



LR automaták

Szakdolgozat

Adrián Patrik

Témavezető:
dr. Falucskai János

Nyíregyházi Egyetem

2022

Abstract

Formal language theory is a fundamental field of computer science that is of substantial academic and practical interest. Despite the vast body of existing literature, most publicly available parser implementations leave a lot to be desired, especially in the presence of nondeterminism. Ambiguities are often considered to be design errors rather than non-context-free constructs in need of further processing.

Tomita's Generalized LR (GLR) and its derivatives are some of the most commonly used algorithms for the interpretation of ambiguous texts. Exponential growth is curbed to polynomial order by sharing parser paths and representation of derivations. The original GLR parser is incapable of handling all grammars with ϵ -rules. Various corrections, including [29], [33] and [45] have been developed.

We describe a variant of GLR which combines previous efforts ([33], [46]) into a single algorithm. The resulting parser, which we call Nullable-eliminated GLR (NEGLR), is a simple, clean implementation of the GLR technique that admits every context-free grammar with at worst cubic worst-case complexity and is linear on $LR(k)$ grammars.

We also investigate the applicability of this technique to extended context-free grammars, where the productions are described using regular expressions. We introduce the ENEGLR parser, along with its supporting machinery, which can perform generalized extended context-free syntax analysis with the same worst-case bounds.

NYILATKOZAT

Alulírott **Adrián Patrik** (Neptun-kód: ZV33LZ), **programtervező informatikus BSc** szakos hallgató jelen nyilatkozat aláírásával kijelentem, hogy az *LR automaták* c. szakdolgozat önálló munkám eredménye, saját szellemi termékem, abban a hivatkozások és idézések standard szabályait következetesen alkalmaztam, mások által írt részeket a megfelelő idézés nélkül nem használtam fel. A dolgozat készítése során betartottam a szerzői jogról szóló 1999. évi LXXVI. törvény szabályait, valamint a Nyíregyházi Egyetem által előírt, a szakdolgozat készítésére vonatkozó szabályokat.

Kijelentem továbbá, hogy sem a dolgozat egészét, sem annak bármely részét nem nyújtottam korábban be szakdolgozatként.

Nyíregyháza, 2022. május 1.

Adrián Patrik
hallgató

Interaction with information processing systems is one of the more challenging areas of information technology. Textual methods are one of the oldest, and despite their inherent problems, one of the most popular and accurate methods of communication. Programming languages in particular allow for clear and precise expression of the user's intent, but require them to learn special formalisms (both in terms of syntax and semantics).

One may consider natural languages to be more user-friendly than the rigid syntax of formal languages, but they are often underspecified, mis- and abused, and generally full of ambiguities. Many entries in an English dictionary map to more than one meaning, often not even similar in nature. Any software trying to extract meaning from human languages must have robust methods to cope with such ambiguities.

The issue is further exacerbated in oral communications, where misunderstandings are common even between humans, let alone in machine interpretation. Speech recognition techniques today are sufficient to understand simple, well-pronounced commands or to decrease the typewriting required to produce written versions of spoken text, but the result is rarely free of errors, requires manual editing and most certainly not precise enough for the direct implementation of complex information processing systems.

Our thesis focuses on modern context-free recognition and parsing techniques designed to help ease the processing burden on human-to-machine and machine-to-machine communications. We will overview current parsing techniques for context-free grammars, investigate a surprisingly simple change to reduce stack movement and simplify parse tree generation in LR-style parsers and finally give modified algorithms for LR-style parsers that work directly on extended context-free grammars without first rewriting them in context-free form.

The work is divided into 4 major sections. Section 1 provides an overview of the most important concepts of formal language theory. Section 2 discusses parsing and related concepts in general. In section 3 we will focus on LR parsing, overview the works that have shaped the path to our algorithm and finally introduce our first contribution, the NEGLR parser. Then, in section 4 we will discuss extended context-free grammars and show how NEGLR can be extended to directly support these grammars without rewriting.

Contents

1	Formal languages and automata	1
1.1	Context-free and regular grammars	2
1.2	Finite automata	4
1.3	Language recognition using finite automata	7
1.4	Pushdown automata	10
2	Parsing overview	15
2.1	A brief overview of parsers	16
2.2	Our goals	19
2.3	Tokenization	19
2.4	Nondeterminism	21
2.5	Representation	21
3	Context-free LR parsing	25
3.1	Generalized parsing	27
3.2	Nullable-eliminated GLR	34
3.3	Performing reductions	36
3.4	The NEGLR parser	39
4	Extended context-free parsing	43
4.1	The grammar of an extended production	44
4.2	Extended context-free rules	48
4.3	Representing derivations	53
4.4	Right-nulled parse tables for extended grammars	58
4.5	The ENEGLR parser	60
4.6	Overview	60

5	Further research	63
5.1	Reducing branching in SPPFs	63
5.2	Error recovery	64
5.3	Combining parsers	64
5.4	Conclusion	65

Chapter 1

Formal languages and automata

An *alphabet* Σ is a finite, non-empty set of primitive *symbols*⁽¹⁾, often (but not necessarily) representing letters, digits, punctuation or other glyphs and graphemes in natural languages. A *string* or *word* of length n is a finite sequence $\sigma = \langle \sigma_0, \sigma_1, \dots, \sigma_{n-1} \rangle$ of symbols over an alphabet. The length of a string σ is $|\sigma|$. The set of strings of length n over Σ is Σ^n ($n \in \mathbb{N}$). The single element of the language Σ^0 is the *empty string* $\varepsilon = \langle \rangle$.

The set of all strings over an alphabet Σ (the *natural* or *Kleene closure*) is defined as

$$\Sigma^* = \bigcup_{i=0} \Sigma^i.$$

$\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ is the set of strings of nonzero length (the *positive closure*). The *optional closure* of $\Sigma^?$ is the set $\Sigma_0 \cup \Sigma_1$. A *language* over an alphabet Σ is a (possibly improper) subset of its natural closure.

The concatenation of two strings $\xi = \langle \xi_0, \dots, \xi_{n-1} \rangle$ and $\psi = \langle \psi_0, \dots, \psi_{m-1} \rangle$ is the string $\sigma = \langle \xi_0, \dots, \xi_{n-1}, \psi_0, \dots, \psi_{m-1} \rangle$ and is written as $\xi\psi$. Concatenation is associative but not commutative. A language with the empty string and the string concatenation operation together form a monoid with ε as the identity element.

¹ Symbols of an alphabet are also referred to as characters.

1.1 Context-free and regular grammars

A *context-free grammar* (CFG) is a quadruple $G = \langle V, \Sigma, P, S \rangle$, where V (the set of *nonterminals* or *grammar variables*) and Σ (the set of *terminals* or *alphabet*) are finite, disjoint sets of symbols with $\Gamma = V \cup \Sigma^{(2)(3)}$; $P \subset V \times \Gamma^*$ is the finite set of *productions* and $S \in V$ is the *start symbol*. Any $\alpha \in \Gamma^*$ string of symbols is called a *sentential form*, if $\alpha \in \Sigma^*$, it is also a *sentence*, otherwise it is an *intermediate form*.

As usual, a production $p \in P$ will be written as $X \rightarrow \xi$, where $\xi \in \Gamma^*$ is an arbitrary sentential form. We will use $\text{LHS}(p)$ to refer to X and $\text{RHS}(p)$ to refer to ξ . A *grammar slot* is a pair $\langle p, k \rangle$ with $p \in P$ and $0 \leq k \leq |\text{RHS}(p)|, k \in \mathbb{N}$.

Let $\lambda X \mu$ be a string and $\Rightarrow$ a relation on Γ^* (called a *derivation step*) such that $\lambda X \mu \Rightarrow \lambda \xi \mu$ iff⁽⁴⁾ $(X \rightarrow \xi) \in P$. The sequence of sentential forms $\sigma_0, \sigma_1, \dots, \sigma_n$ is a *derivation path* of length n from α to ω iff $\alpha = \sigma_0 \Rightarrow \sigma_1 \Rightarrow \dots \Rightarrow \sigma_{n-1} \Rightarrow \sigma_n = \omega$. $\alpha \xRightarrow{*} \omega$ iff there is a derivation path of any length from α to ω (reflexive transitive closure); $\alpha \xRightarrow{+} \omega$ iff the path is of positive length (transitive closure). A string ω is *derivable* from the string α iff $\alpha \xRightarrow{*} \omega$ ⁽⁵⁾.

A nonterminal is *nullable* if it derives ε , *strictly nullable* if it can only derive ε and *unproductive* if it cannot derive any sentences at all. A symbol s is *reachable* from A iff A derives a sentential form containing s ($A \xRightarrow{*} \alpha s \beta : \alpha, \beta \in \Gamma^*$).

The **FIRST** set of a nonterminal A is the set of terminals a derivation of A can begin with, i.e. $\{t \in \Sigma : \exists \sigma \in \Sigma^* (A \xRightarrow{*} t\sigma)\}$. As the existential quantification is impossible to evaluate directly over a potentially infinite set, it is usually calculated as, for any $\alpha \in \Gamma^*$:

$$\text{FIRST}(\alpha) = \begin{cases} \emptyset & \text{where } \alpha = \varepsilon \\ \{t\} & \text{where } \alpha = t\beta \wedge t \in \Sigma \\ \text{FIRST}(X) & \text{where } \alpha = X\beta \wedge X \xRightarrow{*} \varepsilon \\ \text{FIRST}(X) \cup \text{FIRST}(\beta) & \text{where } \alpha = X\beta \wedge X \xRightarrow{*} \varepsilon \end{cases}$$

Then $\text{FIRST}(A) = \bigcup_{p \in P : \text{LHS}(p)=A} \text{FIRST}(\text{RHS}(p))$.

² We will use capital Latin letters to denote nonterminals, lower case for terminals and lower case Greek for strings of symbols.

³ In an abuse of notation symbols will also be used in place of their singleton sequences.

⁴ *If and only if*.

⁵ We also say that α *derives* ω .

A nonterminal A is *recursive* if $A \Rightarrow^+ \alpha A \beta$, *left recursive* if $A \Rightarrow A \alpha$, *indirectly left recursive* if $A \Rightarrow^+ A \alpha$, *hidden left recursive*⁽⁶⁾ if $A \Rightarrow^+ \alpha A \beta$ where $\alpha \Rightarrow^+ \epsilon$ ⁽⁷⁾ and *cyclic* if $A \Rightarrow^+ A$; it is *self-embedding* iff $A \Rightarrow^+ \alpha A \beta$ and neither α nor β are strictly nullable. These definitions can be extended to grammars: a grammar is $\sim$ if it has at least one $\sim$ nonterminal.

The *language* generated by a nonterminal A in the grammar G is the set of sentences derivable from the string A ($L_G(A) = \{\alpha \in \Sigma^* : A \Rightarrow^+ \alpha\}$)⁽⁸⁾. The language generated by a grammar is $L_G = L_G(S)$. Two grammars are *equivalent* if they generate the same language.

A grammar is *noncontracting* or *ϵ -free* iff the right-hand side of every rule is at least of length 1, except maybe for a rule $S \rightarrow \epsilon$, in which case the symbol S cannot be present in the right-hand side of any productions. Every CFG has an equivalent noncontracting grammar⁽⁹⁾.

Linear grammars are context-free grammars that have at most a single nonterminal in the right-hand side of any production. If the nonterminal is always at the left (right) end of the production (if present), the grammar is *left-linear* (*right-linear*). Left- and right-linear grammars are collectively known as *regular grammars*. Regular grammars are a proper subset of non-self-embedding grammars, although they both generate the same set of languages, called the *regular languages*.

For example the grammar $S \rightarrow (X), S \rightarrow \epsilon, X \rightarrow)S(, X \rightarrow \epsilon$ is self-embedding, therefore not regular, but the *language* (which consists of zero or more repetitions of the string $()$) is, proven by the regular grammar $S \rightarrow \epsilon, S \rightarrow ()S$.

A grammar without recursive nonterminals is a *nonrecursive grammar*, and can only perform a finite number of derivation steps before “running out of” nonterminals. Nonrecursive grammars generate languages with only a limited number of words: the *finite languages*.

Relaxed formalisms are usually more expressive than stricter ones are. Languages can be categorized by the “lowest” class of grammar that can generate them, which we call the *class* of a language.

The language classes listed in Table 1.1 are totally ordered in the sense that each is a proper subset of the one in the row above⁽¹⁰⁾.

⁶ Hidden recursion implies indirect recursion, but not the other way around.

⁷ Similar definitions apply for right recursion.

⁸ Alternative definitions using set operations on languages rather than derivation are possible.

⁹ One such example is the Chomsky normal form (CNF) described in [2].

¹⁰ The table is not exhaustive and only contains formalisms discussed in this thesis.

Language class	Generator grammar	Recognizer
context-free	context-free	pushdown a.
linear	linear	1-turn pushdown a.
regular	non-self-embedding	finite a.
finite	nonrecursive	acyclic finite a.

Table 1.1: Language classes, their generator grammars and recognizers.

Context-free grammars are widely used in practice for the syntax description of many programming languages, including those of C++⁽¹¹⁾ ([58]) and Java ([56]).

1.2 Finite automata

Finite automata (FA) or *state machines* (SM) are mathematical models of simple algorithms whose actions are dependent only on the next input symbol and some finite memory. Formally, a finite automaton is a quintuple $\langle Q, \Sigma, \delta, \mathcal{S}, \mathcal{F} \rangle$, where

- Q is the set of the automaton's *states*,
- Σ is the *input alphabet*,
- $\delta : (Q \times \Sigma^?) \rightarrow Q$ is the *transition relation*,
- $\mathcal{S} \subseteq Q$ is the set of *starting states*,
- $\mathcal{F} \subseteq Q$ is the set of *accepting states*.

Automata are generally represented as a graph $\langle Q, \delta \rangle$ with the edges labeled using elements of $\Sigma^?$: the graph has an edge $p \xrightarrow{t} q$ labeled t iff $\delta(p, t) \ni q$. Sometimes we ignore labels and say $p \rightarrow q$ indicating that there is an edge of *some* label from p to q . A transition $p \rightarrow q$ is an *egress (ingress)* transition for p (q). A *path* from a state q_0 to q_n is a sequence of zero or more states $q_0, q_1, \dots, q_n$ such that $q_0 \rightarrow q_1 \rightarrow \dots \rightarrow q_n$. The existence of a path from p to q is written as $p \rightsquigarrow q$ (the transitive extension of $\rightarrow$). A *path through the automaton* is a path $s \rightsquigarrow f$ from some state $s \in \mathcal{S}$ to $f \in \mathcal{F}$.

On the path $q_0 \xrightarrow{t_0} q_1 \xrightarrow{t_1} \dots \xrightarrow{t_{n-1}} q_n$ the automaton *reads* the string $t_0 t_1 \dots t_{n-1}$. $q_0 \xrightarrow{\sigma} q_n$ iff there is a path from q_0 to q_n on which the automaton reads σ . A string $\sigma \in \Sigma^*$ is *accepted* iff there is a path through the automaton reading exactly σ . The language $L(M)$ of an automaton M is the set of strings M accepts.

¹¹ The lexical grammar of C++ is not context-free because the definition of *raw-string* includes the possibility of an arbitrary delimiter. Even though 5.13.5 (2) of the standard limits the maximum length of the delimiting *d-char-sequence* to 16 characters, this restriction is not reflected in the formal grammar.

It is precisely the regular languages that can be recognized by finite automata.

If the relation δ is functional then the automaton is deterministic (DFA), otherwise it is nondeterministic (NFA). δ is often a partial function. If the automaton is in a state q with the next symbol being t and $\delta(q, t)$ is undefined, the automaton halts and rejects its input. δ^* is the extension of δ for strings:

$$\delta^*(p, \sigma) \ni q \iff p \overset{\sigma}{\rightsquigarrow} q.$$

It is customary to allow automata with edges labeled by ε , which represent transitions the automaton can take without reading any input. A state machine with ε -transitions can always be transformed into an equivalent one without them.

A state q of an automaton is *unreachable* iff $\nexists s \in \mathcal{S} : s \rightsquigarrow q$ and a *trap* iff $\nexists f \in \mathcal{F} : q \rightsquigarrow f$. Discarding unreachable and trap states, along with their ingress/egress transitions does not change the language accepted. Automata without unreachable or trap states are *trimmed*.

1.2.1 Determinization of automata

Sometimes it is necessary to simulate the possible executions of an NFA in a deterministic way. The de facto standard method, the Rabin-Scott algorithm [3] (also known as the powerset construction) is briefly described here for reference.

Let $\Pi = 2^Q$ be the set of all possible sets of states of an NFA M . Members of Π are called *configurations*, and represent all possible states the automaton can be in at any given time. One can always construct a DFA from an NFA using elements of Π as its states.

The ε -closure of a state q of an NFA is the reflexive transitive closure of $\overset{\varepsilon}{\rightarrow}$: the set of all states reachable from q via ε -transitions only ($\text{CLOSURE}[q] = \{r \in Q : q \overset{\varepsilon}{\rightsquigarrow} r\}$). The ε -closure of a configuration π of an NFA is the union of the ε -closures of its members:

$$\text{CLOSURE}[\pi] = \bigcup_{q \in \pi} \text{CLOSURE}[q].$$

Members of π form the *kernel* of the closure. Any ε -closure of an NFA is uniquely identified by its kernel.

The transition function δ' of the DFA:

$$\delta'(\pi, t) = \bigcup_{q \in \pi} \delta^*(q, t).$$

The start state is $\mathcal{S}' = \text{CLOSURE}[\mathcal{S}]$, the accepting ones are those that contain at least one of the accepting states of the original NFA: $\mathcal{F}' = \{\pi \in \Pi : \pi \cap \mathcal{F} \neq \emptyset\}$.

The automaton $D(M) = \langle \Pi, \Sigma, \delta', \mathcal{S}', \mathcal{F}' \rangle$ is deterministic, furthermore, $L(D(M)) = L(M)$.

1.2.2 Equivalence of states

Two states $p, q \in Q$ are *equivalent* iff $\forall \sigma \in \Sigma^* : (\delta^*(p, \sigma) \cap \mathcal{F} \neq \emptyset \iff \delta^*(q, \sigma) \cap \mathcal{F} \neq \emptyset)$, that is, a string is accepted starting from p iff it is also accepted when starting from q .

Testing states for equivalence often appears in the minimization of automata, but is only efficient for deterministic machines ([42]). One can use the Rabin-Scott construction on any NFA M to acquire the DFA $D(M)$ and test for the equivalence of the configurations $\{p\}$ and $\{q\}$ using, for example, Hopcroft's partitioning algorithm ([15]).

1.2.3 Direction-aware automata

Automata can be generalized using a *reading direction*. Common automata found in literature that read their input from the left to the right will be referred to as LTR (left-to-right) automata, while those that read from the right end moving towards the left will be denoted RTL (right-to-left).

Formally, a *direction-aware finite automaton* is a sextuple $\langle Q, \Sigma, \delta, \mathcal{S}, \mathcal{F}, \Delta \rangle$ where $Q, \Sigma, \delta, \mathcal{S}$ and $\mathcal{F}$ retain their usual meaning; $\Delta \in \{\rightarrow, \leftarrow\}$ is the *reading direction*. If Δ is $\rightarrow$ then the automaton is a *left-to-right automaton* and behaves like a standard FA does.

If Δ is $\leftarrow$, then by traversing edges labeled $t_0, t_1, \dots, t_{n-1}$ the automaton *reads* the string $t_{n-1} \dots t_1 t_0$. Such a *right-to-left automaton* accepts a string $\sigma \in \Sigma^*$ iff there is a path through it that reads σ .

The *reverse* $\text{REVERSE}(M)$ of an automaton $M = \langle Q, \Sigma, \delta, \mathcal{S}, \mathcal{F}, \Delta \rangle$ is the automaton $\langle Q, \Sigma, \delta', \mathcal{F}, \mathcal{S}, \Delta \rangle$ where $(q \xrightarrow{c} p) \in \delta'$ iff $(p \xrightarrow{c} q) \in \delta$.

The *mirror* of an automaton M is defined to be the same as $\text{REVERSE}(M)$ but with the opposite reading direction⁽¹²⁾. The language accepted by the mirror of an automaton M is the *same* as that of M ($L(\text{MIRROR}(M)) = L(M)$).

Most of the time the directionality of an automaton will either be clear from the context or unimportant to the discussion. When the directionality of an automaton has significance and is not obvious from context we will denote grammar slots by placing arrows over them ($\overrightarrow{\bullet\alpha\beta}$ or $\overleftarrow{ba\bullet cc}$).

1.3 Language recognition using finite automata

In this section we will show how to construct an automaton for a regular grammar. These automata will only have a single start and accepting state.

