop : < | == | != | > | <= | >=
;                                       | formula logop formula          ;
type : id                               | "all" id ":" type "." formula
     | id "[" number "]"                | "ex" id ":" type "." formula
;                                       | "(" formula ")"
                                        ;
```

Figure 2: YakYak Syntax for Parse Tree Logic

To stay in the spirit of Yacc, we have adopted a suitably C-like syntax of formulas shown in Figure 2. In this syntax, the tiny HTML0 example looks like:

```
%constraint   all a:elm[1].!ex b:elm[1].a<b
%constraint   !ex a,b,c,d:elm[4].a<b && b<c && c<d
%constraint   (ex a:elm[1].ex b:elm[2].ex w:elm[5].a<w && b<w)
                => (all w:elm[5].(ex a:elm[1].a<w) => (ex b:elm[2].b<w))
%%

html : html elm
     | elm
;
elm  : '<' 'a' HREF '=' url '>' html '<' '/' 'a' '>'
     | '<' 'b' '>' html '<' '/' 'b' '>'
     | '<' 'i' '>' html '<' '/' 'i' '>'
     | '<' UL '>' list '<' '/' UL '>'
     | WORD
;
list : /* empty */
     | list '<' LI '>' html
;
```

4 Logic and Tree Automata

Formulas over parse trees can be reduced to tree automata if we encode parse trees as simpler, labeled binary trees. First, we explain the connection between a simple logic, WS2S, on binary trees and tree automata. Next, we discuss how this connection can be extended in various ways to accommodate parse tree logic. To explain our encoding in more detail, we then formalize the notion of automata and their languages, and we present the conventional encoding and our new *shape encoding*. Finally, we show how the shape encoding is efficiently supported by a special kind of tree automaton.

WS2S and Tree Automata

WS2S (Weak Second-order theory of 2 Successors) is a logic that in its most simple form consists of formulas containing quantifiers, various set comparisons ($\supseteq, =, \ldots$) and monadic second-order variables that range over finite subsets of the infinite, binary tree. First-order variables, like the ones in parse tree logic, are treated as singleton second-order variables, since singletoness can easily be encoded. The validity status of any WS2S formula can in principle be checked by an automata-theoretic decision procedure, see [6, 15]. The decision procedure works by inductively calculating a tree automaton for each subformula. The language accepted by the automaton is exactly the interpretations that make

the formula hold. (Recall that an *interpretation* is an assignment of values to the free variables.) In this manner, each logical connective corresponds to an automata-theoretic operation; for example, existential qualification corresponds to the subset construction for tree automata. The Mona tool provides an efficient implementation in the sense that automata with thousands of states can be handled (although this is not always enough); also, Mona uses binary decision diagrams to cope with large alphabets.

The decidability of WS2S rests on the simple observation that any subset of the infinite, binary tree is expressible as a Boolean labeling, where each node is flagged according to whether it is in the subset. A number ℓ of subsets can be expressed by a single labeling with Boolean vectors in $\mathbf{B}^\ell$: a node v is in the i'th subset if and only if the i'th component of the label of v is 1. We regard $\mathbf{B}^\ell$ as the alphabet of a tree automaton that arises with any formula as follows. Take the WS2S formula $X \supseteq Y$ as an example; it expresses that the set of positions X is contained in the set of positions Y, and a particular interpretation of free variables X and Y can be coded as a labeling of the infinite tree with labels in $\mathbf{B}^2$. In WS2S, X and Y are required to denote finite subsets only. A tree automaton can easily be exhibited that will read, in a deterministic and bottom-up manner, a finite labeled tree T such that the state reached at the root is a final state if and only if $X \supseteq Y$ holds, where X and Y are interpreted by the labeling of T. This automaton has only two states: the accepting state, which the automaton stays in as long as it has not seen a letter $(0,1)$ in some node v (such a letter means that $v \in Y$ but $v \notin X$, when we assume that the first component encodes X and the second Y), and a reject state. The decision procedure details how this example can be extended to the construction of an automaton for each formula.

