Formal Languages And Compilers *Theory*

Christian Rossi

Academic Year 2023-2024

Abstract

The lectures are about those topics:

- Definition of language, theory of formal languages, language operations, regular expressions, regular languages, finite deterministic and non-deterministic automata, BMC and Berry-Sethi algorithms, properties of the families of regular languages, nested lists and regular languages.
- Context-free grammars, context-free languages, syntax trees, grammar ambiguity, grammars of regular languages, properties of the families of context-free languages, main syntactic structures and limitations of the context-free languages.
- Analysis and recognition (parsing) of phrases, parsing algorithms and automata, push down automata, deterministic languages, bottom-up and recursive top-down syntactic analysis, complexity of recognition.
- Translations: syntax-driven, direct, inverse, syntactic. Transducer automata, and syntactic analysis and translation. Definition of semantics and semantic properties. Static flow analysis of programs. Semantic translation driven by syntax, semantic functions and attribute grammars, one-pass and multiple-pass computation of the attributes.

The laboratory sessions are about those topics:

- Modelisation of the lexicon and the syntax of a simple programming language (C-like).
- Design of a compiler for translation into an intermediate executable machine language (for a register-based processor).
- Use of the automated programming tools Flex and Bison for the construction of syntax-driven lexical and syntactic analyzers and translators.

Contents

CHAPTER 1

Regular Languages

1.1 Formal language theory

A formal language consists of words formed by selecting letters from an alphabet, and these words must adhere to a defined set of rules to be considered well-structured.

Definition (*Alphabet*)**.** An alphabet, denoted by Σ, is a finite set of elements referred to as characters and represented as $\{a_1, a_2, \dots, a_k\}$.

Definition (*Alphabet cardinality*)**.** The alphabet cardinality, denoted as $|\Sigma|$, is the number of characters in a finite alphabet Σ, expressed as $|\Sigma| = k$.

Definition (*String*)**.** A string is a sequential arrangement of elements from an alphabet, potentially containing repetitions.

Example:
The alphabet $\Sigma = a, b$ comprises two distinct characters. This alphabet allows for the generation of various languages, including:

- $L_1 = \{aa, aaa\}$
- $L_2 = \{aba, aab\}$
- $L_3 = \{ab, ba, aabb, abab, \dots, aaabbb, \dots\}$

In these languages, different combinations of the alphabet's characters are used to construct words.

Definition (*Sentence*)**.** The strings of a language are called sentences or phrases.

Definition (*Language cardinality*)**.** The cardinality of a language is the total number of sentences it contains.

Example:
Consider the language $L_2 = bc, bbc$, which consists of two elements, indicating a cardinality of two.

Definition (*Number of occourences*)**.** The number of occurrences is the count of times a specific letter appears in a word.

Definition (*String length*)**.** The length of a string is the total number of elements it contains.

Definition (*String equality*)**.** Two strings are considered equal if and only if they have the same length, and their elements match from left to right, sequentially.

Example:
For the string aab, the counts of occurrences for the letters a and c are represented as follows:

$$|aab|_a = 2$$

$$|aab|_c = 0$$

The length of the string aab is calculated as:

$$|aab| = 3$$

1.2 Operations on strings

Concatenation Given two strings, $x = a_1a_2 \dots a_h$ and $y = b_1b_2 \dots b_k$, concatenation is defined as:

$$x \cdot y = a_1a_2 \dots a_hb_1b_2 \dots b_k$$

Concatenation demonstrates non-commutative and associative properties. The length of the resulting concatenated string equals the sum of the lengths of the individual strings:

$$|xy| = |x| + |y|$$

Empty string The empty string, represented as ε, functions as the neutral element for concatenation and adheres to the identity:

$$x\varepsilon = \varepsilon x = x$$

It's crucial to emphasize that the length of the empty string is zero:

$$|\varepsilon| = 0$$

Moreover, it's worth noting that the set containing this operator is not an empty set.

Substring Consider the string $x = xyv$, which can be expressed as the concatenation of three strings: x, y, and v, each of which may be empty. In this context, the strings x, y, and v are regarded as substrings of x. Additionally, a string u is defined as prefix of x, and v is recognized as a suffix of x. A substring not identical to the entire string x is referred to as a proper non-empty substring.

Reflection The reflection of a string $x = a_1 a_2 \dots a_h$ involves reversing the character order in the string, resulting in:

$$x^R = a_h a_{h-1} \dots a_1$$

The following identities are straightforward and immediate:

$$(x^R)^R = x$$

$$(xy)^R = y^R x^R$$

$$\varepsilon^R = \varepsilon$$

Repetition Repetition, denoted as the m-th power x^m of a string x, involves concatenating the string x with itself $m-1$ times. The formal definition is as follows:

$$\begin{cases} x^m = x^{m-1}x & \text{for } m > 0 \\ x^0 = \varepsilon \end{cases}$$

Operator precedence It's important to note that repetition and reflection operations have higher priority than concatenation.

1.3 Operations on languages

Typically, operations on a language are defined by applying string operations to each of its phrases.

Reflection The reflection, denoted as L^R, of a language L comprises a finite set of strings that are reversals of sentences present in L:

$$L^R = \{x | \exists y \left(y \in L \wedge x = y^R \right)\}$$

Prefix The set of prefixes of a language L is defined as:

$$\text{Prefixes}(L) = \{y | y \neq \varepsilon \wedge \exists x \exists z \left(x \in L \wedge x = yz \wedge z \neq \varepsilon \right)\}$$

A language is deemed prefix-free if none of its sentences have proper prefixes within the language itself:

$$\text{Prefixes}(L) \cap L = \varnothing$$

Example:
The language $L_1 = \{x | x = a^n b^n \wedge n \geq 1\}$ is an example of a prefix-free language. On the contrary, the language $L_2 = \{x | x = a^m b^n \wedge m > n \geq 1\}$ is not prefix-free.