1.3.1 Trivial automata

We consider the following automata to be *trivial*:

- The *empty automaton* only accepts the empty string and is defined as the tuple $\langle\{q\}, \Sigma, \emptyset, \{q\}, \{q\}\rangle$.
- The *character recognizer* accepts a single character $c \in \Sigma$ and nothing else:
 $\langle\{q_S, q_F\}, \Sigma, \{\langle\langle q_S, c \rangle, q_F \rangle\}, q_S, q_F\rangle$.

1.3.2 Operations on finite automata

Finite automata can be combined using simple methods representing common operations on languages. If automaton A accepts some language L_A and B accepts L_B then automata which accept L_AL_B , $L_A \mid L_B$, L_A^+ and $L_A^?$ are seen in **Figure 1.1**⁽¹³⁾.

¹² It would probably be more precise to call the mirror automaton reverse and the reverse automaton mirror, but we are stuck with existing terminology.

¹³ Grammars given using Chomsky's formalism have no need for repetition.

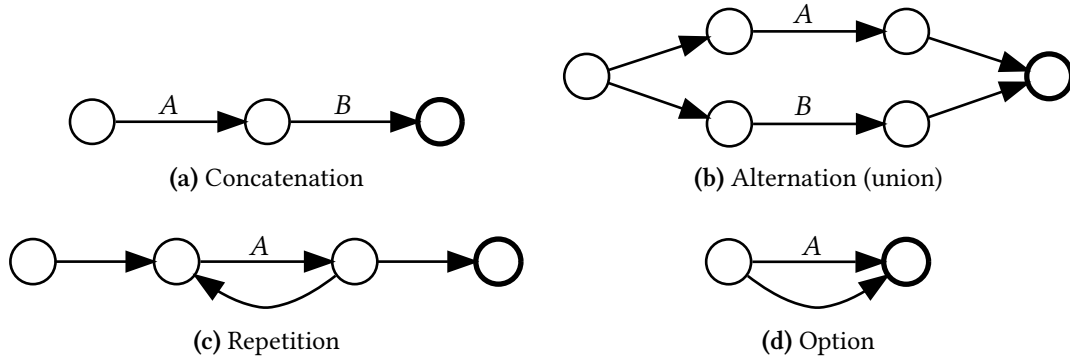


Figure 1.1: Construction of automata for some common operations.

1.3.3 Simple string recognizers

Automata that recognize a single, fixed string of characters can be built by concatenating character recognizers. We will refer to such automata as a *simple string recognizer* (SSR). The states of such automata will be labeled using grammar slots with the dot before the character that is next to be read. For an SSR that accepts the string $\alpha\beta$ the starting state is $\cdot\alpha\beta$, the accepting state is $\alpha\beta\cdot$ and the state reached after reading α is $\alpha\cdot\beta$.

The egress edge of a state is labeled by the symbol after the dot (states where the dot is at the end do not have egress edges, as they mark fully read input and are accepting states). For example, a simple automaton that recognizes the keyword `while` can be seen in figure Figure 1.2.

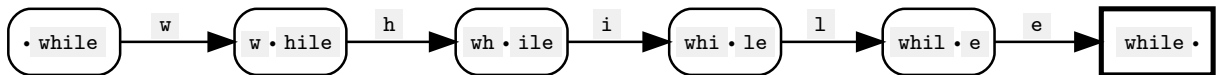


Figure 1.2: An SSR for the `while` keyword.

1.3.4 Recognition of nonterminals

Right-hand sides of productions in a grammar often contain nonterminal symbols. As nonterminals never appear directly in the input, they cannot be recognized directly. One method of avoiding them is by *embedding* (substituting) an automaton for a nonterminal-labeled edge of another. Figure 1.3 shows an automaton for the grammar with only two productions $S \rightarrow aBc$ and $B \rightarrow bb$.

This naive method does not terminate and generates infinite automata⁽¹⁴⁾ for recursive nonterminals.

¹⁴ If *finiteness* was not a requirement, automata generated this way would be correct recognizers for all context-free languages.

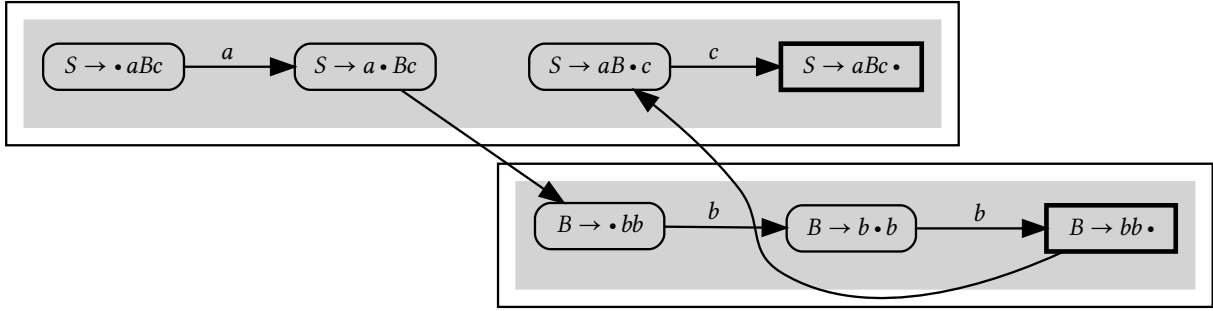
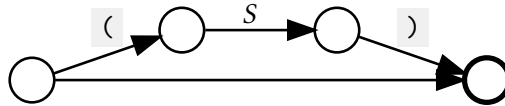
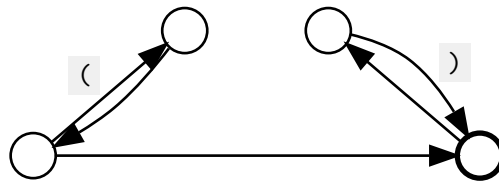


Figure 1.3: Automaton embedding.

Self-embedding automata

Let us consider an automaton for some recursive nonterminal A with the start state s and end state f that has an edge $p \xrightarrow{A} q$. The recursion can be broken by adding two edges to the automaton: one $p \rightarrow s$ and another $f \rightarrow q$. If the grammar is non-self-embedding, the edge with the nonterminal must either be (ignoring ε -transitions) the first or the last edge the automaton traverses⁽¹⁵⁾, which makes one of the two edges added an ε -loop that can be safely removed as it does not influence the language accepted. This method generates correct FA for all non-self-embedding grammars.

Let us try to apply the previous method to the language of correctly nested parentheses⁽¹⁶⁾: $()$, $(())$, $((()))$, etc., generated by the grammar $S \rightarrow \varepsilon, S \rightarrow (S)$. The recursive automaton for this grammar is shown in Figure 1.4, the recursion-eliminated version in Figure 1.5.


 Figure 1.4: Recursive FSM for the grammar $S \rightarrow \varepsilon, S \rightarrow (S)$.

 Figure 1.5: Resolved FSM for the grammar $S \rightarrow \varepsilon, S \rightarrow (S)$.

¹⁵ If it was not, then there would be a path $\alpha A \beta$ through the automaton where $\alpha, \beta \neq \varepsilon$, which would mean the grammar is self-embedding.

¹⁶ Also known as the $a^n b^n$ language.

This new automaton, however, is not correct: it accepts any number of `(` symbols followed by any number of `)` s. This is clearly not the language of correctly nested pairs of parentheses⁽¹⁷⁾.

Let us examine why the automaton fails with this language. Before recursing on `S` a `(` must be read and a `)` after. Having no memory, the automaton cannot “remember” how many times it has entered the recursion, and, as a result, cannot know how many “exit actions” it must perform. (This is why, in general, self-embedding grammars cannot be recognized, as for any self-embedding derivation $A \xRightarrow{+} \alpha A \beta$ the number of times α and β are read must match.) Head and tail recursion do not exhibit this deficiency as at least either the prefix or the suffix is empty and thus makes no observable difference.

1.4 Pushdown automata

When augmented by memory organized as a stack, finite state machines can recognize self-embedding grammars (up to context-free). In one such construction the automaton pushes a marker on the stack before going into recursion, signaling where it was “interrupted”. When the recursion finishes, the marker is popped from the stack and is used to determine which rule to return to. This is known as a *pushdown automaton* and serves as a recognizer for all context-free grammars.

A pushdown automaton is a sextuple $\langle Q, \Sigma, \Theta, \delta, \mathcal{S}, \mathcal{F} \rangle$, where

- Q is the set of the automaton’s *states*,
- Σ is the *input alphabet*,
- Θ is the *stack alphabet*,
- $\delta \subseteq (Q \times \Sigma^? \times \Theta^*) \times (Q \times \Theta^*)$ is the *transition function*,
- $\mathcal{S} \subseteq Q$ is the set of *starting states*,
- $\mathcal{F} \subseteq Q$ is the set of *accepting states*.

Pushdown automata are finite automata augmented with a stack: in every transition the automaton can pop and/or push symbols on the top of the stack. The transition taken can be dependent on the symbols popped.

In the beginning and after a successful execution the stack is empty. If δ is functional then the PDA is deterministic; otherwise it’s nondeterministic. The sets Q, Σ, Θ and δ are all finite.

Figure 1.6 shows the automaton for the previous example.

¹⁷ The failure of this construction does not prove that the language cannot be accepted by *any* FSM. The pumping lemma for regular languages can be used for a full proof of the statement.

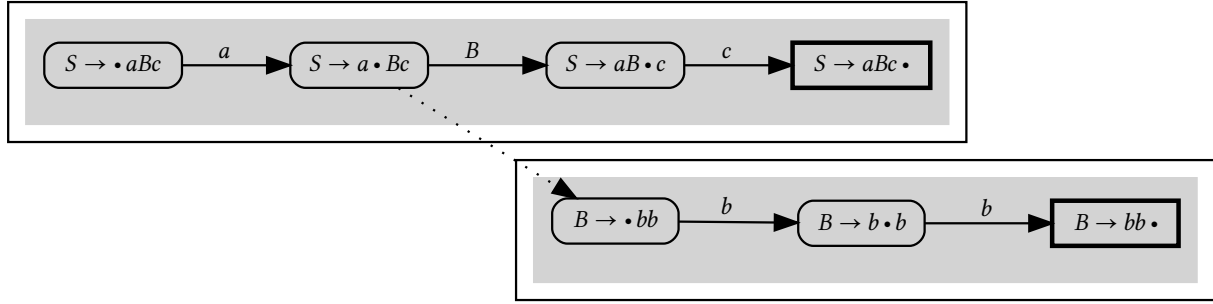


Figure 1.6: Sublanguage recognition using a pushdown automaton.

1.4.1 Context-free recognition

The following algorithm demonstrates how a pushdown automaton might recognize some text using a context-free grammar⁽¹⁸⁾ (**Algorithm 1**). Terminal symbols are read directly from the input stream; for nonterminals the successive grammar slot is pushed on the stack⁽¹⁹⁾ and the automaton nondeterministically transitions to all states that can begin a production for the nonterminal. When the end of a rule is reached the automaton looks at the symbol on the top of the stack and goes to the state with the same label.

$\delta()$

- 1: **if** the label of the current state is of the form $A \rightarrow \alpha \cdot$ **then**
- 2: **if** the stack is empty **then**
- 3: **accept**
- 4: **else**
- 5: **pop** a state s from the stack
- 6: **goto** state s
- 7: **else if** the label of the current state is of the form $A \rightarrow \alpha \cdot t \beta$ where $t \in \Sigma$ **then**
- 8: **if** the next symbol of the input is t **then**
- 9: **consume** t from the input
- 10: **goto** state $A \rightarrow \alpha t \cdot \beta$
- 11: **else if** the label of the current state is of the form $A \rightarrow \alpha \cdot X \beta$ where $X \in V$ **then**
- 12: **push** the state $A \rightarrow \alpha X \cdot \beta$ on the stack
- 13: **goto** states $X \rightarrow \cdot \xi$
- 14: **else**
- 15: **reject**

Algorithm 1: The transition relation of a context-free pushdown automaton.

¹⁸ Different constructions are also possible.

¹⁹ Strictly speaking, a state cannot be pushed on the stack. This notational convenience can be remedied by assigning a unique stack symbol to each state.

The transition algorithm is nondeterministic as the line “**goto** states $X \rightarrow \bullet \xi$ ” specifies a set of states without any instructions on which one to go to. (There can be no such specification as all possible derivation paths have to be explored.)

A more complex example

Consider the following grammar: $S \rightarrow aABbC$, $A \rightarrow bA \mid c \mid Aaa \mid B$, $B \rightarrow \varepsilon$, $C \rightarrow c$. The complete NFA for the grammar can be seen in **Figure 1.7**. The automata for recognition of individual productions are shown on a gray background. Automata for the recognition of a single nonterminal have a solid black border. Embedding transitions are shown with dotted arrows.

As an example, we give the transition relation δ for this grammar in **Table 1.2** in a more “traditional” form (as a table) rather than as an imperative procedure as in **Algorithm 1**.

The starting state of this PDA is the one labeled as $S \rightarrow \bullet aABbc$, the accepting one is $S \rightarrow aABbc \bullet$ with the stack being empty.

State	Read	Stack operation	Goto state
$S \rightarrow \bullet aABbC$	a	–	$S \rightarrow a \bullet ABbC$
$S \rightarrow a \bullet ABbC$	ε	push $S \rightarrow aA \bullet BbC$	$A \rightarrow \bullet bA$
$S \rightarrow a \bullet ABbC$	ε	push $S \rightarrow aA \bullet BbC$	$A \rightarrow \bullet c$
$S \rightarrow a \bullet ABbC$	ε	push $S \rightarrow aA \bullet BbC$	$A \rightarrow \bullet B$
$S \rightarrow a \bullet ABbC$	ε	push $S \rightarrow aA \bullet BbC$	$A \rightarrow \bullet Aaa$
$S \rightarrow aA \bullet BbC$	ε	push $S \rightarrow aAB \bullet bC$	$B \rightarrow \bullet$
$S \rightarrow aAB \bullet bC$	b	–	$S \rightarrow aABb \bullet C$
$S \rightarrow aABb \bullet C$	c	–	$S \rightarrow aABbC \bullet$
$S \rightarrow aABbC \bullet$	ε	pop X	X
$A \rightarrow \bullet bA$	b	–	$A \rightarrow b \bullet A$
$A \rightarrow b \bullet A$	ε	push $A \rightarrow bA \bullet$	$A \rightarrow \bullet bA$
$A \rightarrow b \bullet A$	ε	push $A \rightarrow bA \bullet$	$A \rightarrow \bullet c$
$A \rightarrow b \bullet A$	ε	push $A \rightarrow bA \bullet$	$A \rightarrow \bullet B$
$A \rightarrow b \bullet A$	ε	push $A \rightarrow bA \bullet$	$A \rightarrow \bullet Aaa$
$A \rightarrow bA \bullet$	ε	pop X	X
$A \rightarrow \bullet c$	c	–	$A \rightarrow c \bullet$
$A \rightarrow c \bullet$	ε	pop X	X
$A \rightarrow \bullet B$	ε	push $A \rightarrow B \bullet$	$B \rightarrow \bullet$
$A \rightarrow B \bullet$	ε	pop X	X
$A \rightarrow \bullet Aaa$	ε	push $A \rightarrow A \bullet aa$	$A \rightarrow \bullet bA$
$A \rightarrow \bullet Aaa$	ε	push $A \rightarrow A \bullet aa$	$A \rightarrow \bullet c$
$A \rightarrow \bullet Aaa$	ε	push $A \rightarrow A \bullet aa$	$A \rightarrow \bullet B$
$A \rightarrow \bullet Aaa$	ε	push $A \rightarrow A \bullet aa$	$A \rightarrow \bullet Aaa$
$A \rightarrow A \bullet aa$	a	–	$A \rightarrow Aa \bullet a$
$A \rightarrow Aa \bullet a$	a	–	$A \rightarrow Aaa \bullet$
$A \rightarrow Aaa \bullet$	ε	pop X	X
$B \rightarrow \bullet$	ε	pop X	X
$C \rightarrow \bullet c$	c	–	$C \rightarrow c \bullet$
$C \rightarrow c \bullet$	ε	pop X	X

Table 1.2: The transition relation for the grammar in 1.4.1.

Chapter 2

Parsing overview

Probably the most common practical problem about formal languages is parsing, where, given a grammar G with a start symbol $S \in V$ and some input $\sigma \in \Sigma^*$, an algorithm is to produce one or all derivation paths from S to σ . A parser is an algorithm that solves the parsing problem, usually outputting the derivation path in the form of a tree, or, in the general case, a (possibly infinite) collection of trees.

The decision version of this problem is called recognition and only asks whether $S \xRightarrow{+} \sigma$ or not.

Formally, a *recognizer* is a function $\text{RECOGNIZE}_G : \Sigma^* \rightarrow \{\top, \perp\}$ for a grammar G which, given the input σ , determines whether $S \xRightarrow{+} \sigma$. A *parser* can be defined as the multifunction $\text{PARSE}_G : \Sigma^* \twoheadrightarrow (\Gamma^*)^+$, where the parser's output is a derivation path of nonzero length. The parser is *deterministic* iff PARSE_G is a single-valued function.

Top-down vs. bottom-up parsing

Top-down parsers attempt to *generate* the input string from the start symbol by *applying* productions and pruning the paths that cannot possibly lead to a valid derivation, while bottom-up parsers *interpret* the input, finding where a production might have been applied and *reducing* the symbols into their corresponding nonterminals. Unger's method and the CYK algorithm clearly demonstrate these approaches.

Many parsers, like left corner, Earley and even the LR family of algorithms, are not easy to categorize as either and can be seen as a mix of the two approaches. Even though LR is often cited as the textbook

example of bottom-up parsing, through the inherent top-down construction of the DFA, it is not a “pure” bottom-up parser.⁽¹⁾

The top-down approach is lucrative because as the language generation process is performed, valuable contextual information becomes available. A certain nonterminal might only appear as part of another rule, surrounded by some specific symbols, and the top-down method seems to enjoy a significant advantage over bottom-up approaches in this regard. A pure bottom-up approach is “blind”: it cannot perform context-based pruning because it does not have access to the possible generative patterns of the grammar.

Language *generation* has a curious duality with language *recognition*. Let’s take the set of productions P of a grammar and swap their left and right sides: a rule of the form $A \rightarrow \alpha$ becomes $\alpha \rightarrow A$. We can also say that a top-down parser is trying to generate the input string from the start symbol using the original grammar, while a bottom-up parser is trying to generate the start symbol from the input string, using the swapped grammar described above.

Online vs. offline parsing

An online (incremental) algorithm sequentially reads the input in one direction (typically from left to right) and performs as much work as possible before moving on to the next token. Having read all input up to its current position, it has complete (left) contextual information. This sequential dependency, however, prevents efficient parallelization. Online parsers include the LL, Earley and LR families.

An offline (parallel) algorithm works separately on non-overlapping parts of the input, without any contextual dependence. This leads to efficient parallelization, as the workers are independent from each other, but also precludes context-based pruning. Unger’s method and the CYK algorithm are usually implemented as parallel parsers.⁽²⁾

2.1 A brief overview of parsers

Context-free grammars were introduced by Chomsky ([2], [1]). Research about context-free parsing for analysis of both natural and formal languages, has been active ever since.

¹ A primitive LR-like parser, which has no supporting DFA, but simply scans the top symbols of the stack to see if they match any rule would be a pure bottom-up stack-based parser, but is not used in practice.

² It is technically possible to implement both as incremental algorithms, but there is almost no reason to do so.

Knuth's [5] laid down the foundations for $LR(k)$ parsers, deterministic simulations of PDAs with k tokens of lookahead. DeRemer's PhD thesis ([9]) introduced $LALR(k)$ automata, reducing the number of states in an $LR(1)$ automaton to that of $LR(0)$ while preserving most of its recognition power.

Lang ([16]) investigated the determinization of general pushdown automata but the paper did not include any concrete algorithms.

Tomita ([24]) extended Knuth's deterministic LR parsers to support nondeterminism. His algorithm is correct for ϵ -free grammars but fails on some ϵ -grammars (as demonstrated in [29]). Farshi also gave a correction of Tomita's GLR by rescanning already visited nodes in the state graph. Scott and Johnstone later showed ([45]) that the problem admits a simpler solution by early reduction of right-nullable rules. They also created a worst-case cubic ([46]) version of their algorithm.

Aycock et. al. ([32], [36]) explored substitution of automata in order to reduce stack movement in GLR parsers. Their solution failed to terminate for hidden left recursion; a fully general algorithm (RIGLR) was published later by Scott and Johnstone ([43]).

Visser et. al. researched scannerless GLR parsers ([35], [34], [38]) that use character-level descriptions of the grammar and do not require a separate lexing phase.