A Simple Case of Parse Tree Logic

If our parse trees are just binary trees generated by a single, recursive production, we could express the tree itself in WS2S by a free second-order variable T. A formula WF can be written that asserts that T is indeed a subset of nodes that constitute a parse tree. WF must ensure that any node in T either is a leaf (i.e. no child is in T) or corresponds to a recursive production (i.e. both children are in T). Also, a predicate that distinguishes between internal nodes and leaves can easily be defined. Any parse tree formula for this grammar can then be expressed as a WS2S formula with T as the only free variable. The corresponding automaton calculated according to the decision procedure then constitutes an attribute grammar with one synthesized attribute that ranges over automata states.

Molding Grammars into Binary Trees

To decide the parse tree logic, we could construct a tree automaton concept that directly reflects the heterogeneous nature of grammars. Such automata would read parse trees, labeled with Boolean vectors representing free variables, in a bottom-up manner like the binary tree automata just discussed. However, their implementation would be very complicated and suffer from table size explosion. First, there would be many kinds of transition functions, one for each nonterminal. Second, each transition function word would involve multi-dimensional arrays with a number of entries exponential in the maximum number of nonterminals in a right hand side of a production.

Instead, we consider here morphing the variably-branching grammar into an efficient binary tree framework so that we can use the Mona tool to carry out the tree automata calculations. Thus, we want to encode a grammar $\mathcal{G}$ over some *coding alphabet* Σ such

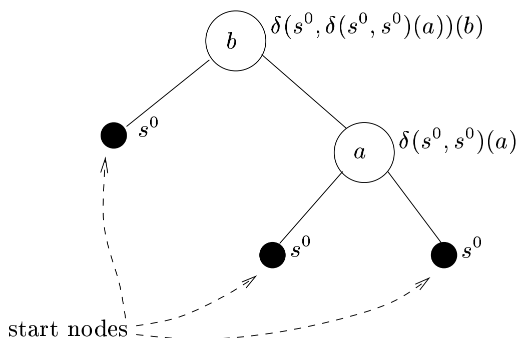
that any parse tree is uniquely represented as a finite, Σ-labeled, binary tree. Also, we want each node in the parse tree to correspond to a node in the binary tree. However, it will not be possible to maintain an inverse correspondence, since some nodes in the binary tree act as intermediate nodes. Thus, any property of a parse tree, can be represented as a subset of the nodes in the encoded binary tree. In this way, a tree automaton calculating the truth-status of a formula will read a $\Sigma \times \mathbf{B}^\ell$-labeled tree.

Note that the encoding of the grammar determines a set $\mathsf{WF}(\mathcal{G})$ of *well-formed* Σ-*labeled, binary trees*, namely those that correspond to actual parse trees. The WF-*automaton* is the canonical automaton that recognizes this set. The WF-automaton is essential to the WS2S translation of formulas, since any parse tree formula over Σ-labeled trees should be dependent only on the values of well-formed Σ-labeled, binary trees. For example, if a programmer has specified a constraint in the parse tree logic that is translated into a formula ϕ on binary trees, and if $\phi \Rightarrow \neg\mathsf{WF}$ holds (presumably unbeknownst to the programmer), then the canonical automaton should not be that of ϕ for that automaton could be arbitrarily big. Instead, we want a canonical automaton that expresses the property "false" under the assumption WF. In fact, we choose to normalize all automata, also the intermediate ones corresponding to subformulas ψ, so that each automaton represents the formula $\mathsf{WF} \wedge \psi$.