Concatenation When working with languages $L^{'}$ and $L^{''}$, the concatenation operation is expressed as:

$$L^{'} L^{''} = \{xy | x \in L^{'} \wedge y \in L^{''}\}$$

Repetition The repetition of languages is defined by:

$$\begin{cases} L^m = L^{m-1}L & \text{for } m > 0 \\ L^0 = \{\varepsilon\} \end{cases}$$

The corresponding identities are:

$$\varnothing^0 = \{\varepsilon\}$$

$$L.\varnothing = \varnothing.L = \varnothing$$

$$L.\{\varepsilon\} = \{\varepsilon\}.L = L$$

The power operator offers a concise way to define the language of strings whose length does not exceed a specified integer k.

> **Example:**
> Consider the language $L = \{\varepsilon, a, b\}^k$ with $k = 3$, which can be represented as:
>
> $$L = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, \dots, bbb\}$$

Set operations Given that a language is essentially a set, it naturally supports standard set operations, encompassing union ($\cup$), intersection ($\cap$), difference ($\setminus$), inclusion ($\subseteq$), strict inclusion ($\subset$), and equality ($=$).

Universal language The universal language is defined as the collection of all strings over an alphabet Σ, irrespective of length, including zero:

$$L_{universal} = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$$

Complement The complement of a language L over an alphabet Σ, denoted as $\neg L$, is determined by the set difference:

$$\neg L = L_{universal} \setminus L$$

In simpler terms, it encompasses the strings over the alphabet Σ that do not belong to the language L. It's noteworthy that:

$$L_{universal} = \neg\varnothing$$

While the complement of a finite language is inevitably infinite, the complement of an infinite language does not necessarily result in a finite set.

Reflexive and transitive closures Consider a set A and a relation $R \subseteq A \times A$, where the pair $(a_1, a_2) \in R$ is often denoted as $a_1 R a_2$. The relation R^* is defined with the following properties:

- Reflexive property:
$$xR^*x \qquad \forall x \in A$$
- Transitive property:
$$x_1Rx_2 \wedge x_2Rx_3 \wedge \dots x_{n-1}Rx_n \implies x_1R^*x_n$$

Example:
For the given relation $R = \{(a,b),(b,c)\}$, its reflexive and transitive closure, denoted as R^*, will be:

$$R^* = \{(a,a),(b,b),(c,c),(a,b),(b,c),(a,c)\}$$

The relation R^+ is defined based on the transitive property:

$$x_1 R x_2 \wedge x_2 R x_3 \wedge \dots x_{n-1} R x_n \implies x_1 R^* x_n$$

Example:
For the given relation $R = \{(a,b),(b,c)\}$, the transitive closure is:

$$R^+ = \{(a,b),(b,c),(a,c)\}$$

Star operator The star operator, also known as the Kleene star, represents the reflexive transitive closure concerning the concatenation operation. It is defined as the union of all powers of the base language:

$$L^* = \bigcup_{h=0\dots\infty} L^h = L^0 \cup L^1 \cup L^2 \cup \dots = \varepsilon \cup L^1 \cup L^2 \cup \dots$$

Example:
Consider the language $L = \{ab, ba\}$. Applying the star operation results in the following language:

$$L^* = \{\varepsilon, ab, ba, abab, abba, baab, baba, \dots\}$$

It's noteworthy that while L is finite, L^* is infinite, illustrating the generative power of the star operation.

Every string within the star language L^* can be decomposed into substrings belonging to the base language L. Consequently, the star language L^* can be regarded as equivalent to the base language L. If the alphabet Σ is taken as the base language, then Σ^* encompasses all possible strings constructed from that alphabet, making it the universal language of alphabet Σ. It's common to express that a language L is defined over the alphabet Σ by indicating that L is a subset of Σ^*, denoted as $L \subseteq \Sigma^*$. The properties of the star operator can be summarized as follows:

- *Monotonicity*: $L \subseteq L^*$.
- *Closure by concatenation*: if $x \in L^* \wedge y \in L^*$, then $xy \in L^*$.
- *Idempotence*: $(L^*)^* = L^*$
- *Commutativity of star and reflection*: $(L^*)^R = (L^R)^*$

Additionally, if L^* is finite, then it follows that $\varnothing^* = \{\varepsilon\}$ and $\{\varepsilon\}^* = \{\varepsilon\}$.

Cross operator The cross operator, also referred to as the transitive closure under the concatenation operation, is defined as the union of all powers of the base language, excluding the first power L^0:

$$L^+ = \bigcup_{h=1\dots\infty} L^h = L^1 \cup L^2 \cup \dots$$

Example:
Let's consider the language $L = \{ab, ba\}$. The application of the cross operator yields the following language:

$$L^* = \{ab, ba, abab, abba, baab, baba, \dots\}$$

Quotient The quotient operator operates on the languages L_1 and L_2 by eliminating suffixes from L_1 that belong to L_2. It is formally defined as follows:

$$L = L_1/L_2 = \{y | \exists x \in L_1 \exists z \in L_2 (x = yz)\}$$

Example:
Let's consider the languages $L_1 = \{a^{2n}b^{2n} | n > 0\}$ and $L_2 = \{b^{2n+1} | n \geq 0\}$. The quotient language L_1/L_2 is:

$$L_1/L_2 = \{aab, aaaab, aaaabbb\}$$

The quotient language L_2/L_1 is:

$$L_2/L_1 = \varnothing$$

This is because no string in L_2 contains any string from L_1 as a suffix.