Earley ([13]) gave a simple algorithm that simulates a primitive nondeterministic recognizer for any context-free grammar in worst-case cubic time. His paper also provides a method for constructing parse trees, but that turned out to be incorrect.

Leo's elimination of "chain reductions" in Earley's algorithm [28] results in linear-time parsing on all $LR(k)$ grammars but complicates the retrieval of parse trees. McLean and Horspool [32] realized that, because Earley's algorithm essentially simulates the same NFA as LR does, the same Earley items which would normally be grouped into an LR DFA state always appear together, therefore pre-computing them significantly speeds the algorithm up. Aycock and Horspool [37] applied what is essentially the ϵ -GLR technique of [33] to Earley parsers.

Stearns and Lewis ([7]) introduced $LL(k)$ grammars, which produce top-down parsers that decide on which production to apply based on at most a fixed number of tokens of lookahead.

Parr's LL^* ([51]), implemented in his popular ANTLR parser generator, extends the LL technique to a dynamic number of lookaheads. The tool is one of the most popular, publicly available deterministic parser generators that handles direct left recursion via internal grammar rewriting but does not accept grammars with indirect left recursion.

Parsing expression grammars and the associated Packrat parser of Ford ([39]) treat productions as *ordered*: alternatives are tried in a specific order and if a rule matches, a later one is never tried, even if the greedy choice ultimately leads to a failed parse. (A similar strategy is employed by ANTLR, but only for ambiguity resolution.) The original packrat parsers are still unable to deal with left recursion, but Warth et. al. ([47]) showed that it is possible to support left recursion by iterative deepening techniques.

Grune ([49]) describes a Generalized LL parser based on the earlier work of Deudekom and Kooiman ([30]). This can, in a sense, be seen as a mirror of the GLR algorithm, where the prediction phase pushes symbols on the stack and symbols read from the input are popped.

Scott and Johnstone gave another generalization of the LL technique first only in the form of a recognizer ([50]), later as a parser ([53]); in [54] they further extended their algorithm to process grammars described in EBNF.

Left-corner (LC) parsers build the first symbol of a rule in a bottom-up fashion, then use top-down predictions for the rest. According to [14], left corner parsing was used as early as in [4]. Nederhof ([31]) generalized left-corner parsing to all context-free grammars.

Cocke, Younger and Kasami ([6], [10]) parallelized the recognition using a dynamic programming approach. Their algorithm as widely known is only applicable to grammars in Chomsky normal form, but it's straightforward to generalize it to nonbinarized grammars at the cost of significantly increased runtime complexity. Unger's method [8] can be seen as the top-down version of the CYK algorithm.

Valiant ([17]) has shown how boolean matrix multiplication can be used to do build a CYK-like tabular recognizer in less than cubic time; Lee ([40]) proved that significantly faster solutions to recognition can be reduced to boolean matrix multiplication. It is unknown whether general matrix multiplication is possible in worst-case $O(n^2)$, but no *practical* improvements since that of Strassen [12] have been found.

Herman ([57]) claims that a new technique “expressed as a series of standard operations on regular languages and relations” and “context-free memoization” achieves generalized parsing speeds significantly faster than other algorithms do.

2.2 Our goals

Most publicly available parsers have some limitations regarding the grammars they're able to accept. Most LL-based parsers reject grammars containing left recursion, while LALR-based alternatives sometimes observe “mysterious” reduce/reduce conflicts in merged states.

Our goal is to develop an algorithm that is free of these limitations and is as general as a context-free parser can reasonably be. In particular, we are in pursuit of the following:

- **Generalized:** Parses according to *any* context-free grammar. No grammar transformations (rewriting to Chomsky normal form, left-recursion elimination etc.) are required or performed.
- **Nondeterministic:** Parses *ambiguous* texts. In practice few but the most simple grammars and languages are without any ambiguity. We require that the algorithm is able to deal with the most vexing parses, including those of pathological grammars.
- **Online:** In literature input is considered to be a randomly accessible string of n symbols. Practice, however, is often different, and the input might not be fully available at the start of a parse. Especially in the case of automatic completion in IDEs, a parser must be able to supply partial results as the user is typing their input.
- **Quasi-scannerless:** Tokenization is performed in a context-sensitive manner as part of the parsing process, not as an independent preliminary step.

2.3 Tokenization

It is clear from examples of parsers for common programming languages that the two-tier architecture, in which the input is first independently processed (tokenized) by a context-insensitive lexer, is insufficient for many real-world applications. Keywords, for example, are generally recognized as such everywhere in the program text, regardless of whether they are contextually valid; operators being mistaken for each other⁽³⁾ is the direct result of an independent lexing phase that does not allow ambiguities, as it (having no access to contextual information) is unable to tell whether a particular token

³ C++ compilers, for a long time, were disallowed to correctly parse the string `vector<vector<int>>>`, mistaking the two closing angle brackets for a right-shift operator. Even the most recent C(++) compilers reject the expression `5 -- 3`, which suffers from a similar problem.

makes any sense in the context where it is found⁽⁴⁾. C# has multiple tokens (`get` / `set` being one example) that are keywords in one context and regular identifiers in others. The symbol `_` has a dual function as either an identifier or the discard symbol.

Completely abandoning the tokenizer is, however, impractical. For example, decoding Unicode text might require the recognition and assembly of text elements from their constituents. While it is technically possible to include the lowest level details in the grammar and perform this task using a full parser, the indiscriminate application of complicated algorithms significantly degrades performance. Tokenization and the recognition of trivial terms is generally better if delegated to a simpler but faster tool.

This dynamic approach to tokenization makes even language embedding almost trivial. Whenever an insert in another language is expected, the parser can simply ask the tokenizer to match a special terminal representing embedded language text. The tokenizer is not limited in the operations performed: it can even invoke another, unrelated program and return the parse tree under an opaque single token.⁽⁵⁾

By not restricting the tokenizer to unambiguous matches, it is no longer required to make decisions it cannot reliably make. For example, in `vector<vector<int>>`, when asked for the kind of symbol right after the letter `t` of `int`, the tokenizer can respond with a set containing a right shift operator spanning two characters, a greater-than operator spanning a single character and a closing angle bracket of a template argument list, also spanning a single character. It is left to the parser to decide which, if any, of these tokenizations lead to a valid parse. This example also illustrates that the tokens matched by the tokenizer can not only differ in their type, but also in their length.

We do not ask questions like “What is the kind of the token starting at position 284?” but “Can a plus sign or an identifier be recognized at position 284?” instead. By providing the expected terminals at any given position, the tokenization phase gains context-sensitivity and only attempts to match terminals viable at the current state of the parsing process⁽⁶⁾.

Formally, the parser’s input is defined a set of tokens with discrete, marked positions labeled using the natural numbers⁽⁷⁾. (The positions would generally represent byte or character boundaries.) A *token* is a triple $\langle t, i, j \rangle$ representing some part of the input stream with left and right extents i and j ($i, j \in \mathbb{N}, i \leq j$) and a terminal $t \in \Sigma$.

⁴ Parsers for C, for example, often employ feedback from the parser into the tokenization pass to deal with context sensitivity (aka. the *lexer hack*).

⁵ We have done this in our implementation for the C family of languages, as they have different whitespacing rules for the lexical and phrase structure grammars.

⁶ If the tokenizer is implemented to make up random but viable tokens, the parser turns into a language *generator*.

⁷ Any other totally ordered set would do.

Parsing in the presence of lexical ambiguities is briefly mentioned in [24], further investigated in [44] and implemented for Earley’s algorithm in Jeffrey Kegler’s MARPA ([55]).

2.4 Nondeterminism

The majority of parser toolkits are deterministic and either report ambiguities as failures or fall back to an arbitrary derivation. As famously demonstrated by the C programming language, nondeterminism is common in practice: the string `A * B`, depending on the nature of `A`, is either a pointer declaration or a multiplicative expression. In C++ the statement `A a();` in the body of a function does not declare a default-constructed local variable, as reasonably expected, but a function called `a` taking no parameters and returning an `A`⁽⁸⁾. The *dangling else* problem is an ambiguity with no proper syntactic resolution.

2.5 Representation

We will first discuss the structure of parse trees for unambiguous derivations (when a string can be derived in exactly one way). A parse tree is a directed graph with two kinds of nodes:

- Nonterminal nodes are labeled with a triple $\langle X, i, j \rangle$ with the subgraph rooted at this node containing a derivation of X with extents i and j .
- Terminal nodes are labeled with a triple $\langle t, i, j \rangle$ and the node stands for the terminal t matched with extents i and j .

Nonterminal nodes have an ordered set of children, one for each symbol in the right-hand side of the production used in the derivation of the nonterminal. Terminal nodes are leaves and do not have any children.

2.5.1 Nulled nodes

A nonterminal is nulled if it was not matched against any input but only “made up” during the parsing process. Using the labeling scheme described above, the label of a nulled nonterminal node would be

⁸ Also known as “the most vexing parse”.

of the form $\langle X, i, i \rangle$. Because derivations of nulled symbols are dependent only on the grammar, not the input, hence the same for any position, it is wasteful to generate multiple copies of them.

Nulled nodes will be labeled without any extents as those are possible to infer from the extents of their parent and siblings.

2.5.2 Parse forests for generalized parsers

Generalized parsers can deal with ambiguous grammars and languages, producing parse trees for *every* possible derivation path of a nonterminal. Producing individual parse trees for each derivation path would be unfeasible, as the number of trees could be exponentially or even infinitely high.

To keep the size of the representation within reasonable bounds, common parts of parse trees will be shared in a data structure known as a shared-packed parse forest (SPPF, [24]). An SPPF is a generalization of the derivation tree, where each node is allowed to have multiple sets of children to represent ambiguous parses. Cycles are allowed for infinitely ambiguous derivations.

Nodes contain their children indirectly via a packed node, so that children belonging to different derivations can be separated. These packed nodes will not be visually represented when there's no ambiguity.

Figure 2.1 shows an example parse forest for the grammar $S \rightarrow abcd \mid abce$ and an input that is ambiguous between $abcd$ and $abce$.

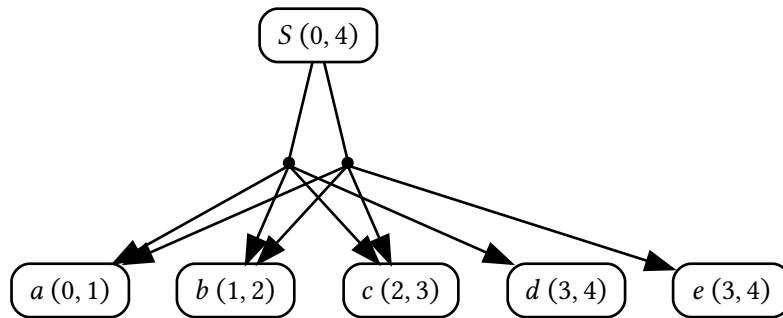


Figure 2.1: SPPF for the parse in 2.5.2

2.5.3 Binarized parse forests

Even though SPPFs prevent derivation trees from reaching exponential sizes, they can still be of unbounded polynomial order ($O(\max(|\text{RHS}(p)|) + 1 : p \in P)$). In [46] binarization is used to curb this

growth: splitting right-hand sides longer than 2 symbols into pieces of length 2. No grammar rewriting is done, all processing happens during runtime.

Intermediate states resulting from the binarization of the grammar are represented by an *intermediate node*.

Some parsers (GLL, Earley) construct their parse trees as they work through the input while LR-style parsers create them during the tracing of a recognized rule in the GSS. Parsers which construct trees in left-to-right order need to create nodes for each recognized prefix of a rule without having seen the end of the match. On the other hand, LR parsers trace the reduction paths from the right, when the matching of a rule is already complete, creating the right end of a match first, without any information on where it began. Forcing an algorithm to produce a different kind of tree is highly impractical.⁽⁹⁾

Partial subtree sharing make parser actions and predicates significantly harder to implement than with non-binarized representations, as changes might have to be introduced to one derivation without also affecting others.

LL-style nodes

These nodes are labeled with grammar slots $\langle p, k \rangle$, where $1 < k < |\text{RHS}(p)|$, and contain derivations for the first k symbols in the right-hand side of p . Nodes for two different productions can be shared as long as their first k symbols match.

An example of an LL-style binarized parse tree is shown in **Figure 2.2**. We have only included the part of the label that is relevant for the identification of the subtree.

LR-style (prefix sharing) nodes

In this case the grammar slot on the node does not label what is under the node, but rather the common prefix of the derivations represented by the node. Any derivation under a node is a possible suffix of a derivation for the prefix on the node's label. Nodes for two different productions can be shared as long as their first k symbols and their left-hand sides (left-contexts) match. An example of an LR-style binarized parse tree is shown in **Figure 2.3**. We have only included the left context part of the grammar slots on the label. These nodes are always labeled with RTL slots.

⁹ Unless various styles of parsers are mixed in the generation of a single parse tree (for example one parser gets called within another and their parse trees are later merged), this becomes irrelevant as one parser will only ever produce one kind of intermediate node.

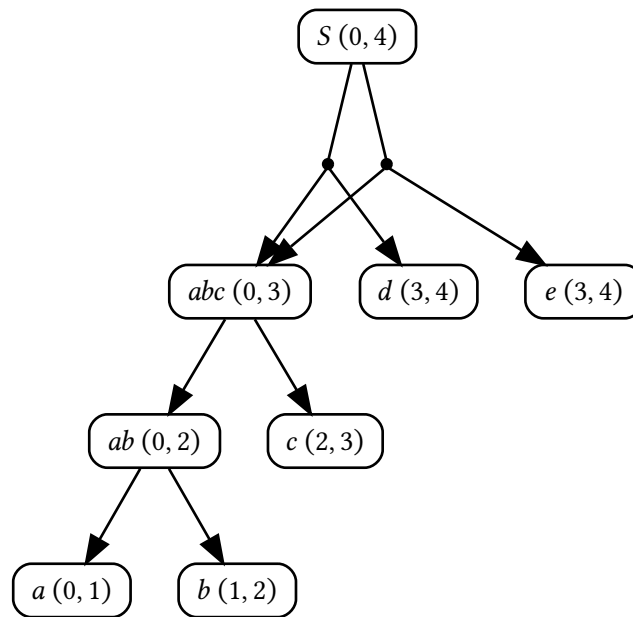


Figure 2.2: LL-style binarized SPPF for the parse in 2.5.2.

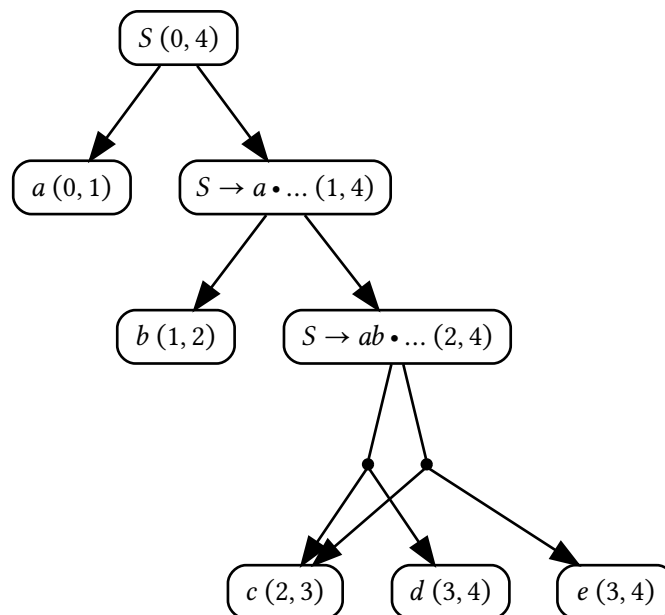


Figure 2.3: LR-style binarized SPPF for the parse in 2.5.2.

Chapter 3

Context-free LR parsing

LR parsing was first described by Knuth in [5]. An LR parser is an efficient simulation of a PDA recognizer for a context-free grammar, which is made deterministic using the Rabin-Scott powerset construction ([3]). An example is in **Figure 3.1**. The driving automaton is usually represented in a tabular form, similar to **Table 1.2**, called the *parse table*.

The stack of an LR recognizer contains states, initially the single start state. The algorithm performs two kinds of actions: it can read an input symbol, push its current state on the stack and go to some state (*shift*), or it can pop a given number of states off the stack and then shift a nonterminal, without reading any symbols (*reduce*). The parser version, in addition, also pushes the derivation trees on the stack.

Sometimes the underlying PDA prescribes conflicting parser actions: reduce/reduce conflicts mean multiple reductions are possible from a state, whereas shift/reduce conflicts indicate that the contents of the stack can both be reduced or the next token can be shifted instead. (Shift/shift conflicts only occur in the presence of lexical ambiguities and are not avoidable by static analysis.) The example in **Figure 3.1** has a shift/reduce conflict on the input symbol a in state #10, where the parser has to choose between reducing by $A \rightarrow bA \bullet$ and shifting a in $A \rightarrow A \bullet aa$.

A common disambiguation technique is peeking ahead in the input stream to see if a reduction is compatible with the next few tokens. This can be used to reduce the number of conflicts in the DFA: whenever an ε -transition is taken in the NFA, the context⁽¹⁾ of this transition is recorded. These strings

¹ All possible terminal sequences of length at most k (for some finite k , usually as low as 1) that can follow the nonterminal in the current rule being recognized.

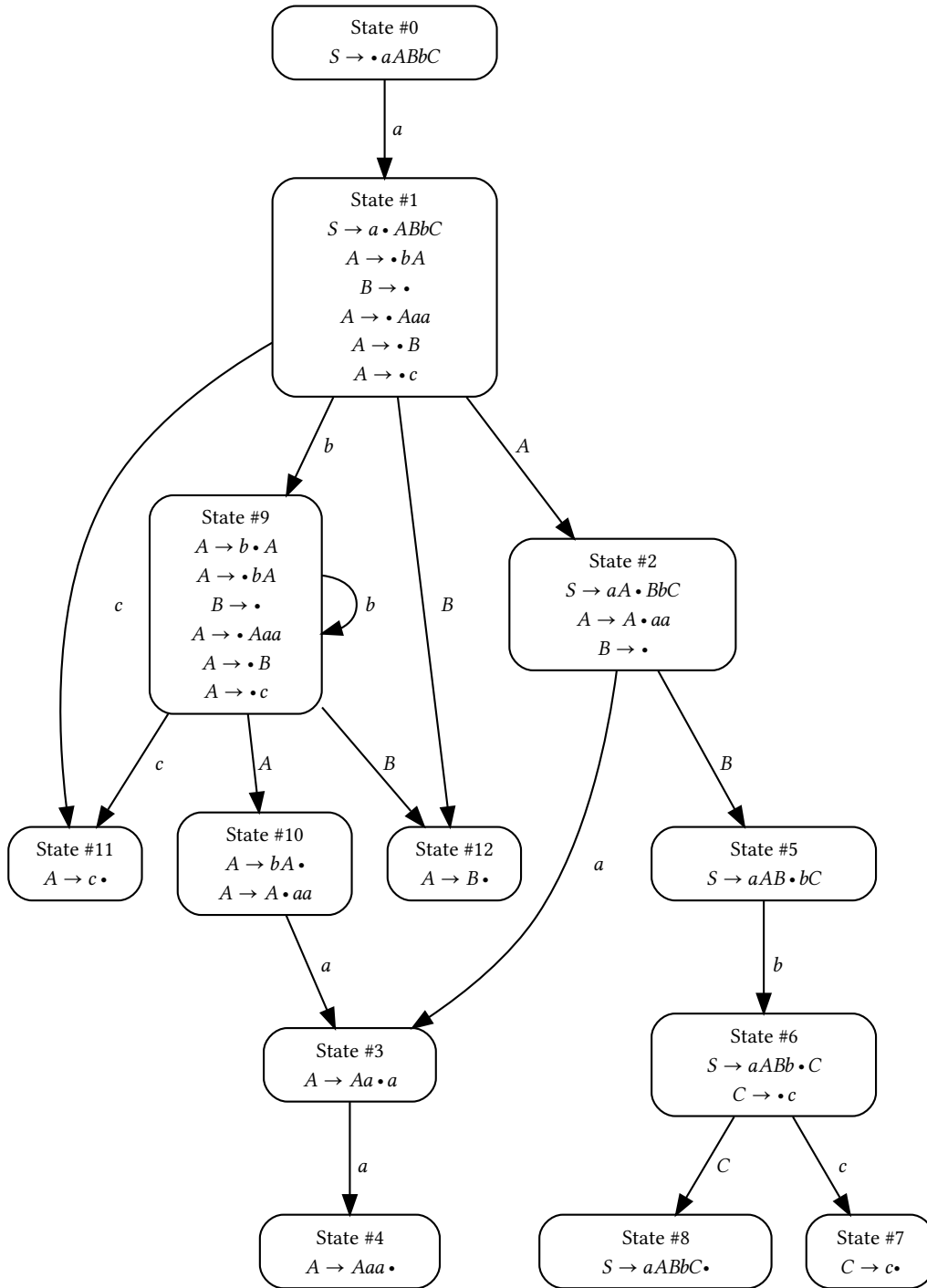


Figure 3.1: An LR-DFA from the PDA in Figure 1.7.

over Σ^* are referred to as lookaheads. An *LR(k) item* is a pair $\langle g, \Lambda \rangle$ where g is a grammar slot and $\Lambda \subseteq \bigcup_{i \in \{0, 1, \dots, k\}} \Sigma^i$ is a set of strings of length at most k .