Trees and Automata

At this point, we need to make our notions more precise. A *binary tree* T is a prefix-closed subset of $\mathbf{B}^*$. A *node* $v \in T$ is a sequence of *successors*: 0 is called the *left* successor and 1 the *right*. The empty string ϵ is called the *root*. T is a Σ-labeled tree if it is equipped with a mapping $\chi : T \to \Sigma$. A tree automaton $\mathcal{A} = (\Sigma, S, s^0, F, \delta)$ consists of an alphabet Σ, a finite state space S, an initial state $s^0 \in S$, a set of final states $F \subseteq S$, and a transition function $\delta : (S \times S) \to \Sigma \to S$. A *run* over a labeled tree (T, χ) is an assignment of states to the nodes in T and to extra *start nodes* of the form $v \cdot b$, where $v \in T$, $b \in \mathbf{B}$, and $v \cdot b \notin T$, such that (1) any start node is assigned s^0, and (2) for any node $v \in T$, v is assigned to state $\delta(s', s'')(\chi(v))$, where s' and s'' are the states assigned to the left and right successor of v. The run so defined is unique for any labeled tree. The tree is *accepted* if and only if the state assigned to the root is in F. The language $L(\mathcal{A})$ accepted by $\mathcal{A}$ is the set of all trees (T, χ) accepted. The class of *regular tree languages* (over binary trees) is the class of all $L(\mathcal{A})$. The *size* of a tree automaton is $|S|^2$. This definition is reasonable since the alphabet-part of the transition relation can often be compressed asymptotically by the BDD representation, see [6].

For example, if $\Sigma = \{a, b\}$, then a two-node Σ-labeled tree could look like:

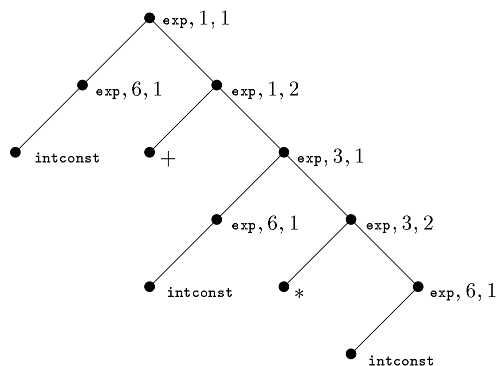


It contains ordinary two nodes: ϵ, labeled b, and 1 labeled a. The figure also shows how the states of a run over the tree are calculated; note that the auxiliary start nodes (0, 10, and 11) are shown in black.

A Conventional Encoding

We now outline a traditional encoding of parse trees over a binary tree, and we argue that the asymptotic complexity is quadratic in the size of the grammar. For each nonterminal N and each production j of N, let $|N[j]|$ be the number of terminals and nonterminals in the right hand side (r.h.s.) of the jth production. We let the coding alphabet Σ consist of all terminal symbols; in addition, for each production $N[j]$, we need a symbol (N, j, i) for each position in the right hand side (except the last). A node v, in the binary tree, that corresponds to a nonterminal N is labeled with $(N, j, 1)$, where j is the number of the production used to rewrite N. Its left child, $v \cdot 0$ is labeled according to the first symbol of the right hand side of the production. If $|N[j]| = 2$, then the right child is labeled according to the second one; otherwise, disregarding the case of $|N[j]| = 1$, we make the right child $v \cdot 1$ an intermediate node labeled $(N, j, 2)$. The left child $v \cdot 10$ of $v \cdot 1$ is labeled according to the second nonterminal, and the right child $v \cdot 11$ is labeled according to the third nonterminal if $|N[j]| = 3$; otherwise, the right child is labeled with $(N, j, 3)$, and so forth.

As an example, consider the expression grammar from Section 3. The string `2+3*4` would yield a parse tree whose binary representation is:



It is natural to define the size of a grammar $\mathcal{G}$ as the total number of occurrences of terminal and nonterminal symbols in its productions. Thus, the size of the encoding alphabet Σ is approximately equal to that of the grammar.