1.4 Regular expressions and languages

The regular language family, the most fundamental among formal language families, can be defined in three distinct ways: algebraically, through generative grammars, and by employing recognizer automata.

Definition (*Regular expression*)**.** A regular expression is a string denoted as r, constructed over the alphabet $\Sigma = \{a_1, a_2, \dots, a_k\}$ and featuring metasymbols: union ($\cup$), concatenation ($\cdot$), star (*), empty string (ε).

The metasymbols adhere to the following rules:

1. *Empty string*: $r = \varepsilon$.
2. *Unitary language*: $r = a$.
3. *Union of expressions*: $r = s \cup t$.
4. *Concatenation of expressions*: $r = (st)$.
5. *Iteration of an expression*: $r = s^*$.

In these rules, the symbols s and t represent regular expressions. The operator precedence is structured as follows: star has the highest precedence, followed by concatenation, and then union. In addition to these operators, derived operators are frequently employed:

- ε, defined as $\varepsilon = \varnothing^*$.
- e^+, defined as $e \cdot e^*$.

The interpretation of a regular expression r corresponds to a language L_r over the alphabet Σ, as outlined in the following table:

Expression r	**Language L_r**
$\varnothing$	$\varnothing$
ε	$\{\varepsilon\}$
$a \in \Sigma$	$\{a\}$
$s \cup t$ or $s\|t$	$L_s \cup L_t$
$s \cdot t$ or st	$L_s \cdot L_t$
s^*	L_s^*

Definition (*Regular language*)**.** A regular language can be described by a regular expression.

Example:
Consider the regular expression $e = (111)^*$, which denotes the language

$$L_e = \{\varepsilon, 111, 111111, \dots\}$$

Similarly, the regular expression $e_1 = 11(1)^*$ represents the language

$$L_{e_1} = \{11, 111, 1111, 11111, \dots\}$$

Definition (*Family of regular languages*)**.** The family of regular languages, denoted as REG, encompasses all languages that can be expressed by regular expressions.

Definition (*Family of finite languages*)**.** The family of finite languages, denoted as FIN, comprises languages with finite cardinality.

Every finite language is considered regular since it can be articulated as the union of a finite set of strings, each formed by concatenating a finite number of alphabet symbols:

$$(x_1 \cup x_2 \cup \cdots \cup x_k) = (a_{1_1}a_{1_2} \dots a_{1_n} \cup \cdots \cup a_{k_1}a_{k_2} \dots a_{k_m})$$

It is essential to recognize that the family of regular languages includes languages with infinite cardinality. Consequently, FIN $\subset$ REG.

Operations The union and repetition operators in regular expressions correspond to various choices, enabling the formulation of sub-expressions that identify specific sub-languages.

Expression r	**Choice of r**
$e_1 \cup \cdots \cup e_n$ or $e_1\|\dots\|e_n$	e_k for every $1 \leq k \leq n$
e^*	ε or e^n for every $n \geq 1$
e^+	e^n for every $n \geq 1$

When manipulating a regular expression, it is possible to derive a new expression by substituting any outermost sub-expression with another that represents a choice of it.

Definition (*Derivation*)**.** We define a derivation relationship between two regular expressions, $e^{'}$ and $e^{''}$, denoted as $e^{'} \implies e^{''}$, when these expressions can be decomposed as follows:

$$e^{'} = \alpha\beta\gamma$$

$$e^{''} = \alpha\delta\gamma$$

Here, δ represents a choice that includes β.

The derivation relation can be applied iteratively, leading to the following relations:

- *Power of n*: $\overset{n}{\implies}$ with $n \in \mathbb{N}$.
- *Transitive closure*: $\overset{*}{\implies}$ with $n \geq 0$.
- *Reflexive transitive closure*: $\overset{+}{\implies}$ with $n > 0$.

Example:
If $e_0 \overset{n}{\implies} e_n$, it implies that e_n is derived from e_0 in n steps. Similarly, $e_0 \overset{+}{\implies} e_n$ indicates that e_n is derived from e_0 in $n \geq 1$ steps. Additionally, $e_0 \overset{*}{\implies} e_n$ suggests that e_n is derived from e_0 in $n \geq 0$ steps.

Some derived regular expressions incorporate metasymbols, including operators and parentheses, while others consist solely of symbols from the alphabet Σ, also known as terminals, and the empty string ε. These expressions define the language specified by the regular expression. It's crucial to observe that in derivations, operators must be selected from the outer to the inner layers. Premature choices could eliminate valid sentences from consideration.

Definition (*Expression equivalence*)**.** Two regular expressions are considered equivalent if they define the same language.

1.4.1 Ambiguity

A regular language may yield identical phrases through various equivalent derivations, introducing ambiguity. The order of choices in these derivations can differ, leading to multiple valid outcomes. To determine expressions with multiple derivations, it's essential to establish numbered subexpressions within a regular expression. Follow these steps:

- Start with a regular expression, considering all possible parentheses.
- Derive a numbered version, represented as e_N, from the original expression, e.
- Identify all numbered subexpressions within the resulting expression.

Example:
For the regular expression $e = (a \cup (bb))^*(c^+ \cup (a \cup (bb)))$, the corresponding numbered expression is:

$$e_N = (a_1 \cup (b_2b_3))^*(c_4^+ \cup (a_5 \cup (b_6b_7)))$$

Sub-expressions can be derived by iteratively removing parentheses and union symbols.

Definition (*Ambiguous regular expression*)**.** A regular expression is termed ambiguous if its numbered version, denoted as f', contains two distinct strings, x and y, that become identical when the numbers are removed.