For example, if the NFA is in a state $A \rightarrow \alpha \cdot Xa\beta$, then it is clear that whatever input X will be matched against, when the NFA returns to parsing this rule, it will have to read an a after, therefore an X should only ever be reduced in the context of this rule iff it is followed by an a . The terminal a will become the lookahead of any LR(1) item generated at $A \rightarrow \alpha \cdot Xa\beta$.

An LR automaton with k symbols of lookahead is an LR(k) automaton. A grammar whose LR(k) automaton is conflict-free is an LR(k) grammar. A language is LR(k) if it has an LR(k) grammar. Any language that has an LR(k) grammar for some $k \geq 2$ also has an LR(1) one: this, in effect, means that LR(1) grammars are just as powerful as those with longer lookaheads. For this reason Knuth's original LR(1) automaton is also known as the Canonical LR.

Lookaheads reduce conflicts as they restrict the immediate right context in which reductions can be performed. The price is paid in automaton size: whereas the LR(0) DFA we built for the phrase structure grammar of C had somewhat over 500 states, the LR(1) had over 14000. This is less of a problem now than it was back in the late mid-20th century, but a 30-fold increase in parser size is still not immaterial due to the still limited availability of fast processor caches. Efficient lookahead-set construction for deterministic LR parsers without an explosion in the number of states is highly researched area of LR parsing ([11], [20], [59], [26], [48], [52]).

DeRemer [9] gave two practical methods for augmenting LR(0) automata without increasing the number of states. SLR uses the FOLLOW set of a nonterminal as lookaheads. (The FOLLOW set of a nonterminal is the set of terminals which can follow it in some derivation: $\text{FOLLOW}(A) = \{t \in \Sigma : \exists \alpha, \beta \in \Gamma^* : S \xRightarrow{+} \alpha A t \beta\}$.) The other, LALR, merges states of the LR(1) automaton that only differ in their lookaheads. A LALR(1) automaton is at most as powerful as an LR(1) automaton, as they can introduce reduce/reduce conflicts where the CLR automaton had none, but its success tells us that this is a small price to pay for practical near-LR(1) recognition power with only the number of states of an LR(0) automaton. Construction of LALR(1) lookahead sets without constructing the full CLR(1) automaton first are discussed in [23] and [59].

3.1 Generalized parsing

A nondeterministic automaton can be thought of as being able to clone itself the required number of times to follow different paths of execution. In theory there is nothing to stop us, when the LR-DFA turns out to have conflicting actions or is faced with a lexical ambiguity, from completely cloning the

parser's state to execute multiple paths, but this is generally resource intensive and nonterminating⁽²⁾ if (a) the grammar contains hidden left recursion or (b) is cyclic.

Tomita's [24] GLR parser eliminates cloning the parser by generalizing the stack into a directed graph known as the graph-structured stack (GSS). The GSS is an append-only data structure that records all possible execution stacks of the LR parser. The nodes of the graph represent states of the LR-DFA and the edges are labeled with the derivation trees for the transition⁽³⁾. The edges point "backwards", i.e. if the parser performs a shift from node p on the symbol s to the state q , then there will be an edge $q \xrightarrow{s} p$.

When the deterministic LR parser would pop $|\text{RHS}(p)|$ symbols from the stack and shift $\text{LHS}(p)$, the GLR instead finds all paths of length $|\text{RHS}(p)|$ and creates a new edge labeled $\text{LHS}(p)$.

Nodes of the GSS are grouped into disjunct sets called *generations*. A generation U_i contains exactly the nodes in the GSS that were created by reading input up to position i . There can be no two nodes for the same state in a single generation and there can be at most one edge from any node to another.

3.1.1 The failure of Tomita's GLR

In Tomita's original parser problems arise when edges are added to nodes that themselves have ingress edges from parents that are already processed. Using the example in [45] we demonstrate the error of Tomita's original GLR algorithm.

Consider the grammar $S \rightarrow aSA \mid \epsilon$, $A \rightarrow \epsilon$ and the input aa . The associated LR automaton is in **Figure 3.2**.

The algorithm starts by creating a node labeled 0 representing the initial state in U_0 . State 0 has a reduction by $S \rightarrow \epsilon$ that yields the empty tree and a shift on a . As that is the next symbol in the input, a node in U_1 is created for state 1 with an edge pointing back to 0 in U_0 . There is a reduction in state 1 by the rule $S \rightarrow \epsilon$, which traces back 0 edges (as the right-hand side is empty), arriving at the same node where it started. From state 1 the resulting S is shifted to create a node for state 2 in the same generation, with an edge pointing back to state 1. In state 2 a similar process leads to the creation of a node for state 3 that has an edge labeled A pointing back to that of state 2.

State 3 has a reduction $S \rightarrow aSA$ with 3 symbols in the RHS so 3 edges of the GSS are traced back arriving at the initial node. Because the LR state associated with this node does not have a transition for the

² [25]

³ Tomita's original description of the GLR algorithm uses edges with labels alternating between parser states and transition symbols; that convention has now been abandoned.

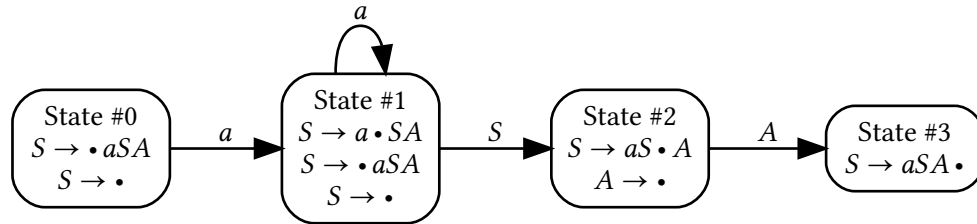
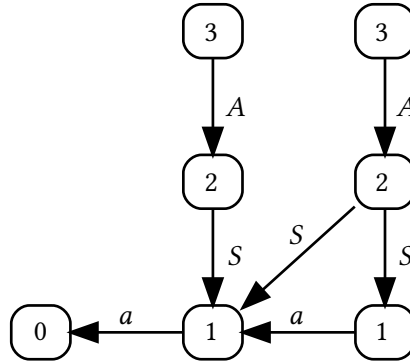

 (a) LR automaton for $S \rightarrow aSA \mid \epsilon, A \rightarrow \epsilon$.

 (b) The GSS for the input aa .

Figure 3.2: Tomita's GLR at work for a grammar with hidden right recursion.

resulting symbol S it is not shifted, but the derivation is nevertheless accepted as a valid derivation for S .

Having completed every reduction available the next input symbol is shifted. Since only state 1 has a shift on a a new node is created in U_2 labeled a pointing back to state 1 in U_1 . Then, once again, the nodes for state 2 and 3 are created, this time in U_2 .

It is again in state 3 that we can reduce by $S \rightarrow aSA$, tracing back to the node for state 1 in U_1 . This is not the initial node, so the parse isn't accepted, but state 1 has a shift on S going to state 2, so an edge from the node for state 2 in U_2 is created pointing back to the node in U_1 .

State 2 has one reduction by the rule $S \rightarrow \epsilon$ resulting in a nulled S being created and moving to state 3. A node for state 3 already exists and has an edge labeled A back to state 2, therefore this path is not followed and the algorithm terminates.

As the processing of the second a did not result in a path back to the initial node being found, the input is rejected, even though it is clearly valid.

The problem was that after the edge between state 2 of U_2 and state 1 of U_1 was created the new reduction path was not discovered.

3.1.2 The analysis of Tomita's error

Reductions are processed whenever a node is created for a state that has a completed item. The reduction would have been performed in state 3 but it was blocked by the already present nulled edge for A .

Whenever a reduction is traced from some node u in U_j via some path to a node v in U_i the algorithm creates an edge from some node w also in U_j to v . As the example clearly demonstrates, w is not necessarily the same as u . The reduction can become blocked if w already has its parents created.

During reductions edges are only created for nodes in the current generation, therefore the only way u can be a parent of w is if they're in the same generation, U_j .

If the rule by which the reduction is performed has the shape $S \rightarrow \alpha S \beta$ then if u represents the state with the item $S \rightarrow \alpha S \beta \bullet$ and w is for $S \rightarrow \alpha S \bullet \beta$ then, because u and w are in the same generation, u must have a nulled path to w and so β must be nullable.

3.1.3 Tomita's attempted fix

Tomita was aware of this deficiency in his parser and attempted to fix it by ensuring that no reduction ends up shifting to a node that already has parents. His reasoning is correct but his solution was not.

His idea was that generations should further be subdivided so that any parents of a node are always in a higher generation and are created later, therefore solving the issue of pre-existing parents.

Figure 3.3 shows the GSS created using this approach. The reader can verify that a reduction from $U_{2,3}$ does indeed accept the input aa .

When cycles are not allowed to form in the GSS (as all edges must strictly be right-to-left) Tomita's new GLR once again fails on hidden left recursion, like the original LR parsers do ([25]).

Suppose the grammar was $S \rightarrow ASa \mid \epsilon$, $A \rightarrow \epsilon$, which is hidden left recursive. The initial node for state 0 is created, which immediately produces a nulled instance of A that is shifted to state 1 in subgeneration 1. State 1 has two nulled reductions, one by A and one by S . Shifting the S creates a node for state 2 in $U_{0,2}$ and shifting A creates one for state 1. From state 2 an a could be shifted to state 3 in U_1 but from state 1 in $U_{0,2}$ a nulled A is reducible, resulting in a new subgeneration $U_{0,3}$ with two nodes, one for states 1 and 2 each, and the process goes on for infinity.

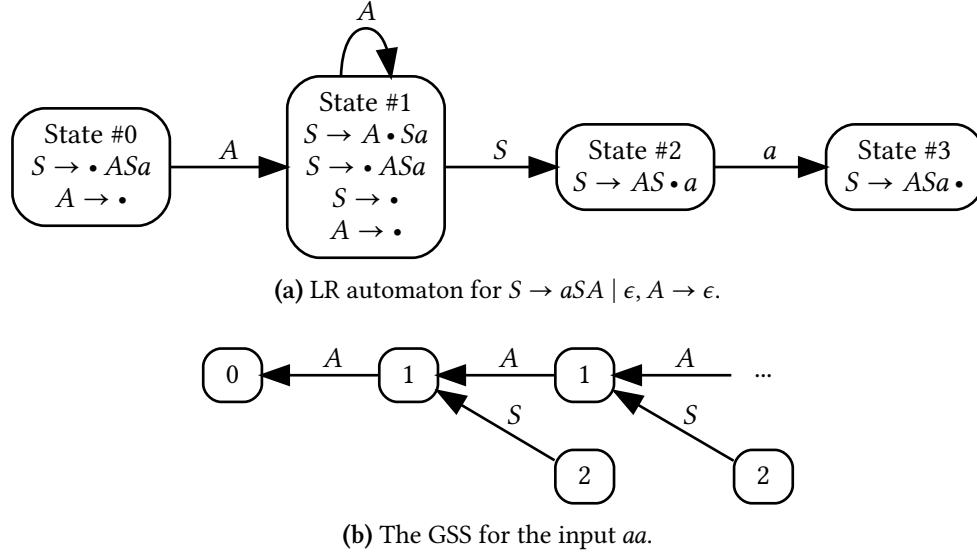


Figure 3.3: Infinite recursion of Tomita's GLR for an ϵ -grammar with hidden left recursion.

Tomita was aware that his algorithm failed on some grammars: “The number of e -symbols is, however, always finite, as we assumed in the preceding sections. Our algorithm will perform as if there were an appropriate number of e 's as normal input symbols between every adjunct pair of input symbols. Thus, an input string $a_1 \dots a_n$ can be thought of as the following: $e \dots ea_1e \dots ea_2 \dots a_ne \dots e\$$ ”. ([24, p. 56])

Arguably a nonterminating algorithm is still better than one that ends with a wrong answer, but still not suitable for generalized parsing.

3.1.4 Farshi's fix

The fix by Nozohoor-Farshi ([29]) builds on the original GLR algorithm: whenever the resulting nonterminal of a reduction is shifted to a node w that already has a parent, all reductions from nodes u that can have a reduction path through w must be reevaluated.

While this is correct and fully generalizes Tomita's GLR, the reevaluation is not beneficial to the algorithm's runtime performance.

3.1.5 Right-nulled GLR

Scott and Johnstone ([45]) found a suprisingly simple fix of Tomita's GLR. As discussed earlier, GLR fails in the presence of some right-nullable rules by not being able to properly perform nulled reductions because they had already been completed.

The authors noticed that the completion of an item is not a necessary condition for reduction, only the non-nulled parts have to be matched. That is, for a production $S \rightarrow \alpha S \beta$, where β is nullable, reductions are not only initiated from the item $S \rightarrow \alpha S \beta \bullet$ but also from $S \rightarrow \alpha S \bullet \beta$.

By eager performance of the reductions their right ends are not necessarily present in the GSS. For example, just as the node for state 1 in U_1 is created, it has a reduction $S \rightarrow a \bullet SA$ (because the suffix SA is nullable), but the edges for S and A have not yet been created.

As the only edges that might not be present in the GSS are those for *nulled* nonterminals, their derivation trees can be precomputed and attached to the parse table. In this case the parse table for state 1 would not only contain a reduction by the rule $S \rightarrow a \bullet SA$ but also that to the derivation found by tracing a in the GSS a prebuilt suffix of SA must be appended.

3.1.6 Binarized RNGLR

The worst-case complexity of a context-free parser is $O(n^k + 1)$ (where k is the length of the longest RHS in the grammar) which is of unbounded polynomial order. For example, grammars of the form $S \rightarrow S^k \mid \dots \mid S^1 \mid x \mid \epsilon$ exhibit this behavior.

The fastest parsers are cubic, but this is achieved by the grammar having no right-hand sides longer than 2 symbols. Such a transformation is always possible but often destroys the inherent semantics embedded in the structure of the grammar itself.

The BRNGLR of Scott and Johnstone ([46]) does the binarization of the grammar on the fly, whenever a reduction is performed. For every rule that is at least 3 symbols long, instead of tracing the whole rule in one go, the GSS is traversed two edges a time, creating *bookkeeping nodes* to represent the partially completed reduction.

The bookkeeping nodes are not associated with the states of the LR automaton, as other nodes in the GSS are, but with grammar slots of the actual production being reduced. These grammar slots record how much of the reduction has already been performed.

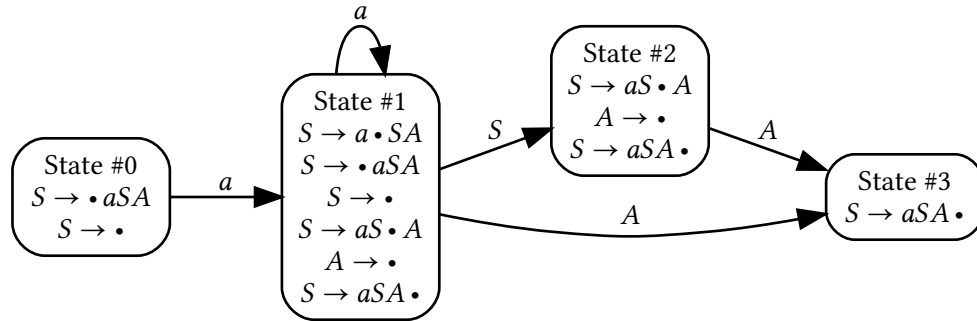
3.1.7 ϵ -GLR

Observing the GSS built by the canonical GLR for the grammar $S \rightarrow aSA \mid \epsilon, A \rightarrow \epsilon$ and the input aa one can notice that 4 out of the total 7 edges created are nulled. Such edges carry no information as they are independent of the input and always created, regardless if eventually used, every time a state with a possible nulled reduction is reached.

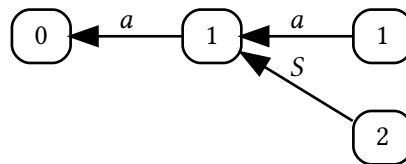
Similar to the shortcutting of right-nulled ends done by the RNGLR parser, it is possible to eliminate not only the right ends, but in fact any nullable parts of a reduction and still have a correct generalized parser for all context-free grammars.

The ϵ -GLR of Nederhof and Sarbo ([33]) uses a construction of the LR-DFA which allows “stepping over” nullable nonterminals. More formally, the closure operation used in the creation of the LR automaton is extended so that a state containing some item $S \rightarrow \alpha \bullet A\beta$ does not only include items $A \rightarrow \bullet \gamma$ but also $S \rightarrow \alpha A \bullet \beta$ if A is nullable. This new closure essentially merges all states of the LR automaton that are reachable via nulled transitions.

The ϵ -GLR automaton for our example grammar is in **Figure 3.4a** and its associated GSS for the input aa is in **Figure 3.4b**.



(a) ϵ -GLR automaton for $S \rightarrow aSA \mid \epsilon, A \rightarrow \epsilon$.



(b) The GSS for the input aa .

Figure 3.4: The ϵ -GLR in presence of hidden right recursion.

With the nulled edges of the GSS being all eliminated, the complete reconstruction of a derivation only using the GSS is not possible. Rather than searching for paths of a fixed length reductions are traced

symbol by symbol. Whenever the next symbol of the reduction is not nullable it must be matched in the GSS; for nullable symbols it is also allowed that its nulled tree is simply attached to the derivation without having to locate it in the parser's history graph.

3.2 Nullable-eliminated GLR

Nullability makes context-free parsing *hard*. [25] identified two conditions under which an LR parser might be nonterminating: cycles and hidden left recursion. Many attempts to generalize LR parsers ([24], [27], [36]) have failed to terminate in the presence of hidden left recursion, a condition ultimately caused by nullability⁽⁴⁾.

There is a well-known algorithm to remove ϵ -productions from a grammar: should there exist a rule $S \rightarrow \alpha X \beta$ ($S, X \in V, \alpha, \beta \in \Gamma^*, X \xrightarrow{*} \epsilon$), add a new rule $S \rightarrow \alpha \beta$; finally, remove all productions with empty or non-productive right-hand sides, repeating if necessary. This can lead to an exponential explosion in the number of productions.

Nederhof and Sarbo ([33]) and Scott and Johnstone ([45]) have both modified Tomita's original GLR to become a truly general context-free parser. The BRNGLR of Scott and Johnstone ([46]) in addition achieves at worst cubic complexity by ad hoc binarization of the grammar, producing binarized SPPFs and rendering the preprocessing of the grammar unnecessary.

Our first parser presented in this thesis is a combination of these previous efforts with a few smaller modifications of our own.

One such modification involves the analysis of the items in an LR automaton and the observation that required-right-nullable items are useless outside the kernel of the LR states. Another involves the treatment of bookkeeping nodes: instead of bloating the GSS with extra nodes our algorithm keeps a hash table of already processed states so that no modification of the GSS is necessary.

As we have found value in nondeterministic tokenization, the parser is also modified appropriately.

⁴ Cycles might or might not be nullable. The production $S \rightarrow S \mid x$ is cyclic but not nullable.

3.2.1 Optimizing the DFA and parse table construction

Items which only have required nullable symbols remaining are useless⁽⁵⁾: when creating the ε -closure of a state any items of the form $X \rightarrow \alpha \bullet \xi$ can be omitted, where ξ is required nullable⁽⁶⁾.

In this construction exactly the right-nullable kernel items of non-starting states are reduction items.

Omission of required-right-nullable kernel items would result in some reduction items not being created in the automaton. In the right-nulled automaton a reduction can be initiated from a state iff it contains an item $X \rightarrow \alpha \bullet \beta$ where $\beta \xrightarrow{*} \varepsilon$. As β might be required nullable the item can be absent. This happens trivially when the item is complete.

Distinguishing states solely on their required-right-nullable-eliminated kernel overcompresses the DFA. Suppose two distinct states only have the rules $X \rightarrow \xi \bullet x$ and $Y \rightarrow v \bullet y$. These obviously lead to different states, one containing $X \rightarrow \xi x \bullet$, the other $Y \rightarrow vy \bullet$, by the transitions on x and y , respectively. If, however, the required-right-nullable items are eliminated, there is nothing to tell these states apart and they get merged.