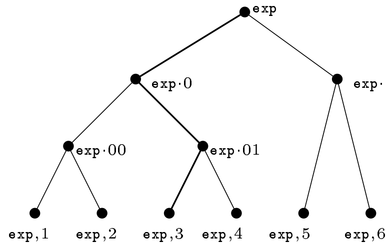
It can now be shown that a tree automaton $\mathcal{A}$ with $|\Sigma| + 2$ states can be constructed such that a labeled, binary tree satisfies WF if and only if it is accepted by $\mathcal{A}$. In fact, the labeling of the intermediate nodes can be shown to prevent exponential explosions in the state space: we may choose the state space to be simply the alphabet Σ itself plus a couple of auxiliary states whereas without such labeling, the automaton would possibly have to process many possible right hand sides at the same time until the nonterminal was reached. (These arguments are standard in automata theory and can be shown through Myhill-Nerode congruence arguments; see [9]). The automaton that we have described is minimum. Thus, we have the following result:

Proposition 1 *Under the conventional encoding, the size of a tree automaton recognizing the well-formedness predicate is quadratic in the size of the grammar.*

This is the encoding we used in the CDL design constraint tool[7], and which sometimes prevented it from scaling to interesting grammars.

A Shape Encoding

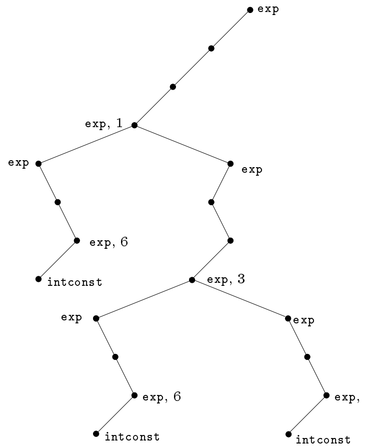
We introduce here a representation that is by shape only. Thus, given a grammar $\mathcal{G}$, we define a subset WF of the finite, binary trees such that there is a bijective correspondence between parse trees of $\mathcal{G}$ and WF. The idea is simple: instead of denoting the choice of a production by a label, we denote it by a sequence of successors, that is, a path. For example, consider the nonterminal `exp`. We observe that there are six production; let us call them $(\texttt{exp}, 1)$,..., $(\texttt{exp}, 6)$. The six choices can be encoded as the sets of *full paths* (those from the root to the leaves) of a tree:



We call this the OR-*tree* for `exp`. With this OR-tree, the production for `*` corresponds to to the path 010, which is shown in bold above. To explain the shape encoding, we still need labels. Above, the root receives the name of the nonterminal, namely `exp`; the intermediate nodes receive a name of the form $(\texttt{exp} \cdot \alpha)$, where α is a sequence of successors; and the leaves receive the names of the productions as just defined. In the shape representation, the right-hand side of each production will similarly be molded into a binary AND-*tree* as for the conventional encoding.

A parse tree is now converted into a binary tree as follows. Label the root of the singleton, binary tree with the start symbol. Select one full path in the OR-tree for the start symbol and add that path to the root. Then, the leaf of the path is labeled with a production name. Expand the right-hand side of that production into an AND-tree, the leaves of which denote terminals or nonterminals. For each nonterminal leaf, proceed inductively as if it were a start symbol.

For example, the parse tree for `2+3*4` is unraveled into the binary tree fragment:



Here, we have for the sake of clarity omitted the labels of intermediate nodes.

The set of well-formed trees has been designed to enjoy the property: any node $v \in \mathbf{B}^*$ always has the same label whenever it occurs in a shape representation of a parse tree. This property is easily proved by induction. Thus, we can a priori label the complete, infinite, binary tree; descendants of nodes labeled with terminals do not receive a label. And, any well-formed tree is then completely specified as a subset of the infinite, binary tree. Moreover, we note that a deterministic, top-down automaton can calculate the labeling of the tree: its state space is the set of labels, the initial state is the start symbol, and the transitions are defined according to the local OR- and AND-trees. We call it the *labeling automaton* $\mathcal{D}$. It consists of state space D, which is identical to the set of labels, root label d^0, and a set of transitions of the form $d \rightarrow (d', d'')$, one for each $d \in D$. For example, the labeling automaton for the expression grammar contains a transition $\texttt{exp} \rightarrow (\texttt{exp} \cdot 0, \texttt{exp} \cdot 1)$. Each such transition will be called a *transition type* to distinguish it from usual automata transitions.