Example:
Consider the regular expression $e = (aa|ba)^*a|b(aa|b)^*$; its numbered version is:

$$e_N = (a_1a_2|b_3a_4)^*a_5|b_6(a_7a_8|b_9)^*$$

Deriving $b_3a_4a_5$ and $b_6a_7a_8$ from this expression results in the same string, baa. Therefore, the regular expression e is deemed ambiguous.

1.4.2 Extended regular expressions

To formulate a regular expression, the introduction of the following operators does not alter its expressive power:

- *Power*: a^h represents the repetition of a for h times, i.e., $aa \dots a$ repeated h times.
- *Repetition*: $[a]_k^n = a^k \cup a^{k+1} \cup \dots \cup a^n$.
- *Optionality*: $(\varepsilon \cup a)$ or $[a]$ denotes the choice between an empty string and the character a.
- *Ordered interval*: $(0 \dots 9)(a \dots z)(A \dots Z)$ specifies an ordered range, combining numeric digits, lowercase, and uppercase letters.
- *Intersection*: this operator proves useful in defining languages through the conjunction of conditions.
- *Complement*: $\neg L$ denotes the complement of language L.

1.4.3 Closure properties of the REG family

Definition (*Closure*)**.** Let *op* denote a unary or binary operator. A family of languages is considered closed under *op* if, and only if, applying the *op* operator to languages within that family results in a language that remains within the same family.

Property 1.4.1. The REG family exhibits closure under concatenation, union, star, intersection, and complement operators.

This signifies that regular languages can be merged and manipulated using these operators without venturing outside the confines of the REG family.

CHAPTER 2

Grammars

2.1 Context-free generative grammars

Regular expressions are highly effective in describing lists, but they have limitations when it comes to defining other commonly encountered constructs. To address this, for more versatile language definition, whether regular or not, we turn to the formal framework of generative grammars. Grammars offer a robust method for language definition through rewriting rules.

Context-free grammar A context-free grammar G is characterized by four components:

1. V: the nonterminal alphabet, representing the set of nonterminal symbols.
2. Σ: the terminal alphabet, comprising the symbols from which phrases or sentences are constructed.
3. P: the set of rules or productions.
4. $S \in V$: the specific nonterminal, referred to as the axiom (S), from which derivations commence.

A grammar rule is formally denoted as:

$$X \to \alpha$$

In this expression, $X \in V$ and $\alpha \in (V \cup \Sigma)^*$ When multiple rules involve the same nonterminal X, these rules can be concisely expressed as:

$$X \to \alpha_1|\alpha_2|\dots|\alpha_n$$

In this context, the strings $\alpha_1, \alpha_2, \dots, \alpha_n$ are termed alternatives for the nonterminal X.

Representation conventions In practical usage, various conventions are employed to distinguish between terminals and nonterminals. The following conventions are commonly adopted:

- Lowercase Latin letters $\{a, b, \dots\}$ are used for terminal characters.
- Uppercase Latin letters $\{A, B, \dots\}$ are reserved for nonterminal symbols.

- Lowercase Latin letters $\{r, s, \ldots, z\}$ represent strings over the alphabet Σ.
- Lowercase Greek letters $\{r, s, \ldots, z\}$ may be used for both terminals and nonterminals.
- The symbol σ is exclusively employed for nonterminals.

Rule classification The rules can be classified into the following types:

Type	Description	Structure
Terminal	The right part contains only terminals, or the empty string	$\rightarrow u\|\varepsilon$
Empty	The right part is empty	$\rightarrow \varepsilon$
Axiomatic	The left part is the axiom	$S \rightarrow$
Recursive	The left part occurs in the right part	$A \rightarrow \alpha A\beta$
Left-recursive	The left part is prefix of the right part	$A \rightarrow A\beta$
Right-recursive	The left part is suffix of the right part	$A \rightarrow \alpha A$
Left-right-recursive	The conjunction of the two previous cases	$A \rightarrow A\beta A$
Copy	The right part is a single nonterminal	$A \rightarrow B$
Linear	At most one nonterminal in the right part	$\rightarrow uBv\|w$
Right-linear	Linear and the nonterminal is a suffix	$\rightarrow uB\|w$
Left-linear	Linear and the nonterminal is a prefix	$\rightarrow Bv\|w$
Homogeneous normal	It has n nonterminals or just one terminal	$\rightarrow A_1 \ldots A_n\|a$
Chomsky normal	It has two nonterminals or just one terminal	$\rightarrow BC\|a$
Greibach normal	It has one terminal possibly followed by nonterminals	$\rightarrow a\sigma\|b$
Operator normal	The strings does not have adjacent nonterminals	$\rightarrow AaB$

2.2 Derivation and language generation

Derivation Given $\beta, \gamma \in (V \cup \Sigma)^*$, we state that β *derives* γ within a grammar G, denoted as $\beta \underset{G}{\Longrightarrow} \gamma$ or $\beta \Longrightarrow \gamma$, if and only if the following conditions are met:

- $\beta = \delta A\eta$.
- There exists a rule $A \rightarrow a$ in the grammar G.
- $\gamma = \delta\alpha\eta$

Closure We can establish the following closure properties:

- *Power*: $\beta_0 \overset{n}{\Longrightarrow} \beta_n$.
- *Reflexive*: $\beta_0 \overset{*}{\Longrightarrow} \beta_n$.
- *Transitive*: $\beta_0 \overset{+}{\Longrightarrow} \beta_n$.

Definition (*String form*). If $A \overset{*}{\Longrightarrow} \alpha$, then $\alpha \in (V \cup \Sigma)$ is called string form generated by G.