Theoretically the NEGLR algorithm *can* work with such an automaton (as it *matches* the symbols in a rule to those present in the GSS, rather than just *assumes* their presence), but this would create a lot of non-actionable reductions in common reduction states and would reduce the parser's efficiency.

3.2.2 The NELR(0) algorithm

In **Algorithm 2** we present the code to build an NELR(0) automaton from a context-free grammar.

The construction of a NELR(0) automaton is similar to that of standard LR(0). The difference is in lines 19 and 21, where the nullable nonterminals are skipped. All insertion operations are guarded to prevent the addition of items with required nullable right ends. GET-STATE-FROM-ITEMS is a helper function which is given a set of items and finds, or in case of nonexistence, creates a state with the given items as its kernel.

Lookaheads for NELALR(k) or NELR(k) can be calculated using the same principles as for a non ε -eliminated automaton.

⁵ An item $X \rightarrow \alpha \bullet \beta$, where β is required nullable can only ever generate nulled transitions on the symbols in β because all items generated from the symbols in β must be required nullable, too. Because ε -transitions and nulled reductions are *never* performed, these items can never be acted upon and therefore are useless.

⁶ The empty string ε is, by definition, required nullable.

BUILD-AUTOMATON(*grammar*)

```
1: let start-state  $\leftarrow$  CREATE-STATE()
2: for each production in GET-PRODUCTIONS(START-SYMBOL[grammar]) do
3:   if not REQUIRED-NULLABLE[production] then
4:     add  $\langle$ production, 0 $\rangle$  to KERNEL[start-state]
5: while there remains an unprocessed state do
6:   let state  $\leftarrow$  an unprocessed state
7:   let transition-sets  $\leftarrow$   $\emptyset$ 
8:   while there remains an unprocessed item in state do
9:     let  $\langle$ production, index $\rangle$   $\leftarrow$  an unprocessed item
10:    if REQUIRED-RIGHT-NULLABLE[production, index] then
11:      continue
12:    let symbol  $\leftarrow$  production[index]
13:    if not REQUIRED-NULLABLE[symbol] then
14:      add  $\langle$ production, index + 1 $\rangle$  to transition-sets[symbol]
15:    if NONTERMINAL[symbol] then
16:      for each production in GET-PRODUCTIONS(symbol) do
17:        if not REQUIRED-NULLABLE[production] then
18:          add  $\langle$ production, 0 $\rangle$  to CLOSURE[state]
19:        if NULLABLE[symbol] then
20:          if not REQUIRED-RIGHT-NULLABLE[production, index + 1] then
21:            add  $\langle$ production, index + 1 $\rangle$  to CLOSURE[state]
22:    for each symbol in GET-TRANSITION-SYMBOLS(transition-sets) do
23:      TRANSITIONS[state, symbol]  $\leftarrow$  GET-STATE-FROM-ITEMS(transition-sets[symbol])
```

Algorithm 2: Generating a NELR(0) automaton.

3.3 Performing reductions

Once all of the non-nulled parts of the right-hand side of a rule have been matched and built into the GSS the *reducer* takes over and traces the paths taken by the recognizer back to the initial node where the matching of the rule began, from where it shifts the newly assembled nonterminal.

The original LR and GLR parsers would push all symbols, including nulled ones, onto the stack. The ε -GLR of Nederhof and Sarbo eliminates this at the cost of more searching: whenever a nullable nonterminal is to be located on the stack the tracing algorithm has to make a choice on whether a non-nulled version of this symbol should be read from the stack (if it was matched at all) or it should be made up and added to the derivation as a nulled instance.

The BRNGLR of Scott, Johnstone and Economopoulos ([46]) performs reductions longer than 2 symbols in pieces of 2: the last two symbols are “popped off” the GSS and an intermediate bookkeeping state is created to represent the partially completed reduction.

Bookkeeping nodes serve a double purpose: their egress edges hold the intermediate nodes and, per the usual rules of creating nodes and edges in the GSS, they guarantee that the tracing of a grammar slot happens only once per node. Even though the latter is not necessary for correctness, it provides the cubic runtime complexity bound for any nonbinary grammar.

We implement on-the-fly binarization a bit differently. In the BRNGLR pending reductions are represented as triples $\langle v, q, m \rangle$ where v is the GSS node from which the reduction is to be performed/continued, q is the rule to reduce by and m is the remaining number of symbols in (the length of) the reduction. Our representation is extended to a quadruple: $\langle v, q, m, t \rangle$ with t being the SPPF node for the part of the reduction already performed. When starting a reduction, the initial value of t can be read from the parse table: it is **null** for reductions which start at the end or the node representing the nulled right end otherwise.

In our implementation no bookkeeping nodes are created, in fact the GSS is not modified unless an edge must be created for a completed reduction. The reduction descriptors are stored separately in a hash table and are thrown away after a generation in the GSS is processed.

3.3.1 Nulled symbols in reductions

Once the nulled symbols are eliminated from the GSS, the reduction process, when encountering a nullable symbol in the rule, has to make a choice whether to assume that the symbol was nulled and attach the corresponding empty node to the SPPF, or whether it should look for the derivation of that symbol in the GSS. The safe choice, as is the general governing principle of the GLR technique, to try *both* and let the failed paths die. Unlike the non-eliminated algorithm, failed predictions can lead to failures in both the shifter and the reducer part of the parser. (These failures are not *fatal*: valid parses will still be found, but some reductions will have to be abruptly abandoned.) We will demonstrate this through an example.

Consider the grammar $S \rightarrow sAB, A \rightarrow a \mid \varepsilon, B \rightarrow b$ and the input sab . The associated LR DFA and the GSS are shown in figure **Figure 3.5**.

We will skip forward in the explanation to the point where the parser is processing the reduction by $S \rightarrow sAB \bullet$ in state #4. The nonterminal B is not nullable, so it is read from the GSS and the remaining

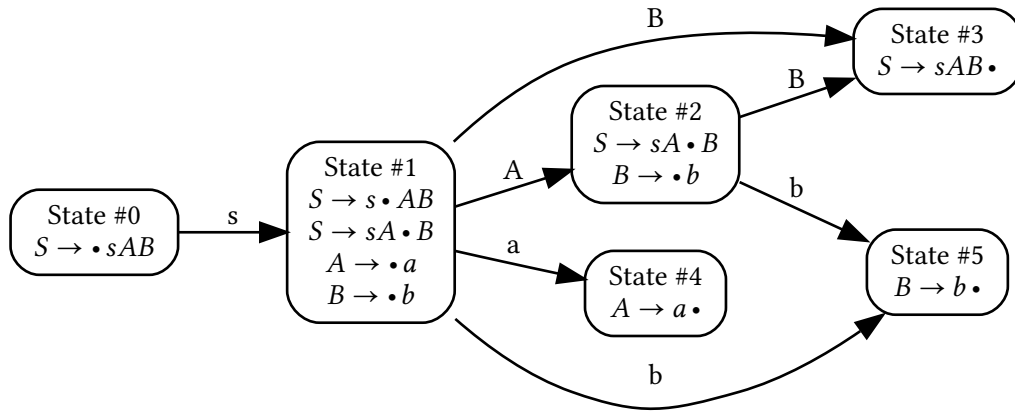
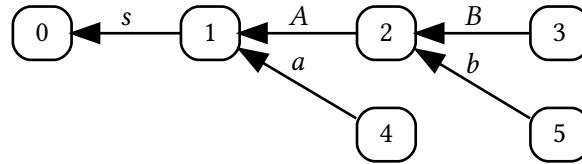

 (a) NELR(0) automaton for $S \rightarrow sAB, A \rightarrow a \mid \epsilon, B \rightarrow b$.

 (b) The GSS for the input ab .

 Figure 3.5: Invalid reduction trace when A is assumed to be nulled.

of the reduction, $S \rightarrow sA \cdot B$ is queued to be continued from state #5 in U_2 . The next symbol to be reduced, A , is nullable, and the reducer has to make a choice whether to read it from the GSS or make it up as a nulled symbol. In the former case the edge between states #5 and #1 is traversed the reduction continues as normal, ultimately yielding the start symbol, S . If, however, the nulled path is taken, the reduction is already complete in state #5, from which there is no edge labeled s that would be required to complete the reduction. This kind of error does not occur in the non-eliminated automata.

Using the grammar $S \rightarrow AB, A \rightarrow a \mid \epsilon, B \rightarrow b$ and the input ab it can be shown that the issue is not limited to the reduction algorithm, but sometimes the resulting nonterminal can not be shifted on the stack.

It would be possible to use a representation of the underlying automaton complete with LR items to prevent these errors from happening (before nulling some symbol we would check whether the precedent item is also present in the state), but that is not necessary. As mentioned earlier, this misbehavior creates some superfluous edges and states in the GSS, but does not block the parser from finding all correct parses. In our implementation, we simply augment the algorithm with a few extra checks to verify that the operation to be performed is indeed viable and ignore it if it is not.

3.4 The NEGLR parser

The (B)RNGLR parsers depend on right-nulled parse tables for *correctness* as they ensure that all reduction paths are discovered even if edges are added to a node with an already processed parent. This can only happen when the rule is right-nullable. Because the nullable-eliminated algorithm never creates edges within the same generation, our parser does not *require* this optimization and is correct without ([33]). We still use right-nulled tables for *simplicity* and a bit of *performance*: right-nulled tables allow us to pre-generate the parse trees for the nulled right end of a production and always begin the reduction from the last non-nulled symbol.

The parser, as usual, consists of two main parts, the *reducer* (lines 11–16), which traces back the paths of and creates the edges for nonterminals, and the *SHIFTER*, which processes the terminals. Reductions are processed one symbol at a time: the first (non-nulled) symbol is read from the GSS, while the rest are either read from the GSS or generated on the fly (if they are nullable). A reduction is considered complete if the dot reaches the left end of the right-hand side of the rule, where the resulting nonterminal is shifted and the reductions through the new edge (if one was created) are evaluated.⁽⁷⁾

Reductions retrieved from the parse table via *VARIABLE-REDUCTIONS* have three components: the production being reduced, the position of the dot and the right-nulled part of the tree. Pending reductions, which are descriptions of incomplete reductions created by the parser, are labeled with the production being reduced, the position of the dot, the GSS node from which the tracing is to be continued and the SPPF for the part of the rule after the dot.

The GSS nodes store four pieces of data: the ID of the DFA state they represent, the input position at which they were created, the grammar symbol labeling its egress transitions and the set of edges leading out of it. An edge has a target, which is the GSS node representing the DFA state from which the transition was taken, and a label, which is the SPPF node representing all derivations of the transition symbol for the input between the two nodes.

The parser does not *return* the parse trees but *yields* them: it is a coroutine that produces parse trees for prefixes of ever increasing lengths of the input. The algorithm terminates when the set of valid parses cannot be further extended.

⁷ The algorithm is presented with clarity and not performance in mind. There are numerous practical optimization opportunities for real-world implementations.

NEGLR(*start-symbol*, *start-position*)

```
1: if NULLABLE[start-symbol] then
2:   yield NULLED-TREE[start-symbol]
3: CREATE-INITIAL-NODE(start-position)
4: while there exists an unprocessed generation in the GSS do
5:   let current-position  $\leftarrow$  MIN(position of any unprocessed generation)
6:   let current-generation  $\leftarrow$  GENERATION[current-position]
7:   let processed-reductions  $\leftarrow$   $\emptyset$ 
8:   let pending-reductions  $\leftarrow$   $\emptyset$ 
9:   for each node in current-generation do
10:    SHIFTER(node)
11:    for each reduction in VIABLE-REDUCTIONS(node, current-position) do
12:      for each edge in node do
13:        START-REDUCTION(reduction, edge)
14:      while COUNT[pending-reductions] > 0 do
15:        let reduction  $\leftarrow$  POP(pending-reductions)
16:        CONTINUE-REDUCTION(reduction)
17:      for each node in current-generation do
18:        SHIFTER(node)
19:      if EXISTS-NONTERMINAL-NODE(start-symbol, start-position, current-position) then
20:        yield GET-NONTERMINAL-NODE(start-symbol, start-position, current-position)
```

START-REDUCTION(*reduction*, *edge*)

```
1: let tree  $\leftarrow$  GET-TREE(PRODUCTION[reduction], LENGTH[reduction] - 1, POSITION[TARGET[edge]])
2: if NULLED-PART[reduction]  $\neq$  null then
3:   ADD-CHILDREN(tree, LABEL[edge], NULLED-PART[reduction])
4: else
5:   if LENGTH[reduction] > 1 then
6:     tree  $\leftarrow$  LABEL[edge]
7:   else
8:     ADD-CHILDREN(tree, LABEL[edge])
9: ADD-PENDING-REDUCTION(reduction, TARGET[edge], tree)
```

CONTINUE-REDUCTION(*reduction*)

```
1: let symbol  $\leftarrow$  SYMBOL[reduction]
2: if NULLABLE[symbol] then
3:   EXTEND-REDUCTION(reduction, TOP[reduction], NULLED-TREE[symbol])
4: if SYMBOL[TOP[reduction]] = symbol then
5:   for each edge in EDGES[TOP[reduction]] do
6:     EXTEND-REDUCTION(reduction, TARGET[edge], LABEL[edge])
```

Algorithm 3: The NEGLR parser.

EXTEND-REDUCTION(*reduction*, *node*, *left*)

- 1: **let** *tree* $\leftarrow$ GET-TREE(production[*reduction*], length[*reduction*] – 1, position[*node*])
- 2: ADD-CHILDREN(*tree*, *left*, TREE[*reduction*])
- 3: ADD-PENDING-REDUCTION(*reduction*, *node*, *tree*)

FINISH-REDUCTION(*production*, *node*)

- 1: **let** *result-node* $\leftarrow$ GET-SHIFT-NODE(*node*, LHS[*production*], current-position)
- 2: **if** *result-node* $\neq$ null **then**
- 3: **if** *result-node* has no edge to *node* **then**
- 4: **let** *edge* $\leftarrow$ CREATE-EDGE(*result-node*, TOP[*reduction*], TREE[*reduction*])
- 5: **for each** *reduction* **in** VIABLE-REDUCTIONS(*result-node*, current-position) **do**
- 6: START-REDUCTION(*reduction*, *edge*)

ADD-PENDING-REDUCTION(*reduction*, *node*, *tree*)

- 1: **let** *length* $\leftarrow$ length[*reduction*] – 1
- 2: **if** *length* = 0 **then**
- 3: FINISH-REDUCTION(production[*reduction*], *node*)
- 4: **else if** *processed-reductions* does not contain $\langle$ production[*reduction*], *length*, *node* $\rangle$ **then**
- 5: **add** $\langle$ production[*reduction*], *length*, *node* $\rangle$ **to** *processed-reductions*
- 6: **add** $\langle$ production[*reduction*], *length*, *node*, *tree* $\rangle$ **to** *pending-reductions*

SHIFTER(*node*)

- 1: **for each** *token* **in** GET-TOKENS(current-position, EXPECTED-TERMINALS(*node*)) **do**
- 2: **let** *result-node* $\leftarrow$ GET-SHIFT-NODE(*node*, SYMBOL[*token*], RIGHT-EXTENT[*token*])
- 3: CREATE-EDGE(*result-node*, *node*, GET-TERMINAL-NODE(*token*)))

GET-TREE(*production*, *dot-position*, *left-extent*)

- 1: **if** *dot-position* = 0 **then**
 - 2: **return** GET-NONTERMINAL-TREE(LHS(*production*), *left-extent*, current-position)
 - 3: **else**
 - 4: **return** GET-INTERMEDIATE-TREE(*production*, *dot-position*, *left-extent*, current-position)
-

Algorithm 3: The NEGLR parser. (cont.)

If one does not apply the optimizations described in 3.2.1 but directly uses the binarized right-nulled tables from [46] instead, it must be verified that the reduction returned by `Viable-Reductions` has a nonzero length and that the symbol preceding the dot is indeed the label of the inbound edges.

Should right-nulled tables not be used, in addition to the above, completed reductions must be verified to not being nulled (i.e. the left and right extents of the resulting parse tree are not the same). Allowing nulled reductions to complete would result in the creation of nulled edges in the GSS.

The parser can be implemented with few changes to create either binarized or non-binarized trees. We opted to create the binarized versions because this makes the parser worst-case cubic. In order create non-binarized SPPFs, instead of the binarized tree nodes keep passing on the list of children discovered so far.

Chapter 4

Extended context-free parsing

A context-free grammar is a quadruple $G = \langle V, \Sigma, P, S \rangle$, where V and Σ are finite, disjoint sets of symbols, $P \subset V \times \Gamma^*$ is the finite set of *productions* of the grammar and S is the start symbol.

Let $P_A = \{\alpha \in \Gamma^* : (A \rightarrow \alpha) \in P\}$ be the set of right-hand sides of productions which have A as a left-hand side symbol. P_A , for every $A \in V$, is essentially a finite language over Γ .

Suppose P is relaxed to be a possibly infinite relation where each P_A is a *regular* language. Grammars satisfying this new criterion generate the same set of languages as the context-free ones do, but often allow for easier specification, as regular expressions or EBNF can be used to describe the language of each P_A . These are known as *extended context-free grammars* (ECFG) and have been studied at least since [19]⁽¹⁾.

A small restriction for the maintenance of simpler discussion is that we allow only one production for each nonterminal. Rules previously specified as $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots, A \rightarrow \alpha_n$ will now have to be written as $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$. Real-world implementations may decide not to impose this restriction.

Rewrite rules, as discussed in 4.1.1 (also in [21]), are often used to transform an ECFG into a usual one. The transformation preserves the *language* generated, but the *grammar* is altered. This results in parse trees that contain automatically generated symbols, where the tree must be reshaped during or after parsing. This is not optimal.

We are not the first to investigate LR parsing for ECFGs, as the works of Madsen and Kristensen ([18]), Purdom and Brown ([22]) and Celentano ([21]) demonstrate. Their analysis, however, does not cover

¹ In early literature they were called regular right part (RRP) grammars. As for when they were introduced, [19] cites an 1970 work of DeRemer titled “Extended LR(k) grammars and their parsers” a copy of which we have been unable to locate.

```
Production → Symbol → Alternation
Alternation → Concatenation
Alternation → Alternation | Concatenation
Concatenation → Concatenation Term
Concatenation →
Term → Symbol
Term → ( Alternation )
Term → Option
Term → Repetition
Term → Kleene
Option → Term ?
Repetition → Term +
Kleene → Term *
```

Figure 4.1: A context-free grammar for extended rules.

```
Production → Symbol → Alternation
Alternation → (Concatenation | ) * Concatenation
Concatenation → Atom *
Atom → Symbol | ( Alternation ) | Option | Repetition | Kleene
Option → Atom ?
Repetition → Atom +
Kleene → Atom *
```

Figure 4.2: An extended grammar for extended rules.

generalized parsing (as Tomita’s algorithm only came years later). Scott and Johnstone also recently extended their GLL algorithm to EBNF grammars ([54]); we will show how to do the same to our own NEGLR parser. We will build a representation of finite automata that is nearly identical to the grammar slots used in the context-free parser and show how to build binarized parse forests for representation of the results. Finally we will demonstrate that the difference between NEGLR and its variant for extended grammars, the ENEGLR parser, is only a few lines of code.

4.1 The grammar of an extended production

Regular languages are often described using *regular expressions*. REs are a terse syntax formalism for common operations on regular languages: alternation (union), concatenation and closure (natural, positive or optional). We give a context-free grammar for a possible syntax of an extended production using REs, see **Figure 4.1**. To demonstrate the relative simplicity of this method, we give the definition of the same language using the extended rules in **Figure 4.2**.

Some operators can be omitted without reducing the expressive power of regular expression. For example, the Kleene-star is the same as a repetition followed by an option ($\xi^* \equiv \xi^{+?}$) and the option is an alternation with the alternative being ε ($\xi^? \equiv (\xi \mid \varepsilon)$). Their discussion will be omitted here for brevity; the interested reader should have no problem deducing their implementation from those of others.

4.1.1 Grammar rewrites for extended rules

Any ECFG specified using REs can be transformed into common CFGs using two simple rewrite rules:

- Let $\xi_1 \mid \xi_2 \mid \dots \mid \xi_n$ be an alternation with $n \geq 2$. A new nonterminal Ξ is introduced and the productions $\Xi \rightarrow \xi_1, \Xi \rightarrow \xi_2, \dots, \Xi \rightarrow \xi_n$ are added to the grammar. The alternation is replaced with Ξ .
- Let ξ^+ be a repetition. A new nonterminal Ξ is introduced and the productions $\Xi \rightarrow \xi$ and $\Xi \rightarrow \Xi\xi$ are added to the grammar. ξ^+ is replaced with Ξ .