Guided Tree Automata

The well-formedness property of the shape encoding can defined in WS2S, since both the behavior of the labeling automaton as well as the various requirements on paths can be expressed. But the size of the WF-automaton is potentially exponential in the size of the grammar. The reason is that the automaton may have to maintain several assumptions about what is the meaning of the subtree it has read, since it has been deprived of the explicit labels used in the traditional encoding.

However, the labeling permits a factorization of the state spaces of any automaton recognizing a regular tree language L over alphabet Σ if we let each label d correspond to a separate tree automaton state space S_d. The state space S_d is used only in the positions labeled with d. In particular, if $\mathbf{T}(\Sigma)$ is the set of Σ-labeled finite trees and the congruence $\sim^d_{\mathcal{D}}$ on $\mathbf{T}(\Sigma)$ is defined as

$$T \sim^d_{L,\mathcal{D}} T' \text{ if and only if for all } C,\ C[T] \in L \Leftrightarrow C[T'] \in L,$$

where C is a *context*, defined as a Σ-labeled tree with a designated leaf and $C[T]$ is the tree C with the designated leaf replaced by T, then the canonical state space for d is the set $\mathbf{T}(\Sigma)/\sim^d_{\mathcal{D}}$ of equivalence classes of $\sim^d_{\mathcal{D}}$. For each of the $|D|$ transition types $d \rightarrow (d', d'')$, a transition function of type $(\mathbf{T}(\Sigma)/\sim^{d'}_{\mathcal{D}} \times \mathbf{T}(\Sigma)/\sim^{d''}_{\mathcal{D}}) \rightarrow \mathbf{T}(\Sigma)/\sim^d_{\mathcal{D}}$ is naturally defined, just as in the case of the Myhill-Nerode regular theorem for string languages. Generalizations of the usual tree automata algorithms, such as the subset construction and minimization, are described in [2], where these partitioned automata are called *guided tree automata*. (They generalize the tree-shaped binary decision diagrams of [10], which assign different state spaces to a fixed, finite tree.) The size of a guided tree automaton is the sum of the table sizes, that is, $\Sigma_{d \rightarrow (d', d'')} |S_{d'}| \cdot |S_{d''}|$. It can now be shown that:

Proposition 2 *Under the shape encoding, the size of a guided tree automaton recognizing the well-formedness predicate is linear in the size of the grammar.*

In fact, each state space has at most four states if the three-valued automata of Mona are used.

```
exp : exp '+' exp [ex p:exp[5].$$.3<=p]
      { if ($4) printf("identifier in second argument"); }

exp : exp '+' exp
      {$$ = make_exp_wrapper();
       x1 = $1->state; x2 = $3->state;
       q1 = trans(auto_exp_1,x1,x2);
       q2 = trans(auto_exp_or_00,q1,auto_exp_2_initial);
       q3 = trans(auto_exp_or_0,q2,auto_exp_or_01_initial);
       q4 = trans(auto_exp,q3,auto_exp_or_1);
       $$->state = q4;
       if (accept_exp($$->state)) printf("identifier in second argument");
      }
```

Figure 3: Original and Instrumented Action Code

5 From YakYak to Yacc

A YakYak specification is transformed into a Yacc specification as follows. The encoding, determining AND and OR trees for productions, is calculated from the context-free grammar, and the labeling automaton is described in a Mona declaration. The WF-automaton is described as a Mona formula, and parse tree formulas are also translated into Mona formulas. This translation is easy, since for any label d Mona offers a built-in predicate that tests whether a first-order variable denotes a node labeled d. Then Mona is run on each parse tree constraint, and the resulting automata are written as C data structures, which become part of the produced Yacc parser. The Yacc productions are augmented with action code that executes transitions of the tree automaton. Usually, several transitions are necessary since the automaton must traverse both an AND-tree and an OR-tree. A sketch of the code before and after instrumentation is shown in Figure 3.