Definition (*Sentential*). If $S \overset{*}{\Longrightarrow} \alpha$, then α is called sentential or phrase form.

Definition (*Phrase*). If $A \overset{*}{\Longrightarrow} s$, then $s \in \Sigma^*$ is called phrase or sentence.

Example:
Let's examine the grammar G_l responsible for generating the structure of a book. This grammar comprises a front page f and a series A of one or more chapters. Each chapter starts with a title t and contains a sequence B of one or more lines l. The corresponding grammar rules are as follows:

$$\begin{cases} S \to fA \\ A \to AtB|tB \\ B \to lB|l \end{cases}$$

In this context:

- From A, one can generate the string form $tBtB$ and the phrase $tlltl \in L_A(G_l)$.
- From S, one can generate the phrase forms $fAtlB$ and $ftBtB$.
- The language generated from B is $L_B(G_l) = l^+$.
- The language $L(G_l)$ is generated by the context-free grammar G_l, categorizing it as a context-free language.

Definition (*Context-free language*)**.** A language is deemed context-free if there exists a context-free grammar that generates it.

Definition (*Grammars equivalence*)**.** Two grammars, denoted as G and G' are equivalent if they both generate the same language.

2.3 Erroneous grammars

Clean grammar A grammar G is deemed clean if, and only if, for every nonterminal A:

- A is reachable from the axiom S, and thus contributes to the generation of the language. In other words, there exists a derivation:

$$S \overset{*}{\Longrightarrow} \alpha A \beta$$

- A is defined, meaning it generates a non-empty language:

$$L_A(G) \neq \varnothing$$

Note that the rule $L_A(G) \neq \varnothing$ also includes the case where no derivation from A terminates with a terminal string s.

The process of cleaning a grammar involves a two-step algorithm:

1. Establish the set UNDEF, comprising undefined nonterminals.
2. Identify the set of unreachable nonterminals.

2.3.1 Phase one

We define the set DEF as follows:

$$\text{DEF} := \{A | (A \to u) \in P, \text{with } u \in \Sigma^*\}$$

We initiate the process by examining the terminal rules. Then, we apply the following update iteratively until a fixed point is reached:

$$\text{DEF} := \text{DEF} \cup \{B | (B \to D_1 D_2 \dots D_n) \in P \wedge \forall i (D_i \in \text{DEF} \cup \Sigma)\}$$

During each iteration, two cases may occur:

1. New nonterminals are discovered, and they all have their right-hand side symbols defined as nonterminals or terminals.
2. No new nonterminals are found, and the algorithm terminates.

At this stage, the nonterminals in UNDEF are removed.

2.3.2 Phase two

The produce relation, denoted as A produce B, is valid only when there is a production rule $(A \to \alpha B \beta) \in P$, where $A \neq B$ and α, β can be any strings. Subsequently, it can be asserted that a nonterminal C is reachable from the initial symbol S if and only if there exists a path in the produce relation graph from S to C. Nonterminals that lack reachability from the initial symbol can be eliminated.

2.3.3 Additional requirement

In addition to the aforementioned criteria for cleanliness, a third condition is frequently imposed:

3. G must avoid circular deviations, as they are non-essential and may introduce ambiguity.

Circular derivation occurs when, given $A \overset{+}{\Longrightarrow} A$, the derivations $A \overset{+}{\Longrightarrow} x$ and $A \overset{+}{\Longrightarrow} A \overset{+}{\Longrightarrow} x$ are possible. It's important to note that even if a grammar is clean, it might still contain redundant rules that lead to ambiguity.

Example:
Instances of unclean grammars are illustrated below:

$$\begin{cases} S \to aASb \\ A \to b \end{cases} \qquad \begin{cases} S \to a \\ A \to b \end{cases} \qquad \begin{cases} S \to aASb \\ A \to S | b \end{cases}$$

In the first case, the axiomatic rule fails to generate any phrase. In the second case, A is unreachable. In the third case, the grammar exhibits circularity involving both S and A.

2.4 Recursion and language infinity

Recursive grammars play a crucial role in generating infinite languages.

Definition (*Recursive derivation*)**.** A derivation $A \overset{n}{\Longrightarrow} xAy$ is considered recursive if $n \geq 1$.

Definition (*Immediately recursive derivation*)**.** When $n = 1$, the derivation $A \overset{n}{\Longrightarrow} xAy$ is termed immediately recursive.

Definition (*Recursive nonterminal*)**.** In the derivation $A \overset{n}{\Longrightarrow} xAy$, the symbol A is referred to as a recursive nonterminal.

Definition (*Left recursive derivation*)**.** If $x = \varepsilon$, the derivation $A \overset{n}{\Longrightarrow} xAy$ is labeled as left recursive.

Definition (*Right recursive derivation*)**.** When $y = \varepsilon$, the derivation $A \overset{n}{\Longrightarrow} xAy$ is denoted as right recursive.

It is important to emphasize that a grammar can possess recursion without exhibiting circularity.

Language infinity The condition both necessary and sufficient for the language $L(G)$ to be infinite is that, assuming the grammar G is clean and devoid of circular derivations, it permits recursive derivations.

Proof. We prove the necessary condition. If no recursive derivation is possible, every derivation would have a finite length, implying that $L(G)$ would be finite. □

Proof. We prove the sufficient condition. The derivation $A \overset{n}{\Longrightarrow} xAy$ implies the derivation $A \overset{+}{\Longrightarrow} x^m A y^m$ for any $m \geq 1$ with $x, y \in \Sigma^*$ not both empty. Additionally, the cleanliness of G implies:

- $S \overset{*}{\Longrightarrow} uAv$, indicating that A is reachable from S.
- $A \overset{+}{\Longrightarrow} w$, signifying that the derivation from A terminates successfully.