Because all of these transformations preserve the language generated by the grammar, extended context-free grammars are not more powerful than their non-extended variants.

4.1.2 Regular expression trees

For easier algorithmic processing of the semantics of REs, they will be represented by their abstract syntax trees (*regular expression trees*, RET) using the following kinds of nodes:

- ε nodes, which are always leaves and are labeled with ε .
- Symbol nodes, which are always leaves and are labeled with a grammar symbol.
- Alternation nodes, which have 2 or more children. They are labeled with ALT.
- Concatenation nodes, which have 2 or more children. They are labeled with CAT.
- Repetition nodes, which have exactly 1 child. They are labeled with REP.
- The unique root node, which has exactly 1 child. We will not explicitly include it in our figures.

The rules above for the structure of a regular expression tree are stricter than the syntax of a grammar. For example, a concatenation is, according to the grammar, allowed to have no children, yet we require that it has at least two. This is not a real restriction: a concatenation with no children is the same as an

ε node; one with a single child can be replaced with that child. These rules can be relaxed in an actual implementation.

A simple regular expression tree for the rule $a^+(b \mid c)$ can be seen in **Figure 4.3**.

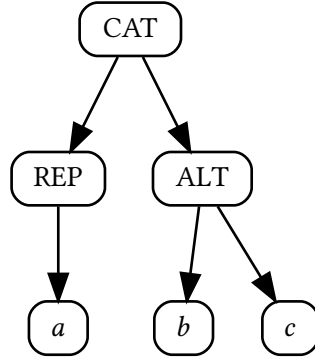


Figure 4.3: The RET tree for the rule $a^+(b \mid c)$.

4.1.3 NFA construction from a RET

There is a simple construction turning a RET into an NFA recognizing exactly the language generated by the RE. We will use the left and right ends of all tree nodes (which we will call *ports*; see **Figure 4.4** for a visual example) as the states of the NFA and define its transitions as follows⁽²⁾:

- There is an ε -transition from the left port of an ε -node to its right port.
- There is a symbol transition labeled v from the left port of a symbol node labeled v to its right port.
- There is an ε -transition from the left port of a concatenation node to the left port of its leftmost child.
- There are ε -transitions from the left port of an alternation node to the left ports of its children.
- There is an ε -transition from the left port of a repetition node to the left port of its child.
- For any non-rightmost child of a concatenation node, there is an ε -transition from the right port of a node to the left port of its right sibling.
- For any rightmost child of a concatenation node, any child of an alternation or repetition, there is an ε -transition from the right port of the node to the right port of its parent node.
- There is an ε -transition from the right port of a repetition node to its left port.
- There is an ε -transition from the left port of the root to the left port of its child and an edge from the right port of its child to its right port.

² These rules are effectively the same as those of Thompson, but restated for syntax trees.

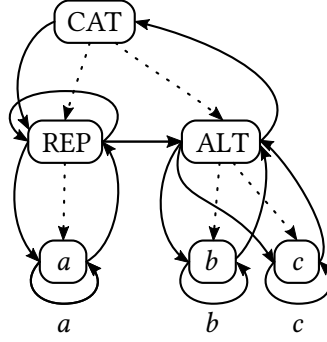


Figure 4.4: The RET for $a^+(b \mid c)$ converted to an NFA.

Figure 4.4 shows an example of an RET converted to an NFA using the above rules. The start and end states of the NFA are the left and right ports of the root node. Symbol transitions are only ever performed on symbol nodes: the number of symbol transitions in the NFA is exactly the same as the number of symbol nodes in the RET. The start state of the NFA is the left port of its root and its accepting state is the right port.

A state q of the RET-NFA is right-nullable if there is a path $p \rightsquigarrow f$ to the right root port f such that for the string γ read by the automaton on this path $\gamma \xRightarrow{*} \varepsilon$ (i.e. the automaton can finish reading the rule by at most reading nullable nonterminals).

Construction of an LR-DFA from extended rules is exactly the same as with the standard LR parser: the starting state will contain all NFA states where the left-hand side is the starting symbol and the dot is at the beginning of the rule, plus the ε -closure of these items. Transitions are created for every item of the form $X \rightarrow \alpha \cdot v\beta$ ($v \in V$) on the symbol v and the target state will contain all successor items of $X \rightarrow \alpha \cdot v\beta$, which are precomputed.

Lookahead information (if computed) is determined the same way as for context-free rules: from an item $\langle A \rightarrow \alpha \cdot X\omega, \Pi \rangle$ the lookahead set of the closure item $X \rightarrow \xi$ is given by the union of $\text{FIRST}[s]$ for any state $A \rightarrow \beta \cdot sy$ that is a direct or indirect successor of $A \rightarrow \alpha \cdot X\omega$ reachable via transitions over nullable symbols only, and also that of Π iff $A \rightarrow \alpha \cdot X\omega$ is right-nullable.

4.1.4 Canonical RET-NFA

We define the canonical form of the RET-NFA as the result of the following algorithm:

1. For each pair of distinct states p and q where $p \xrightarrow{\varepsilon} q$ add q to $\delta(p, \varepsilon)$ (transitive closure).

2. Delete all states of the automaton that are not endpoints of any non- ε -transition and are not start or accepting states.

This preserves twice the number of states as there are symbol transitions (their left and right ports) plus two (the start and accepting states). The internal states of this automaton (that are neither starting nor accepting) are paired around symbol transitions: one is always the start, the other is always the end of *exactly one* symbol transition. The former only have ingress, the latter only have egress transitions.

We can consider these internal pairs to be quasi-states⁽³⁾ of a different kind of automaton that reads a symbol upon entering a state (as opposed to traversing an edge), or terminates with failure if that's not possible, and the remaining ε -transitions between these quasi-states only determine the direction the automaton can go in. We will refer to the constituents of these quasi-states as left and right ports as we did earlier with the RET.

States with the egress symbol transition only have ingress ε -edges; those with ingress symbol transitions only have egress ε -transitions. (The start state only has egress, the accepting state only has ingress ε -transitions).

4.2 Extended context-free rules

An alternative definition of a context-free grammar that better fits our goals is as follows: a context-free grammar is a quadruple $\langle V, \Sigma, P, S \rangle$ where $P : V \rightarrow A_\Gamma$ is a function with A_Γ being the set of all finite automata accepting languages over Γ . (The rest of the symbols are as defined earlier.)

For all strings $\gamma \in \Gamma^*$ and a nonterminal $A \in V$, $A \rightarrow \gamma$ iff $\gamma \in L(P(A))$.

Expression of regular languages by their representative automata allows for an easier handling of grammars in implementations.

4.2.1 Extended grammar slots

The original algorithm encodes the evaluation of grammar rules into *slots*, a string of grammar symbols augmented with an index. Such a representation is both easy to understand and convenient to implement. Our aim is to extend that format just enough so that it can represent any finite automata.

³ We call these quasi-states because this goes against the usual, formal notion of a state. A FA reads its input in transition, not in any particular state. Yet we believe this is a useful (if a bit unusual) way to think about how these automata work.

Grammar slots can be seen as a simple representation of an execution state of a string recognizer. $AabB \bullet bBC$ refers to a state where the automaton is about to read the second b , which is different from state $Aa \bullet bBbBC$, where it is also about to read a b , but the successors of these states ($AabBb \bullet BC$ and $Aab \bullet BbBC$, respectively) are different. The state with the dot at the end (in this example, $AabBbBC \bullet$) is the accepting state where no more symbols are to be read; its only purpose is to mark the end of a successful execution.

Grammar slots only encode the states of the automaton but do not tell us anything about its transitions. For the simple recognizers used with context-free grammars this information is unnecessary as all transitions can be realized by adding one to the index stored in the grammar slot. For general FA this is insufficient.

Recall that a grammar slot is defined as a pair $\langle \sigma, i \rangle$ such that $\sigma \in \Sigma^{*(4)}$ is a string and $0 \leq i \leq |\sigma|$ is an index into, or at the end of this string.

Such a string can be constructed from the original regular expression by an operation we call *linearization*. The resulting *linear form* is the RE's projection to the alphabet Γ , i.e. the string remaining after the removal of all operators and miscellaneous characters from the RE. These strings will be the basis of the grammar slots used for extended productions.

Formally, the *linear form* of a RET node u is the string

$$\text{LINEAR-FORM}(u) = \begin{cases} \varepsilon & \text{(a)} \\ \text{LABEL}[u] & \text{(b)} \\ \text{LINEAR-FORM}(v_1) \text{ LINEAR-FORM}(v_2) \dots \text{LINEAR-FORM}(v_n) & \text{(c, d)} \\ \text{LINEAR-FORM}(v) & \text{(e)} \end{cases}$$

where (a) u is an ε -node; (b) u is a symbol node; (c) u is a concatenation node with children $v_1, v_2, \dots, v_n$; (d) u is an alternation node with children $v_1, v_2, \dots, v_n$ and (e) u is a repetition node with the child v .

The state with the dot before a symbol θ is the state where θ is about to be read. The only exception is the accepting state (with the dot at the end of the string), which only serves as a terminator and does not involve reading.

For any linear form of length n there are only $n + 1$ possible grammar slots but the canonical automaton has $n + 2$ states, thus one of those states cannot be directly encoded into a grammar slot. This is not a

⁴ Notice that in the language of the regular expressions describing the right-hand sides of the ECFG the symbols of the original grammar, whether terminal or nonterminal, are terminals.

significant issue as the list of start states is only used to bootstrap the algorithm. The transition sets of linearized automata are different for the two reading directions.

4.2.2 Encoding transitions

A linearized automaton can be expressed as a quadruple $\langle \sigma, \delta, \mathcal{T}, \mathcal{E} \rangle$ where

- $\sigma \in \Gamma^n$ is the linear form;
- $\delta \in \{0, 1\}^{n \times n}$ is the *transition matrix*;
- $\mathcal{T} \in \{0, 1\}^{n+1}$ is the *entry vector*;
- $\mathcal{E} \in \{0, 1\}^{n+1}$ is the *exit vector*.

The transition matrix is a bitmatrix (with rows and columns numbered from 0 to $n - 1$ for consistency with the rest of this thesis) where $\delta_{i,j}$ is 1 iff j is a successor of i (directed adjacency matrix); $\mathcal{T}$ is the bitvector-encoded set of state indices to which the start state has an egress ε -transition to; $\mathcal{E}$ is similarly for the ingress transitions of the accepting state.

The mirror of an automaton in this representation is the quadruple $\langle \sigma, \delta^T, \mathcal{E}, \mathcal{T} \rangle$ formed by the transpose of the transition matrix and the exchange of the entry and exit vectors.

The representation can also be written as a $(n + 1) \times (n + 1)$ matrix:

$$\begin{pmatrix} \mathcal{T} & \varepsilon \\ \delta & \mathcal{E}^T \end{pmatrix}$$

is a complete description of an automaton and its mirror is⁽⁵⁾

$$\begin{pmatrix} \mathcal{E} & \varepsilon \\ \delta^T & \mathcal{T}^T \end{pmatrix}.$$

The common bit of the $\mathcal{T}$ and $\mathcal{E}$ vectors must be the same: this bit (ε) signals whether the automaton is directly nullable.

We shall refer to a row of δ as the SUCCESSOR set of a state and to a column as the PREDECESSOR set.

⁵ We could arrange the matrix as

$$\begin{pmatrix} \delta & \mathcal{E}^T \\ \mathcal{T} & \varepsilon \end{pmatrix}$$

so that the representation of the mirror would simply be the transpose.

Consider the automaton in **Figure 4.4**. The linear form of the RET $a^+(b \mid c)$ is $\sigma = abc$ and the representative matrix is

$$\left(\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{array} \right).$$

4.2.3 Generating the linear forms

The follow algorithm creates the linear form of a RET, which consists of three parts: the linear form of the automaton as a string, the SUCCESSOR table and the set of start states. The process maintains, for each node, the set of indices (*reachable-from*) from which it can be reached. These sets of indices flow along the ε -edges of the automaton and reset when passing symbol nodes, as their right ports can only be reached from the single state that is their left port. Repetition nodes require a second pass to “push” the set down their left spine.

Most of the work happens when the algorithm encounters a symbol node. When the node is entered for the first time it is added to the linear form and, if it’s a start state, to $\mathcal{S}$. The SUCCESSOR table is built in a “backwards” fashion in the sense that an edge is added to the source when its target is encountered.

Concatenation nodes pass data down their children in sequence; alternation nodes do it in parallel and collect their results in the form of a union. Repetition nodes enter the recursion a second time to pass the reachability data down their left spine.

4.2.4 Linear forms and the canonical RET-NFA

If $\overrightarrow{\alpha \bullet \theta \beta}$ is a grammar slot with the dot before some symbol $\theta \in \Gamma$, and this instance of θ corresponds to the symbol in the quasi-state q in the canonical RET-NFA, then the state represented by $\overrightarrow{\alpha \bullet \theta \beta}$ is equivalent⁽⁶⁾ to the quasi-state q . Also, state q of the mirrored automaton is written as $\overleftarrow{\alpha \theta \bullet \beta}$.

Suppose that some string α caused the LTR automaton to reach some state $\overrightarrow{\psi \bullet \theta \xi}$ while the RTL automaton has read ω and is in state $\overleftarrow{\psi \theta \bullet \xi}$. Then $\alpha \theta \omega$ is a word in the common language accepted by both automata.

⁶ Equivalent in the usual sense: exactly the same set of strings cause the automata, when starting from this state, to accept.

LINEARIZE(*root*)

- 1: **let** *linear-form* $\leftarrow []$
- 2: **let** $\mathcal{S} \leftarrow \emptyset$
- 3: **let** $\mathcal{E} \leftarrow \text{LINEARIZE-CORE}(\text{CHILD}[\text{root}], 0, \{-1\})$

LINEARIZE-CORE(*node*, *index*, *reachable-from*)

- 1: **if** *node* is an empty node **then**
 - 2: **return** *reachable-from*
 - 3: **else if** *node* is a symbol node **then**
 - 4: **if** *index* = $|\text{linear-form}|$ **then**
 - 5: **add** LABEL[*node*] **to** *linear-form*
 - 6: **if** *reachable-from* $\ni -1$ **then**
 - 7: **add** *index* **to** $\mathcal{S}$
 - 8: **for each** *pred* **in** *reachable-from* **do**
 - 9: **if** *pred* $\neq -1$ **then**
 - 10: **add** *index* **to** SUCCESSORS[*pred*]
 - 11: **return** {*index*}
 - 12: **else if** *node* is a concatenation node **then**
 - 13: **for each** *child* **in** CHILDREN[*node*] **do**
 - 14: *reachable-from* $\leftarrow \text{LINEARIZE-CORE}(\text{child}, |\text{linear-form}|, \text{reachable-from})$
 - 15: **return** *reachable-from*
 - 16: **else if** *node* is an alternation node **then**
 - 17: **let** *result* $\leftarrow \emptyset$
 - 18: **for each** *child* **in** CHILDREN[*node*] **do**
 - 19: **add** LINEARIZE-CORE(*child*, $|\text{linear-form}|$, *reachable-from*) **to** *result*
 - 20: **return** *result*
 - 21: **else if** *node* is a repetition node **then**
 - 22: *reachable-from* $\leftarrow \text{LINEARIZE-CORE}(\text{CHILD}[\text{node}], |\text{linear-form}|, \text{reachable-from})$
 - 23: LINEARIZE-CORE(CHILD[*node*], *index*, *reachable-from*)
 - 24: **return** *reachable-from*
-

Algorithm 4: Generating the linear form of a regular expression tree.

To see why this is true, consider the quasi-state q in the RET that has the symbol θ labeling it. State $\overrightarrow{\psi \cdot \theta \xi}$ of the LTR automaton is equivalent to the left port of q and represents some prefix of a word in the language of the automaton. At the same time, state $\overleftarrow{\psi \theta \cdot \xi}$ of the RTL automaton is equivalent to the right port of q and stands for some suffix. As the transition from the left port to the right causes the automaton to read s , it follows that $\alpha\theta\omega$ is a string accepted by the automaton.

4.3 Representing derivations

In [54] Scott and Johnstone present a format for the representation of extended derivations. **Figure 4.5** shows the example of their SPPF for the nulled rule $S \rightarrow S^{?+}$. (The authors use EBNF for the description of the grammar and write the rule as $S := <[S]>.$)

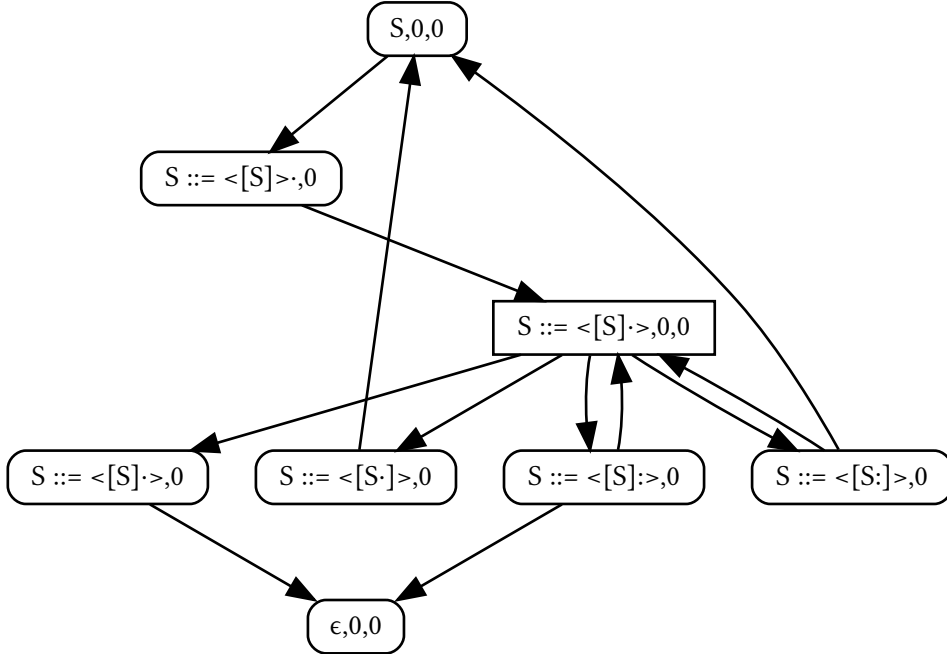


Figure 4.5: The nulled SPPF for $S \rightarrow S^{?+}$ by Scott and Johnstone.

We will use a simpler form of binarized SPPFs instead that allows for a much simpler encoding of derivations, as demonstrated by **Figure 4.6**. The nodes are labeled using grammar slots. As the linear form of $S \rightarrow S^{?+}$ has only 2 states, the SPPF also has only 2 nodes.

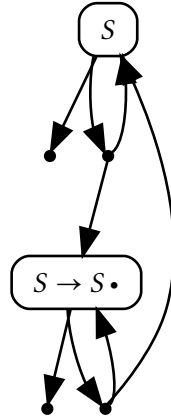


Figure 4.6: The nulled SPPF for $S \rightarrow S^{2+}$ using binarized SPPFs.

4.3.1 Parse trees for right-nulled parts

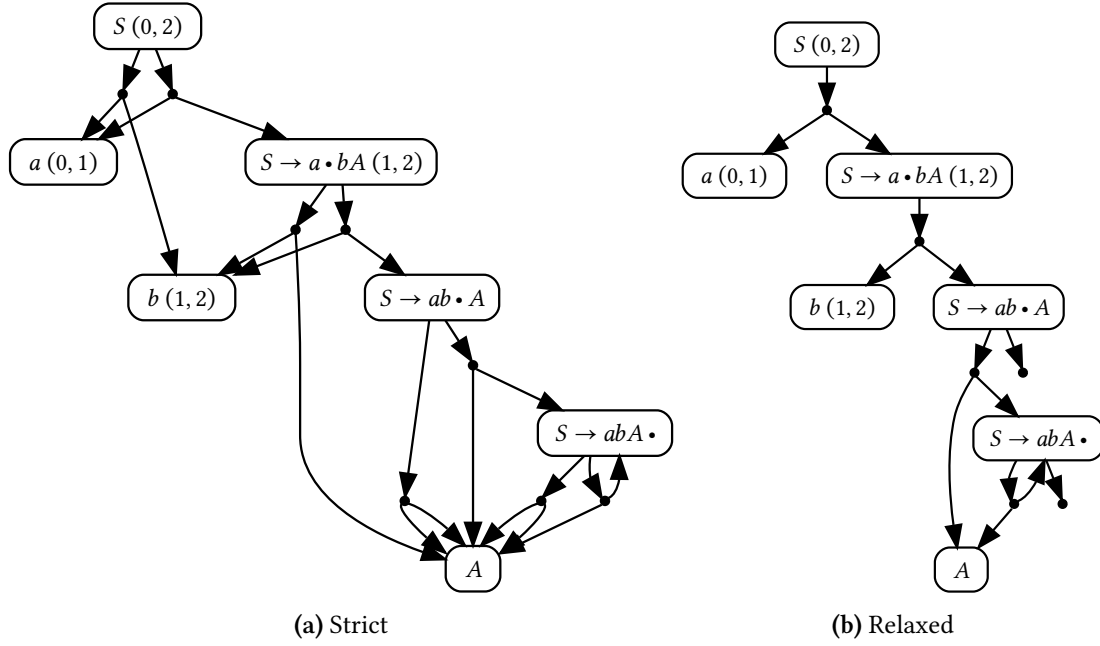
In the SPPF construction scheme of Scott and Johnstone introduced for the BRNGLR parser *intermediate* nodes representing partially-complete reductions are inserted into the representation to break up long strings into pieces of at most 2. This is to reduce the number of possible derivations of highly ambiguous grammar to order $O(n^3)$ down from $O(n^{k+1})$ with k being the length of the right-hand side. Here this is no longer only an optimization, as in an extended grammar the maximum length of the right-hand side of any rule is potentially unbounded⁽⁷⁾.