Note that since each node in the parse tree is represented by several nodes in the binary tree representation, there is potentially an induced logarithmic overhead. In practice, however, the branching of the context free grammar is quite limited, so it is reasonable to assume that each parse tree node is treated in constant time.

6 YakYak in Practice

We must investigate three aspects. How expensive is it to transform YakYak specifications into equivalent Yacc specifications? How efficient are the instrumented parsers? And, can we express interesting constraints? To answer these questions, we consider several example grammars:

- EXP: the expression language (2 nonterminals, 8 productions, 2 tokens, size 18).
- HTML0: the HTML example (3 nonterminals, 10 productions, 12 tokens, size 19).
- YAKYAK: the YakYak language itself (24 nonterminals, 79 productions, 52 tokens, size 185).
- JAVA: a large Java subset (66 nonterminals, 144 productions, 65 tokens, size 310).
- HTML3: all of HTML3.0 (147 nonterminals, 368 productions, 204 tokens, size 698).

grammar	size	WF formula	1 formula	2 formulas	3 formulas	10 formulas
EXP	18	1 sec 2 MB 68 states	2 sec 3 MB 100 states	3 sec 4 MB 200 states	4 sec 4 MB 296 states	5 sec 6 MB 837 states
HTML0	19	1 sec 3 MB 77 states	3 sec 7 MB 142 states	4 sec 7 MB 348 states	5 sec 7 MB 468 states	10 sec 9 MB 1,061 states
YAKYAK	185	4 sec 12 MB 603 states	11 sec 27 MB 626 states	21 sec 34 MB 1,269 states	27 sec 34 MB 1,903 states	54 sec 55 MB 6,245 states
JAVA	310	7 sec 18 MB 988 states	39 sec 48 MB 1,086 states	49 sec 53 MB 2,116 states	57 sec 54 MB 3,140 states	160 sec 58 MB 10,757 states
HTML3	698	16 sec 39 MB 2,159 states	19 sec 61 MB 2,531 states	134 sec 117 MB 5,375 states	285 sec 151 MB 9,204 states	584 sec 171 MB 26,974 states

Figure 4: Generating Automata with Mona

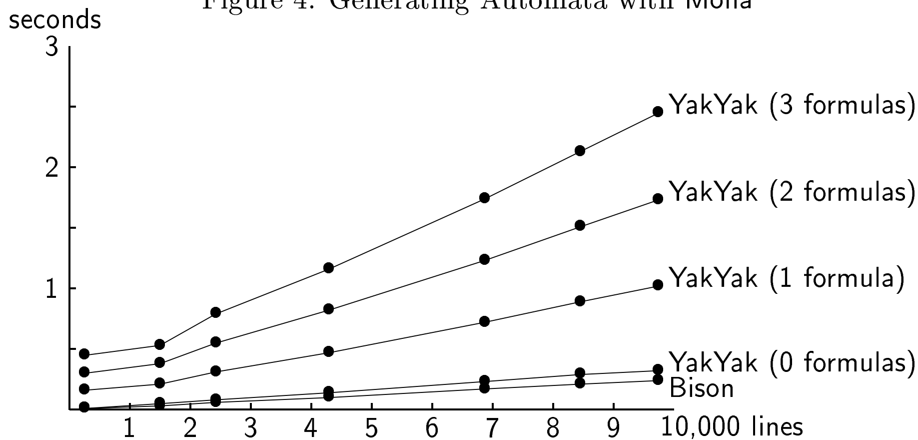


Figure 5: Running Times for Different JAVA Parsers

We have run YakYak on these grammars enriched with a varying number of formulas. The results are summarized in Figure 4, where we for each entry give the running time in seconds (on a 266 MHz Pentium II PC with 128MB RAM), the memory usage, and the total number of states in the resulting automaton. The WF-column shows the computation of the well-formedness automaton, which must be done once for each grammar. It is seen that the size as well as the time and space consumption of this automaton is linear in the size of the grammar. The number of states shown is the total over all state spaces; for example, WF for HTML3 has 2,159 states spread over 560 spaces (for an average of 3.9 states per space, cf. Proposition 2). The other columns total the additional computation for an increasing number of formulas, each of which is similar in spirit to the ones shown previously, involving at most a handful of quantifiers. These experiments clearly demonstrate that our concept is feasible: for the largest grammar, the automata for 10 formulas are computed in less than 10 minutes.