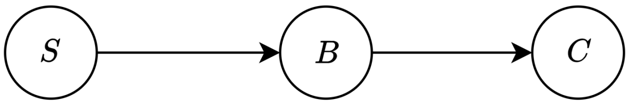
Therefore, nonterminals exist that generate an infinite language. □

Property 2.4.1. A grammar lacks recursive derivations if and only if the graph of the produce relation is acyclic.

Example:
Consider the following grammar:

$$\begin{cases} S \to aBc \\ B \to ab|Ca \\ C \to c \end{cases}$$

The corresponding graph of the produce relation is depicted below:



This acyclic graph indicates that the grammar is non-recursive.

2.5 Syntax trees and canonical derivations

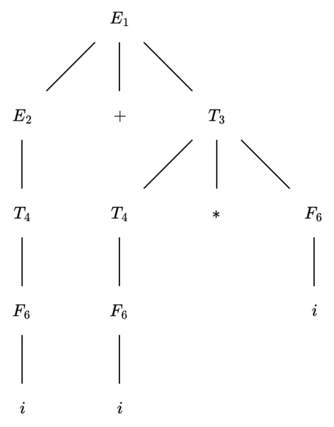
Definition (*Syntax tree*)**.** A syntax tree is a directed, ordered graph devoid of cycles. In this graph, nodes are arranged from left to right, and for any pair of nodes, there exists only one connecting path.

Key features of a syntax tree include:

- Visualization of the derivation process.
- Relationships such as parent-child, descendants, root node, and leaf or terminal nodes.
- The degree of a node is determined by the number of its children.
- The root node represents the axiom S.
- The tree's frontier contains the generated phrase.

Various subtrees can be defined from a syntax tree by selecting a node N as the new root.

Example:
The sentence $i + i * i$ can be represented in a syntax tree, following the rules for sum and product, as depicted below:



It can also be expressed in a linear form:

$$[[[[i]_F]_T]_E + [[[i]_F]_T * [i]_F]_T]_E$$

Ambiguity Right derivation (expanding the rightmost non-terminal at each step) and left derivation (expanding the leftmost non-terminal at each step) are possible. For a given syntax tree, there exists a unique right derivation and a unique left derivation corresponding to that tree. Both right and left derivations are useful for defining parsing algorithms. The ambiguity of a grammar is determined by whether a given sentence has a unique syntax tree.

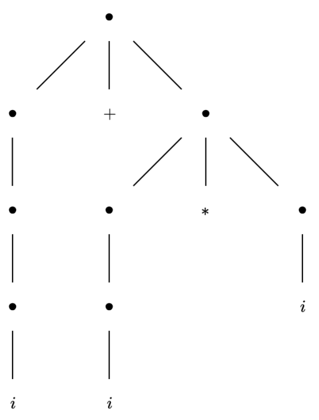
Construction To construct a correct syntax tree, it is important to consider:

- Deriving nonterminals for low-precedence operators first.
- Deriving nonterminals for high-precedence operators later.

Definition (*Skeleton tree*)**.** A skeleton tree is a syntax tree that preserves only the frontier and structure.

Definition (*Condensed skeleton tree*)**.** A condensed skeleton tree is a syntax tree in which internal nodes on non-branching paths are merged.

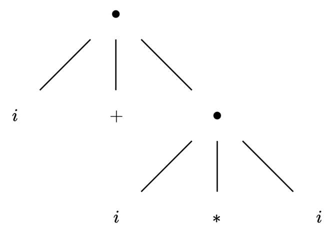
Example:
The syntax tree from the previous example can be transformed into a skeleton tree:



With the corresponding linear form:

$$[[[[i]]] + [[[i]] * [i]]]$$

It can also be transformed into a condensed skeleton tree:



With the corresponding linear form:

$$[[i] + [[i] * [i]]]$$

2.6 Parenthesis languages

Numerous constructed languages incorporate parenthesized or nested structures, which are created by matching pairs of opening and closing marks. These parentheses may exhibit nesting, allowing for the inclusion of other parenthesized structures within a pair. Additionally, nested structures can be arranged in sequences at the same level of nesting. This abstraction, independent of specific representation and content, is referred to as Dyck language.

Example:
As an illustration, consider an alphabet for a Dyck language: $\Sigma = \{'(',')','[',']'\}$. A valid sentence over this alphabet is: ()[[()[]]()].

2.7 Regular composition of context-free languages

Context-free languages exhibit closure properties under union, concatenation, and star operations. Consider two grammars, $G_1 = (\Sigma_1, V_1, P_1, S_1)$ and $G_2 = (\Sigma_2, V_2, P_2, S_2)$ defining languages L_1 and L_2, respectively. Let's assume that the sets of nonterminal symbols, V_{N_1} and V_{N_2} are disjoint and that $S \notin (V_{N_1} \cup V_{N_2})$.