With the trees constructed by the BRNGLR parser (and also used by our NEGLR without any modification) derivations of at most 2 were directly attached to the nonterminal node at the root of the subtree being constructed and only those with a length of 3 or more were split up using intermediate nodes. To minimize the number of such nodes the following rules were introduced that guarantee intermediate nodes will only ever have families of exactly 2 children:

- Nodes labeled $X \rightarrow \overleftarrow{\alpha} \bullet$ are replaced by the nonterminal node for their left-hand side;
- Nodes labeled $X \rightarrow \overleftarrow{\alpha} \bullet$ are completely eliminated;
- Nodes labeled $X \rightarrow \overleftarrow{\beta} \bullet \theta$ are replaced by their single child θ .

These rules aren't *required* for correctness, they are only to ensure the required branching factor at all times. This is not always desirable for extended rules as the states for which the intermediate nodes were eliminated get revisited anyway. It is simpler, if a little less memory-efficient, to relax the last two rules and require that intermediate nodes are created for *all* states of the RET automaton.

⁷ For any input of finite length unboundedness can only occur as a result of nulled cycles; we could also choose to represent those with a special new kind of node instead of binarizing.

Figure 4.7: Binarized SPPF for the rule $S \rightarrow abA^*$.

Observe, for example, how an SPPF for the rule $S \rightarrow abA^*$, $A \rightarrow \epsilon$ looks with and without the rules applied in Figure 4.7. The tree built by the strict construction is in figure Figure 4.7a and that built after relaxing the rules is in Figure 4.7b. Not only the latter is simpler (both simpler to understand and simpler to store) but it is also significantly easier to construct, even by hand.

4.3.2 Context sharing

In a derivation tree for a context-free grammar two partial derivations can be shared iff they share a common prefix, i.e. a node labeled $S \rightarrow \alpha \cdot \dots$ contains the possible suffixes of all derivations that start with α . If a node is labeled $S \rightarrow \alpha \cdot \dots$ and it derives symbol sequences $\beta_1, \beta_2, \dots$ then $S \rightarrow \alpha\beta_1, S \rightarrow \alpha\beta_2, \dots$ are productions in the grammar.

A prefix of a rule used to be the sequence of symbols before the dot, which is easy to hash so that a lookup for an intermediate node for the same prefix can be fast. Two different sequences of symbols are never equivalent, as that would allow mixing prefixes and suffixes of different rules in a derivation, leading to spurious derivations.

As the right-hand sides of all context-free rules were simple string recognizers the equivalence of whether two states represent the same prefix was a simple question of whether the parts of the gram-

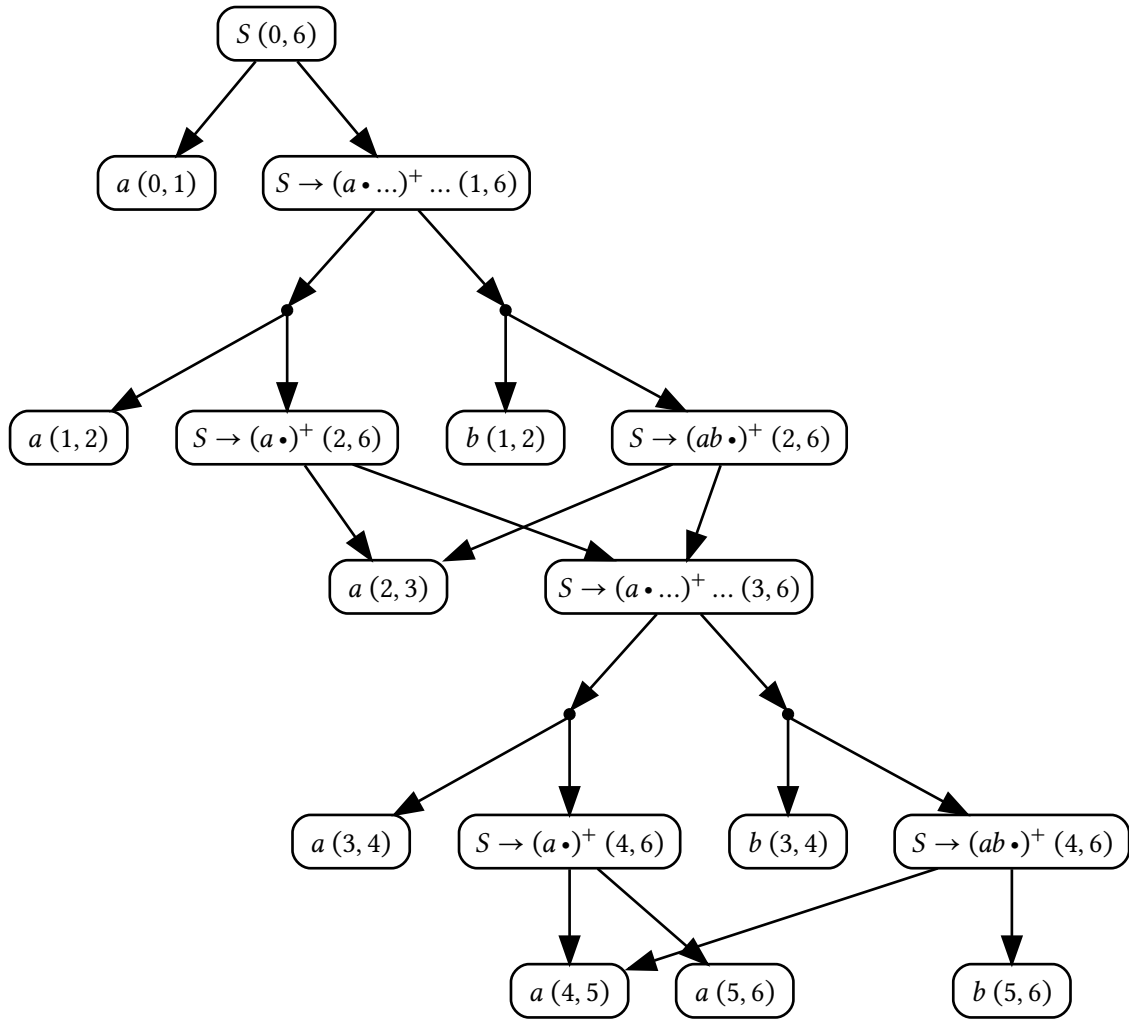


Figure 4.8: An SPPF containing a spurious derivation caused by incorrect left context sharing.

mar slots before the dot were the same. Nodes $S \rightarrow \alpha \bullet \beta_1$ and $S \rightarrow \alpha \bullet \beta_2$ could always be merged, whatever β_1 and β_2 were (provided the extents also matched).

Regular expressions complicate this by allowing similar-looking prefixes to be completely different and different-looking prefixes to be the same. The example in **Figure 4.8** exhibits one of these problems: even though the prefixes of $(a \bullet)^+$ and $(ab \bullet)^+$ are indistinguishable in their linear forms (both are $a \bullet \dots$), they are not equivalent, as one can accept the string *aaa* while the other accepts *aba*. This is reflected in the difference of the associated **PREDECESSOR** sets.

This problem is avoidable if different prefixes are never merged, equivalent or not.

On the other hand, not merging equivalent states creates repeated derivations. For example, given the grammar rule $A \rightarrow abc \mid abc$, the states $a \bullet bcabc$ and $abca \bullet bc$ are equivalent as they both correspond to

the single-character prefix a . The equivalence of $(a^2b)^*$ and $(ab \mid b^2)^+$ is probably not even immediately apparent.

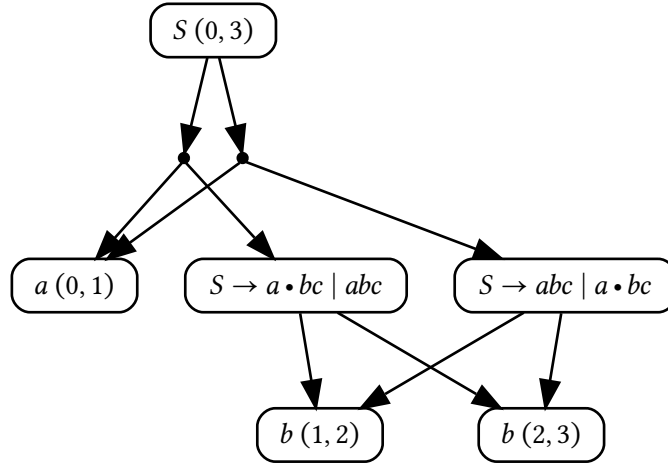


Figure 4.9: An SPPF with a repeated derivation.

To avoid introducing repeated derivations into the SPPF all states of the tracing NFA have to be sorted into equivalence classes and exactly those in the same class can be merged. The problem is solvable (for example by the algorithm in 1.2.2) but NP-complete for most nondeterministic automata.

Implementation-wise it is recommended that each state is assigned a number, globally unique for each equivalence class for $O(1)$ lookup of the associated SPPF node.

4.3.3 The order of the resulting tree

The maximum possible number of nodes in the SPPF is still at worst cubic on an input of length n :

- A terminal can be matched for any pair of input positions, the number of terminal nodes is at most $O(|\Sigma| \times n^2)$.
- Every nonterminal can represent every segment of the input, giving $O(|V| \times n^2)$ nodes.
- Intermediate nodes can be created for every pair of input positions and grammar slot, of which there are as many as $O(|P| \times k)$ where k is the most number of symbols in the right-hand side of any rule, a total of $O(|P| \times n^2 \times k)$.
- A terminal node has no derivations.
- A nonterminal or intermediate node can have derivations of length of at most 2 with $O(n)$ possibilities for splitting between the left and right children. The left node is always a symbol node, of

which there are $O((|V| + |\Sigma|) \times n)$ many, with the right possibly being a symbol ($O(|V| + |\Sigma|)$) or an intermediate node ($O(|P| \times k)$ possibilities, as the extents are determined by those of the left child and the parent).

As all of $|V|$, $|\Sigma|$, $|P|$ and k are properties of the grammar and therefore independent of the length of the input, the order of the tree is dominated by the number of derivations, but is still $O(n^3)$, if with a very high constant.

4.4 Right-nulled parse tables for extended grammars

Just like the context-free NEGLR, the extended algorithm can be used with non-right-nulled parse tables. Right-nulling, however, simplifies the implementation and should be applied.

Determination of the right-nullability of a production in a context-free grammar is a simple right-to-left linear search for the first non-nullable symbol in the right-hand side of the production. Also, if a production is right-nullable from some index i , then it is also right-nullable from any index $j > i$.

This simple property does not carry over to extended representations. Given $ABC \mid bc$, with all nonterminals being nullable and using the linearization scheme described in 4.2.1, the rule is nullable from position 0 but not from 3. When creating the right-nulled tables, one cannot just loop until the first non-nullable symbol is found, but the NFA has to be simulated in a right-to-left direction to find all indices from which the right end of the rule can be reached via nullable symbol and ϵ -transitions only. Additionally, the parse trees for both the skipped parts of the rules and the nullable nonterminals also have to be built.

4.4.1 Parse table generation

In the NELR automaton exactly the right-nullable kernel items of a state were reduction items. This simplification was possible because both the LTR successor and the reduction item had the dot moving one position forward (for an item $\alpha \bullet x\omega$ both were $\alpha x \bullet \omega$), and the distinction between successor and reduction items was not necessary. This, however, does not generalize well. Consider the right-hand side of a rule $ab \mid cd$. The sole successor of the state $\overrightarrow{a \bullet bcd}$ is $\overrightarrow{abcd \bullet}$, reached via a transition on b , but the matching *reduction* is $\overleftarrow{ab \bullet cd}$.

Let q be a state of the LR-DFA containing an item $X \rightarrow \overleftarrow{\alpha} \bullet x \beta$, where $x \in \Gamma$ is a symbol that is not required nullable. Let δ be the transition function of the LR-DFA. $X \rightarrow \overleftarrow{\alpha} x \bullet \beta$ is a reduction in $\delta(q, x)$ iff $\overleftarrow{\alpha} x \bullet \beta$ is right-nullable. The LTR grammar slots in $\delta(q, x)$ based on the SUCCESSOR table are *not related* to the reductions.

```

CREATE-PARSE-TABLE(automaton)
1: BUILD-NULLED-TREES()
2: for each lr-state in automaton do
3:   for each item in state do
4:     let  $\langle \text{state}, \text{lookaheads} \rangle \leftarrow \text{item}$ 
5:     if HAS-NULLED-TREE(state) then
6:       let tree  $\leftarrow$  NULLED-TREE[state]
7:       let target  $\leftarrow$  TARGET[TRANSITIONS[lr-state, SYMBOL[state]]]
8:       CREATE-REDUCTION(target, state, tree)
9:     for each transition in TRANSITIONS[lr-state] do
10:      CREATE-SHIFT(lr-state, SYMBOL[transition], TARGET[transition])

BUILD-NULLED-TREES()
1: let pending-states  $\leftarrow \emptyset$ 
2: for each production in P do
3:   for each state in PREDECESSORS[END[production]] do
4:     let tree  $\leftarrow$  NULLED-TREE[state]
5:     ADD-CHILDREN(tree)
6:     add state to pending-states
7:   while |pending-states| > 0 do
8:     let state  $\leftarrow$  POP(pending-states)
9:     if state = START[production] then
10:      continue
11:     if NULLABLE[SYMBOL[state]] then
12:       let this-tree  $\leftarrow$  NULLED-TREE[state]
13:       let left-tree  $\leftarrow$  NULLED-TREE[SYMBOL[state]]
14:       for each predecessor in PREDECESSORS[state] do
15:         let parent-node  $\leftarrow$  NULLED-TREE[predecessor]
16:         if ADD-CHILDREN(parent-node, left-tree, this-tree) then
17:           add predecessor to pending-states

```

Algorithm 5: Parse table generation.

Algorithm 5 generates the nulled nonterminal and intermediate SPPF nodes exactly for the nullable nonterminals and right-nullable automaton states. This also implicitly answers the question of whether a state is right-nullable: it is iff an SPPF node was created for it.

4.5 The ENEGLR parser

The challenge of modifying the NEGLR parser to deal with extended grammars lies more in the changes required in the supporting algorithms rather than those of the parser itself. In the main ENEGLR and CONTINUE-REDUCTION procedures we need to check whenever an edge is traversed that it's indeed labeled with the correct symbol (as the egress edges of a GSS node are no longer required to share the same label); in the procedures START-REDUCTION and CONTINUE-REDUCTION an extra loop must be introduced so that the tracing algorithm can discover all predecessors of a state. As parse tables no longer contain **null** as an entry⁽⁸⁾ START-REDUCTION has become simpler.

4.6 Overview

Much of the extra cost of parsing ECFGs is paid not during runtime but as higher preprocessing costs. The following list presents an overview of the major steps required to implement the ENEGLR parser. Those that are new to the extended variant are marked with (*).

1. Preprocess the grammar to determine the nullability of its nonterminals. Also calculate the FIRST sets if the algorithm is to use lookaheads.
2. Create the linear forms of the regular expressions and populate the SUCCESSOR/PREDECESSOR table. (*)
3. Create the NELR-DFA as usual, but instead of always adding one to the current index, use the SUCCESSOR table to find successor items.
4. Determine, using the PREDECESSOR table, the equivalence of the automaton's states. All productions for any given nonterminal must be considered if multiplicity is allowed. Assign every group of equivalent states a unique number and store this, for every state, in a table with constant-time lookup. (*)
5. Create the nulled parse trees for nullable nonterminals and suffixes.
6. Create the parse tables. If an NELR-DFA state has an item $\overrightarrow{X \rightarrow \alpha \bullet x \omega}$, then $\overleftarrow{X \rightarrow \alpha x \bullet \omega}$ is a reduction item in the target state of the transition on x . (A state is right-nullable iff a nulled node was created for it in the previous step.)
7. Run the parser. To find the predecessor of a grammar slot, use the PREDECESSOR table. It is no longer the case that in the GSS all egress edges of a node have the same label.

⁸ See 4.3.1 for an explanation.

```
ENEGLR(start-symbol, start-position)
1: if NULLABLE[start-symbol] then
2:   yield NULLED-TREE[start-symbol]
3: CREATE-STATE(0, start-position)
4: while there exists an unprocessed generation in the GSS do
5:   let current-position  $\leftarrow$  MIN(position of any unprocessed generation)
6:   let current-generation  $\leftarrow$  GENERATION[current-position]
7:   let processed-reductions  $\leftarrow$   $\emptyset$ 
8:   let pending-reductions  $\leftarrow$   $\emptyset$ 
9:   for each node in current-generation do
10:    for each reduction in VIABLE-REDUCTIONS(node, current-position) do
11:      for each edge in node do
12:        if SYMBOL[edge] = SYMBOL[reduction] then
13:          START-REDUCTION(reduction, edge)
14:      while |pending-reductions| > 0 do
15:        let reduction  $\leftarrow$  POP(pending-reductions)
16:        CONTINUE-REDUCTION(reduction)
17:      for each node in current-generation do
18:        SHIFTER(node)
19:      if EXISTS-NONTERMINAL-NODE(start-symbol, start-position, current-position) then
20:        yield GET-NONTERMINAL-NODE(start-symbol, start-position, current-position)

START-REDUCTION(reduction, edge)
1: for each predecessor in PREDECESSORS[reduction] do
2:   let tree  $\leftarrow$  GET-TREE(predecessor, POSITION[TARGET[edge]])
3:   ADD-CHILDREN(tree, TREE[edge], NULLED-PART[reduction])
4:   ADD-PENDING-REDUCTION(predecessor, TARGET[edge], tree)

CONTINUE-REDUCTION(reduction)
1: if NULLABLE[SYMBOL[reduction]] then
2:   EXTEND-REDUCTION(reduction, TOP[reduction], NULLED-TREE[SYMBOL[reduction]])
3: for each edge in EDGES[TOP[reduction]] do
4:   if SYMBOL[edge] = SYMBOL[reduction] then
5:     EXTEND-REDUCTION(reduction, TARGET[edge], TREE[edge])

EXTEND-REDUCTION(reduction, node, left-tree)
1: for each predecessor in PREDECESSORS[reduction] do
2:   let tree  $\leftarrow$  GET-TREE(predecessor, POSITION[node])
3:   ADD-CHILDREN(tree, left-tree, TREE[reduction])
4:   ADD-PENDING-REDUCTION(predecessor, node, tree)
```

Algorithm 6: The ENEGLR parser.

```

ADD-PENDING-REDUCTION(predecessor, node, tree)
1: if predecessor = START[PRODUCTION[predecessor]] then
2:   FINISH-REDUCTION(PRODUCTION[predecessor], node)
3: else if processed-reductions does not contain  $\langle$ predecessor, node $\rangle$  then
4:   add  $\langle$ predecessor, node $\rangle$  to processed-reductions
5:   add  $\langle$ predecessor, node, tree $\rangle$  to pending-reductions

```

Algorithm 6: The ENEGLR parser. (cont.)

Chapter 5

Further research

5.1 Reducing branching in SPPFs

SPPFs are binarized to limit the data structure to at most cubic order for highly ambiguous grammars. The binarization, however, is done eagerly: a derivation is binarized at every single opportunity, even if it only increases the size and complexity of the SPPF.

Take, for example, the grammar $S \rightarrow abc(d \mid e)$ and the ambiguous input that can be $abcd$ or $abce$ (from 2.5.2) and the binarized SPPF representation of the parse in Figure 2.3. In this case, the parse tree could be simpler, as demonstrated in Figure 5.1.