The efficiency of the generated parsers is described in Figure 5, which shows the running times of YakYak parsers for the JAVA grammar with between 0 and 3 formulas. Each parser is given a number of input programs ranging between 100 and 10,000 lines of code. As a baseline, we show the running times for a parser generated by the Bison version of

Yacc. The YakYak parsing times all appear linear in the size of the input programs, but they are somewhat slower than for the Bison version. We are working on optimizing the instrumented action code further. The extra space usage at runtime is not significant.

As we would expect, the complexity of a constraint influences the time needed to compute the corresponding automaton. However, the running time of an automaton is virtually independent of its size, so complicated formulas are checked as quickly as simple ones during parsing. For the present implementation of YakYak, the overhead works out to 74 microseconds/formula/line of code (under the assumption we made at the end of Section 5).

The expressiveness of our parse tree logic is best illustrated by an explanation of some of the formulas in Figure 4. From the JAVA grammar, we mention: "Are the classes of all abstract methods themselves declared to be abstract?" , "Do static methods refer to this or super?", "Does an expression have possible side-effects?", and "Does a void method contain a return with an expression?". Examples from the HTML3 grammar are: " Are there nested anchors?", "Is an anchor text only the word 'here'?", "Are there empty table cells?", " Are there headers inside lists?", and " Does a boldface style extend beyond a paragraph?". These formulas reflect tests performed in Java compilers and suggestions in HTML style guides.

References

[1] Abdelwaheb Ayari, David Basin, and Andreas Podelski. LISA: A specification language based on WS2S. In *CSL '97 Proceedings*, LNCS 1414, 1998.

[2] M. Biehl, N. Klarlund, and T. Rauhe. Algorithms for guided tree automata. In *First International Workshop on Implementing Automata, WIA '96, Lecture Notes in Computer Science, 1260*. Springer Verlag, 1996.

[3] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. *Extensible Markup Language (XML)*, 1997. URL: `http://www.w3.org/TR/PR-xml-971208`.

[4] Roger F. Crew. ASTLOG: a language for examining abstract trees. In *Proceedings of the Conference on Domain-Specific Languages*, pages 229–242. USENIX, 1997.

[5] P Devanbu. GENOA: A language and front-end independent source code analyzer. *ACM Transactions in Software Engineering*, 1999. (to appear).

[6] N. Klarlund. Mona & Fido: the logic-automaton connection in practice. In *CSL '97 Proceedings*. LNCS 1414, Springer-Verlag, 1998.

[7] N. Klarlund, J. Koistinen, and M. Schwartzbach. Formal design constraints. In *Proc. OOPSLA '96*, 1996.

[8] Nils Klarlund and Anders Møller. *MONA Version 1.2 User Manual*. BRICS, ns-98-3 edition, 1998. ISSN 0909-3206.

[9] D. Kozen. On the Myhill-Nerode theorem for trees. *EATCS Bulletin*, 47, 1992.

[10] K. McMillan. Hierarchical representations of discrete functions, with applications to model checking. In *Proc. Computer Aided Verification, LNCS 818*. Springer-Verlag, 1994.

[11] Frank Morawietz and Tom Cornell. The logic-automaton connection in linguistics. In *Proceedings of LACL 1997*, LNAI. Springer, To appear.

[12] F. Pereira and D. Warren. Definite Clause Grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Journal of Artificial Intelligence*, 13:231–278, 1980.

[13] T. Reps. *Generating Language-Based Environments*. The M.I.T. Press, 1984.

[14] James Rogers. *Studies in the logic of trees with applications to grammar formalisms*. PhD thesis, University of Delaware, 1994.

[15] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. MIT Press/Elsevier, 1990.