Union The union of languages $L_1 \cup L_2$ is defined by a grammar that incorporates the rules of both individual grammars, along with the additional initial rule $S \to S_1|S_2$. In formulaic terms, the grammar is expressed as:

$$G = (\Sigma_1 \cup \Sigma_2, \{S\} \cup V_{N_1} \cup V_{N_2}, \{S \to S_1|S_2\} \cup P_1 \cup P_2, S)$$

Example:
The language $L = \{a^i b^j c^k | i = j \vee j = k\}$ can be defined as the union of two languages:

$$L = \{a^i b^i c^* | i \geq 0\} \cup \{a^* b^i c^i | i \geq 0\} = L_1 \cup L_2$$

The grammars for these two languages, G_1 and G_2, are defined as follows:

$$G_1 = \begin{cases} S_1 \to XC \\ X \to aXb|\varepsilon \\ C \to cC|\varepsilon \end{cases} \qquad G_2 = \begin{cases} S_2 \to AY \\ Y \to bYc|\varepsilon \\ A \to aA|\varepsilon \end{cases}$$

The union language is defined with the rule:

$$S \to S_1|S_2$$

It's important to note that the nonterminal sets of grammars G_1 and G_2 are distinct.

If the nonterminals in the grammars are not disjoint, meaning they share some common nonterminals, the resulting grammar generates a superset of the union language. This may lead to the generation of spurious additional sentences.

Concatenation The concatenation $L_1 L_2$ is defined by the grammar a grammar that incorporates the rules of both individual grammars, along with the additional initial rule $S \to S_1 S_2$. The grammar is formulated as follows:

$$G = (\Sigma_1 \cup \Sigma_2, \{S\} \cup V_{N_1} \cup V_{N_2}, \{S \to S_1 S_2\} \cup P_1 \cup P_2, S)$$

Star For the star language $(L_1)^*$, the grammar G is obtained by augmenting G_1 with the rules $S \to SS_1|\varepsilon$.

Cross The language $(L_1)^+$ is generated by the grammar G, which is derived by adding to G_1 the rules $S \to SS_1|S1$.

Mirror language The mirror language of $L(G)$, denoted as $(L(G))^R$, is generated by a mirror grammar. This grammar is obtained by reversing the right-hand side of the rules.

2.8 Ambiguity

Definition (*Ambiguous grammar*)**.** A sentence x defined by grammar G is considered ambiguous if it possesses multiple distinct syntax trees. In such instances, we characterize the grammar G as ambiguous.

Definition (*Degree of ambiguity*)**.** The degree of ambiguity of a sentence x in a language $L(G)$ is defined as the count of distinct syntax trees compatible with G. For a grammar, the degree of ambiguity is the maximum among the degrees of ambiguity for its sentences.

The problem of determining whether a grammar is ambiguous is undecidable because there exists no general algorithm that, for any context-free grammar, can guarantee termination with the correct answer in a finite number of steps. Consequently, establishing the absence of ambiguity in a specific grammar often requires a manual, case-by-case analysis through inductive reasoning, involving the examination of a finite number of cases.

To demonstrate the ambiguity of a grammar, one can provide a witness, which is an example of an ambiguous sentence generated by the grammar. Therefore, it is advisable to design grammars with the aim of being unambiguous from the outset to avoid potential issues related to ambiguity. Ambiguity can be classified into various categories, as outlined below.

2.8.1 Ambiguity from bilateral recursion

Bilateral recursion occurs when a non-terminal symbol A displays both left and right recursion.

Example:
Consider grammar G_1:

$$G_1 = E \to E + E|i$$

This grammar can generate the string $i + i + i$ in two distinct ways. Notably, the language generated by $L(G_1) = i(+i)^*$ is regular. Hence, it's possible to create simpler, unambiguous grammars, such as:

- A right-recursive grammar: $E \to i + E|i$.
- A left-recursive grammar: $E \to E + i|i$.

Let's now consider the grammar G_2:

$$G_2 = A \to aA|Ab|c$$

The language generated by G_2, $L(G_2) = a^*cb^*$, is regular. However, grammar G_2 allows derivations where the a and b characters in a sentence can be obtained in any order, making it ambiguous. To resolve this ambiguity, two nonambiguous grammars can be constructed:

1. Generate a's and b's separately using distinct rules:

$$G_2 = \begin{cases} S \to AcB \\ A \to aA|\varepsilon \\ B \to bB|\varepsilon \end{cases}$$

2. First generate the a's then the b's:

$$G_2 = \begin{cases} S \to aS|X \\ X \to Xb|c \end{cases}$$

2.8.2 Ambiguity from language union

If languages $L_1 = L(G_1)$ and $L_2 = L(G_2)$ share some sentences, constructing a grammar G for their union language introduces ambiguity. For any sentence $x \in L_1 \cap L_2$, it permits two distinct derivations: one following the rules of G_1 and the other following the rules of G_2. This ambiguity persists when utilizing a single grammar G that amalgamates all the rules. However, sentences exclusively belonging to $L_1 \setminus L_2$ and $L_2 \setminus L_1$ are nonambiguous. To resolve this ambiguity, a solution is to provide separate sets of rules for $L_1 \cap L_2$, $L_1 \backslash L_2$, and $L_2 \backslash L_1$.

2.8.3 Inherent ambiguity

A language is deemed inherently ambiguous when every grammar defining it is ambiguous.

Example:
Consider the language $L = \{a^i b^j c^k | i = j \vee j = k\} = \{a^i b^i c^* | i \geq 0\} \cup \{a^* b^i c^i | i \geq 0\}$. This language is characterized by two grammars:

$$G_1 = \begin{cases} S_1 \to XC \\ X \to aXb|\varepsilon \\ C \to cC|\varepsilon \end{cases} \qquad G_2 = \begin{cases} S_2 \to AY \\ Y \to bYc|\varepsilon \\ A \to aA|\varepsilon \end{cases}$$

The union grammar of these two grammars is ambiguous. This observation suggests the intuitive conclusion that any grammar for the language L is also ambiguous, reflecting the inherent ambiguity of the language itself.