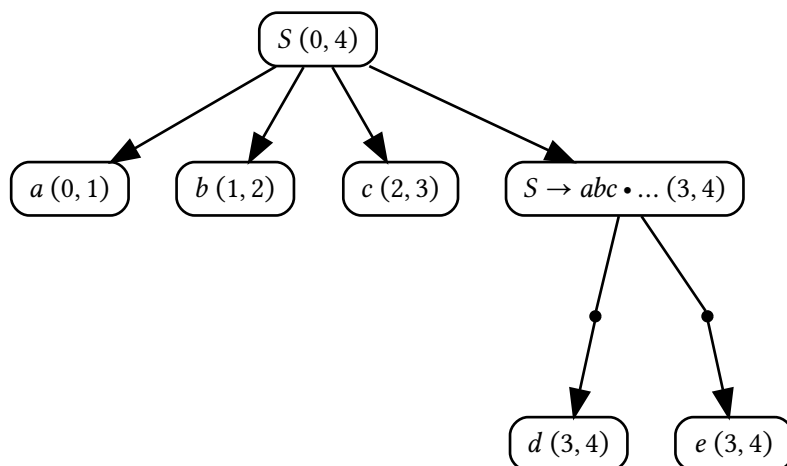


Figure 5.1: Debinarized version of the SPPF in Figure 2.3.

Instead of full binarization, subtree sharing could be used only when adding a derivation to a nonterminal node would duplicate the left context of some already found derivation. This would allow one to use the simple, linear representation of derivations when there is nothing share, gracefully degrading back to the binarized format when needed.

5.2 Error recovery

Probably the hardest problem faced by parsers is that of errors. Users do not just want applications to tell them their input is *wrong*; they also expect a clear and concise explanation of *why* it is wrong and *how* to fix it.

There is no silver bullet to error recovery, or even reporting, in parsers. Various methods, including the ad-hoc insertion/removal of tokens, marking some part of the input as garbage etc. have been proposed and developed with various rates of success. Because there's no mathematics to properly deduce user intent, solutions will always necessarily be based on trial and error.

It also doesn't help that errors are not always localized; sometimes they are only discovered later, possibly only at the end of the file.

Context-sensitive tokenization has the advantage that it can, on request, generate fake tokens that can help the parser get back on the right track. This does not help with extraneous tokens in the input, but in error recovery mode the tokenizer could simply ignore some characters at the site of an error and try matching the expected terminals at some later point in the input.

5.3 Combining parsers

The discussion of pathologically high levels of ambiguity is mostly of academic interest: most practical grammars and inputs are reasonably deterministic and can be parsed by simpler algorithms, for example by a commonplace LALR(1) parser or even finite automata. It is worth investigating whether nonterminals whose languages are demonstrably easier to parse should be handled by simpler but faster algorithms.

Care must be taken that lexical ambiguities (shift/shift conflicts) can cause a deterministic parser to fail even if the grammar is otherwise deterministic. In this case we can either restart parsing the sublanguage using generalized machinery, or, if possible, “upgrade” the progress already made so it does not

have to be completely redone. Such a hybrid parser could have significant runtime savings, as most of the runtime costs of GLR parsers come from having to manage the GSS. Mostly eliminating the stack graph could be a huge benefit to implementations.

5.3.1 Regular sublanguages

The majority of execution time in a GLR parser is spent by managing (building and tracing) the GSS. Regular languages are parsable using finite automata using only $O(1)$ memory. Parsers of regular languages do not need auxiliary data to track the execution state, hence they are much faster.

It is undecidable in general whether the language of a context-free grammar is regular. Certain methods, such as the pumping lemma can be used to show that a language *cannot* be regular; others, like the grammar being right-linear says that it must be, but there is no general algorithm which can make this decision for every grammar.

Non-self-embedding (NSE) grammars are a proper superset of context-free grammars that only generate regular languages and a grammar being NSE is decidable ([41] Theorem 3). Large parts of the grammars of commonplace programming languages are in fact NSE and a faster, simpler parser for the recognition of these parts can result in a significant overall speedup compared to the generic GSS-based approach. [41] contains an algorithm to determine whether a grammar is NSE or not.

5.4 Conclusion

The aim of our thesis was to overview and possibly improve upon existing LR parsing techniques. We achieved this goal in two ways:

- We gave a modified technique combining mostly the ε -GLR of Nederhof and Sarbo and the BRNGLR of Scott and Johnstone to build the automaton driving the LR parser, significantly reducing parser activity by the skipping nulled nonterminal instances in the GSS and shortcutting nulled right ends. As the automaton is not fully compatible with the existing algorithms we gave a modified procedure, the nullable-eliminated GLR (NEGLR), that works with these new automata. Being able to parse any ambiguous input by additionally incorporating the tokenization phase into parsing, we believe the result to be a highly practical solution that stands its ground anywhere a generalized parser is required.

- We also developed a variant of the NEGLR parser for extended context-free grammars. While most parser implementations accept a regular expression or EBNF description of a grammar, they do so by rewriting the grammar to a context-free form. Our solution avoids all grammar rewriting by using a NFA generated from the rules to trace reductions in the GSS. This extension, when implemented as an AOT-compiled parser, does not suffer a performance penalty compared to NEGLR.

We believe our NEGLR parser is highly practical, simple to implement and should be considered whenever a deterministic LALR(1) parser is insufficient for the problem at hand. The ENEGLR's similar performance makes it a viable choice for ECFG parsing, but for a one-time use this is partially offset by the algorithm's more complex implementation.

Bibliography

- [1] N. Chomsky, “A note on phrase structure grammars,” *Information and Control*, vol. 2, no. 4, pp. 393–395, 1959, ISSN: 0019-9958. DOI: 10.1016/S0019-9958(59)80017-6.
- [2] N. Chomsky, “On certain formal properties of grammars,” *Information and Control*, vol. 2, no. 2, pp. 137–167, 1959, ISSN: 0019-9958. DOI: 10.1016/S0019-9958(59)90362-6.
- [3] M. O. Rabin and D. Scott, “Finite automata and their decision problems,” *IBM Journal of Research and Development*, vol. 3, no. 2, pp. 114–125, Apr. 1959, ISSN: 0018-8646. DOI: 10.1147/rd.32.0114.
- [4] E. T. Irons, “A syntax directed compiler for ALGOL 60,” *Communications of the ACM*, vol. 4, no. 1, pp. 51–55, Jan. 1961, Place: New York, NY, USA, ISSN: 0001-0782. DOI: 10.1145/366062.366083.
- [5] D. E. Knuth, “On the translation of languages from left to right,” *Information and Control*, vol. 8, no. 6, pp. 607–639, Dec. 1, 1965, ISSN: 0019-9958. DOI: 10.1016/S0019-9958(65)90426-2.
- [6] D. H. Younger, “Recognition and parsing of context-free languages in time n^3 ,” *Information and Control*, vol. 10, no. 2, pp. 189–208, Feb. 1, 1967, ISSN: 0019-9958. DOI: 10.1016/S0019-9958(67)80007-X.
- [7] P. M. Lewis and R. E. Stearns, “Syntax-directed transduction,” *Journal of the ACM*, vol. 15, no. 3, pp. 465–488, Jul. 1968, Place: New York, NY, USA Publisher: Association for Computing Machinery, ISSN: 0004-5411. DOI: 10.1145/321466.321477.
- [8] S. H. Unger, “A global parser for context-free phrase structure grammars,” *Commun. ACM*, vol. 11, no. 4, pp. 240–247, Apr. 1968, ISSN: 0001-0782. DOI: 10.1145/362991.363001.
- [9] F. L. DeRemer, “Practical translators for LR(k) languages.” Ph.D. dissertation, Massachusetts Institute of Technology, 1969.
- [10] T. Kasami and K. Torii, “A syntax-analysis procedure for unambiguous context-free grammars,” *J. ACM*, vol. 16, no. 3, pp. 423–431, Jul. 1969, ISSN: 0004-5411. DOI: 10.1145/321526.321531.

- [11] A. J. Korenjak, "Efficient LR(1) processor construction," in *Proceedings of the First Annual ACM Symposium on Theory of Computing*, ser. STOC '69, New York, NY, USA: Association for Computing Machinery, 1969, pp. 191–200, ISBN: 978-1-4503-7478-1. DOI: 10.1145/800169.805433.
- [12] V. Strassen, "Gaussian elimination is not optimal," *Numerische Mathematik*, vol. 13, pp. 354–356, 4 Aug. 1, 1969. DOI: 10.1007/BF02165411.
- [13] J. Earley, "An efficient context-free parsing algorithm," *Commun. ACM*, vol. 13, no. 2, pp. 94–102, Feb. 1970, Place: New York, NY, USA, ISSN: 0001-0782. DOI: 10.1145/362007.362035.
- [14] D. J. Rosenkrantz and P. M. Lewis, "Deterministic left corner parsing," in *11th Annual Symposium on Switching and Automata Theory (swat 1970)*, Oct. 28, 1970, pp. 139–152. DOI: 10.1109/SWAT.1970.5.
- [15] J. Hopcroft, "A $n \log n$ algorithm for minimizing states in a finite automaton," in *Theory of Machines and Computations*, Z. Kohavi and A. Paz, Eds., Academic Press, 1971, pp. 189–196, ISBN: 978-0-12-417750-5. DOI: 10.1016/B978-0-12-417750-5.50022-1.
- [16] B. Lang, "Deterministic techniques for efficient non-deterministic parsers," in *Automata, Languages and Programming*, J. Loeckx, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1974, pp. 255–269, ISBN: 978-3-662-21545-6. DOI: 10.1007/978-3-662-21545-6_18.
- [17] L. G. Valiant, "General context-free recognition in less than cubic time," *Journal of Computer and System Sciences*, vol. 10, no. 2, pp. 308–315, Apr. 1, 1975, ISSN: 0022-0000. DOI: 10.1016/S0022-0000(75)80046-8.
- [18] O. L. Madsen and B. B. Kristensen, "LR-parsing of extended context free grammars," *Acta Informatica*, vol. 7, no. 1, pp. 61–73, 1976, ISSN: 1432-0525. DOI: 10.1007/BF00265221.
- [19] W. R. LaLonde, "Regular right part grammars and their parsers," *Commun. ACM*, vol. 20, no. 10, pp. 731–741, Oct. 1977, ISSN: 0001-0782. DOI: 10.1145/359842.359853.
- [20] D. Pager, "A practical general method for constructing LR(k) parsers," *Acta Informatica*, vol. 7, no. 3, pp. 249–268, Sep. 1, 1977, ISSN: 1432-0525. DOI: 10.1007/BF00290336.
- [21] A. Celentano, "An LR parsing technique for extended context-free grammars," *Computer Languages*, vol. 6, no. 2, pp. 95–107, 1981, ISSN: 0096-0551. DOI: 10.1016/0096-0551(81)90069-2.
- [22] P. W. Purdom and C. A. Brown, "Parsing extended LR(k) grammars," *Acta Informatica*, vol. 15, no. 2, pp. 115–127, 1981, ISSN: 1432-0525. DOI: 10.1007/BF00288959.
- [23] F. DeRemer and T. Pennello, "Efficient computation of LALR(1) look-ahead sets," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 4, pp. 615–649, Oct. 1982, ISSN: 0164-0925. DOI: 10.1145/69622.357187.

-
- [24] M. Tomita, *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. USA: Kluwer Academic Publishers, 1985, ISBN: 0-89838-202-5.
 - [25] E. Soisalon-Soininen and J. Tarhio, "Looping LR parsers," *Information Processing Letters*, vol. 26, no. 5, pp. 251–253, 1988, ISSN: 0020-0190. DOI: 10.1016/0020-0190(88)90149-4.
 - [26] M. E. Bermudez and K. M. Schimpf, "Practical arbitrary lookahead LR parsing," *Journal of Computer and System Sciences*, vol. 41, no. 2, pp. 230–250, 1990, ISSN: 0022-0000. DOI: 10.1016/0022-0000(90)90037-L.
 - [27] J. R. Kipps, "GLR parsing in time $O(n^3)$," in *Generalized LR Parsing*, M. Tomita, Ed. Boston, MA: Springer US, 1991, pp. 43–59, ISBN: 978-1-4615-4034-2. DOI: 10.1007/978-1-4615-4034-2_4.
 - [28] J. M. Leo, "A general context-free parsing algorithm running in linear time on every LR(k) grammar without using lookahead," *Theoretical Computer Science*, vol. 82, no. 1, pp. 165–176, May 22, 1991, ISSN: 0304-3975. DOI: 10.1016/0304-3975(91)90180-A.
 - [29] R. Nozohoor-Farshi, "GLR parsing for ϵ -grammars," in *Generalized LR Parsing*, M. Tomita, Ed., Boston, MA: Springer US, 1991, pp. 61–75, ISBN: 978-1-4615-4034-2. DOI: 10.1007/978-1-4615-4034-2_5.
 - [30] A. v. Deudekom and P. Kooiman, "Top-down non-correcting error recovery in LLgen," Vrije Universiteit, Faculteit Wiskunde en Informatica, Technical Report IR 338, Oct. 1993. [Online]. Available: https://dickgrune.com/Books/PTAPG_2nd_Edition/LowAvailability/1993,DeudekomAvan,KooimanP,Top-Down_Non-correcting_Error_Recovery_in_LLgen.pdf.
 - [31] M.-J. Nederhof, "Generalized left-corner parsing," in *Proceedings of the Sixth Conference on European Chapter of the Association for Computational Linguistics*, ser. EACL '93, Utrecht, The Netherlands: Association for Computational Linguistics, 1993, pp. 305–314, ISBN: 9054340142. DOI: 10.3115/976744.976780.
 - [32] P. McLean and R. N. Horspool, "A faster earley parser," in *Compiler Construction*, T. Gyimóthy, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 281–293, ISBN: 978-3-540-49939-8.
 - [33] M.-J. Nederhof and J. J. Sarbo, "Increasing the applicability of lr parsing," in *Recent Advances in Parsing Technology*, H. Bunt and M. Tomita, Eds., Kluwer Academic Publishers, 1996, pp. 35–57, ISBN: 0-7923-4152-X. [Online]. Available: <https://mjn.host.cs.st-andrews.ac.uk/publications/1996b.pdf>.
 - [34] E. Visser, "Scannerless generalized-LR parsing," Programming Research Group, University of Amsterdam, Amsterdam, Technical Report P9707, 1997.
 - [35] E. Visser, "Syntax definition for language prototyping," Ph.D. dissertation, University of Amsterdam, Sep. 1997.

- [36] J. Aycock, N. Horspool, J. Janoušek, and B. Melichar, “Even faster generalized LR parsing,” *Acta Informatica*, vol. 37, no. 9, pp. 633–651, Jun. 1, 2001, ISSN: 1432-0525. DOI: 10.1007/PL00013319.
- [37] J. Aycock and R. N. Horspool, “Practical Earley parsing,” *The Computer Journal*, vol. 45, no. 6, pp. 620–630, 2002, Publisher: OUP.
- [38] M. G. J. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser, “Disambiguation filters for scannerless generalized LR parsers,” in *Compiler Construction*, R. N. Horspool, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 143–158, ISBN: 978-3-540-45937-8.
- [39] B. Ford, “Packrat parsing: Simple, powerful, lazy, linear time, functional pearl,” *SIGPLAN Not.*, vol. 37, no. 9, pp. 36–47, Sep. 2002, Place: New York, NY, USA Publisher: Association for Computing Machinery, ISSN: 0362-1340. DOI: 10.1145/583852.581483.
- [40] L. Lee, “Fast context-free grammar parsing requires fast boolean matrix multiplication,” *J. ACM*, vol. 49, no. 1, pp. 1–15, Jan. 2002, ISSN: 0004-5411. DOI: 10.1145/505241.505242.
- [41] M. Anselmo, D. Giammarresi, and S. Varricchio, “Finite automata and non-self-embedding grammars,” in *Implementation and Application of Automata*, J.-M. Champarnaud and D. Maurel, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 47–56, ISBN: 978-3-540-44977-5.
- [42] A. Malcher, “Minimizing finite automata is computationally hard,” *Theoretical Computer Science*, vol. 327, no. 3, pp. 375–390, 2004, Developments in Language Theory, ISSN: 0304-3975. DOI: 10.1016/j.tcs.2004.03.070.
- [43] E. Scott and A. Johnstone, “Generalized bottom up parsers with reduced stack activity,” *The Computer Journal*, vol. 48, no. 5, pp. 565–587, Jan. 1, 2005, ISSN: 0010-4620. DOI: 10.1093/comjnl/bxh102.
- [44] A. Begel and S. L. Graham, “XGLR: An algorithm for ambiguity in programming languages,” *Science of Computer Programming*, vol. 61, no. 3, pp. 211–227, Aug. 2006, ISSN: 0167-6423. DOI: 10.1016/j.scico.2006.04.003.
- [45] E. Scott and A. Johnstone, “Right nulled GLR parsers,” *ACM Transactions on Programming Languages and Systems*, vol. 28, no. 4, pp. 577–618, Jul. 2006, Place: New York, NY, USA, ISSN: 0164-0925. DOI: 10.1145/1146809.1146810.
- [46] E. Scott, A. Johnstone, and R. Economopoulos, “BRNGLR: A cubic tomita-style GLR parsing algorithm,” *Acta Informatica*, vol. 44, no. 6, pp. 427–461, Oct. 1, 2007, ISSN: 1432-0525. DOI: 10.1007/s00236-007-0054-z.
- [47] A. Warth, J. R. Douglass, and T. Millstein, “Packrat parsers can support left recursion,” in *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ser. PEPM ’08, New York, NY, USA: Association for Computing Machinery, 2008, pp. 103–110, ISBN: 978-1-59593-977-7. DOI: 10.1145/1328408.1328424.

-
- [48] J. E. Denny and B. A. Malloy, “The IELR(1) algorithm for generating minimal LR(1) parser tables for non-LR(1) grammars with conflict resolution,” *Science of Computer Programming*, vol. 75, no. 11, pp. 943–979, 2010, Special Section on the Programming Languages Track at the 23rd ACM Symposium on Applied Computing, ISSN: 0167-6423. DOI: 10.1016/j.scico.2009.08.001.
- [49] D. Grune and C. J. H. Jacobs, *Parsing Techniques*, 2nd ed. Springer New York, 2010, 688 pp., ISBN: 978-1-4419-1901-4.
- [50] E. Scott and A. Johnstone, “GLL parsing,” *Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009)*, vol. 253, no. 7, pp. 177–189, Sep. 17, 2010, ISSN: 1571-0661. DOI: 10.1016/j.entcs.2010.08.041.
- [51] T. Parr and K. Fisher, “LL(*): The foundation of the ANTLR parser generator,” *SIGPLAN Not.*, vol. 46, no. 6, pp. 425–436, Jun. 2011, ISSN: 0362-1340. DOI: 10.1145/1993316.1993548.
- [52] D. Pager and X. Chen, “The lane table method of constructing LR(1) parsers,” 2012. [Online]. Available: https://www.researchgate.net/publication/266006409_The_Lane_Table_Method_Of_Constructing_LR1_Parsers (visited on 09/07/2020).
- [53] E. Scott and A. Johnstone, “GLL parse-tree generation,” *Special section on Language Descriptions Tools and Applications (LDTA’08 & ’09) & Special section on Software Engineering Aspects of Ubiquitous Computing and Ambient Intelligence (UCAmI 2011)*, vol. 78, no. 10, pp. 1828–1844, Oct. 1, 2013, ISSN: 0167-6423. DOI: 10.1016/j.scico.2012.03.005.
- [54] E. Scott and A. Johnstone, “GLL syntax analysers for EBNF grammars,” *Science of Computer Programming*, vol. 166, pp. 120–145, Nov. 15, 2018, ISSN: 0167-6423. DOI: 10.1016/j.scico.2018.06.001.
- [55] J. Kegler, “Marpa, a practical general parser: The recognizer,” *arXiv preprint arXiv:1910.08129*, 2019.
- [56] J. Gosling, B. Joy, G. Steele, *et al.*, “The Java® language specification, Java SE 15 edition,” Oracle America, Specification, Aug. 10, 2020. [Online]. Available: <https://docs.oracle.com/javase/specs/jls/se15/html/index.html>, Final Release.
- [57] G. Herman, “Faster general parsing through context-free memoization,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, London, UK: Association for Computing Machinery, 2020, pp. 1022–1035, ISBN: 9781450376136. DOI: 10.1145/3385412.3386032.
- [58] ISO JTC1/SC22, “Programming languages — C++,” International Organization for Standardization, International Standard 14882:2020, Dec. 2020, p. 1853, Published.
- [59] P. S. L. M. Barreto, “An efficient LALR(1) and LR(1) lookahead set algorithm.” [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.89.4595>.