2.8.4 Ambiguity from concatenation of languages

Ambiguity can arise in the concatenation of languages when a suffix of a sentence in the first language also acts as a prefix of a sentence in the second language. To eliminate this ambiguity, one should avoid situations where a substring from the end of a sentence in the first language seamlessly connects to the beginning of a sentence in the second language. A practical solution involves introducing a new terminal symbol, acting as a separator, which does not belong to either of the two alphabets.

Example:
Given two languages, L_1 and L_2, if concatenation introduces ambiguity, the issue can be resolved by adding a new terminal symbol, denoted as #. The axiomatic rule can then be transformed as follows:

$$S \to S_1 \# S_2$$

It is crucial to note that this modification also alters the language itself.

2.8.5 Other cases of ambiguity

There are additional, less significant instances of ambiguity, including:

- *Ambiguity in regular expressions*: Resolve this by eliminating redundant productions from the rule.
- *Lack of order in derivations*: Address this issue by introducing a new rule that enforces the desired order.

2.9 Strong and weak equivalence

Definition (*Grammars weakly equivalence*)**.** Two grammars are considered weakly equivalent if they generate the same language, expressed as:

$$L(G) = L(G')$$

It's crucial to recognize that with weak equivalence, two grammars can generate the same language while still producing different syntax trees. The structural aspect is significant, especially in the context of translators and interpreters.

Definition (*Grammars strong equivalence*)**.** Two grammars are strongly equivalent if they not only generate the same language but also produce identical condensed skeleton trees.

Consequently, it can be inferred that strong equivalence encompasses weak equivalence. Additionally, it is noteworthy that the problem of strong equivalence is decidable, whereas the problem of weak equivalence is undecidable.

2.10 Grammar normal forms and transformations

Grammars in normal forms impose constraints on the rules without diminishing the family of generated languages. They serve utility in both proving properties and language design.

2.10.1 Nonterminal expansion

The expansion of a nonterminal is employed to eliminate it from the rules where it appears.

Example:
Consider the grammar:

$$\begin{cases} A \to \alpha B \gamma \\ B \to \beta_1 | \beta_2 | \dots | \beta_n \end{cases}$$

By expanding the nonterminal B, we obtain:

$$A \to \alpha\beta_1\gamma | \alpha\beta_2\gamma | \dots | \alpha\beta_n\gamma$$

2.10.2 Elimination of the axiom from right parts

It is always possible to obtain the right part of rules as strings by introducing a new axiom S_0 and the rule $S_0 \to S$.

2.10.3 Normal form without nullable nonterminals

A non-terminal A is nullable if it can derive the empty string. Consider the set Null $\subseteq V$ of nullable non-terminals. The set is composed of the following logical clauses, applied until a fixed point is reached:

$$A \in \text{Null} \implies \begin{cases} (A \to \varepsilon) \in P \\ (A \to A_1 A_2 \dots A_n) \in P \qquad \text{with } A_i \in V \backslash \{A\} \\ \forall 1 \leq i \leq n \qquad \text{with } A_i \in \text{Null} \end{cases}$$

The construction of the non-nullable form consists of:

1. Compute the Null set.
2. For each rule within P, add as alternatives those obtained by deleting the nullable nonterminals from the right part.
3. Remove all empty rules, except for $A = S$.
4. Clean the grammar and remove any circularity.

The normal form without nullable nonterminals requires that no nonterminal other than the axiom is nullable. In that case, the axiom is nullable only if the empty string ε is in the language.

2.10.4 Copy rules and their elimination

Copy rules are utilized to factorize common parts, reducing the size of the grammar. However, eliminating copy rules shortens derivations and reduces the height of syntax trees. The typical trade-off involves defining the Copy$(A) \subseteq V$ set, representing nonterminals into which the nonterminal A can be copied, possibly transitively:

$$\text{Copy}(A) = \{B \in V | \text{there exists a derivation } A \implies B\}$$

To eliminate copy rules, the following steps are undertaken:

1. Compute Copy (assuming a grammar with non-empty rules) by applying logical clauses until a fixed point is reached. This involves determining the reflexive transitive closure of the copy relation defined by the copy rules:
$$C \in \text{Copy}(A) \text{ if } (B \in \text{Copy}(A)) \land (B \to C \in P)$$
2. Define the rules of a grammar G', equivalent to G but without copy rules. Remove the copy rules:
$$P^{'} := P \backslash \{A \to B | A, B \in V\}$$
Add compensating rules:
$$P^{'} := P^{'} \cup \{A \to \alpha | \exists B (B \in \text{Copy}(A) \land (B \to \alpha) \in P)\}$$
The set of rule may increase considerably in size.

2.10.5 Conversion of left recursion to right recursion

Grammars without left recursion are necessary for designing top-down parsers. To convert from left recursion to right recursion, there are multiple possibilities. The main case involves the conversion of immediate left recursion:

$$\begin{cases} A \to A\beta_1 | A\beta_2 | \dots | A\beta_n \\ A \to \gamma_1 | \gamma_2 | \dots | \gamma_k \end{cases}$$

Here, $\beta_i \neq \varepsilon \, \forall i$. This grammar can be transformed into:

$$\begin{cases} A \to A^{'}\gamma_1 | A^{'}\gamma_2 | \dots | A^{'}\gamma_k | \gamma_1 | \gamma_2 | \dots | \gamma_k \\ A^{'} \to A^{'}\beta_1 | A^{'}\beta_2 | \dots | A^{'}\beta_h | \beta_1 | \beta_2 | \dots | \beta_h \end{cases}$$

In this grammar, right recursion is achieved since the string is generated from the left.

2.10.6 Chomsky normal form

The Chomsky normal form consists of two types of rules:

1. *Homogeneous binary rules*: $A \to BC$ with $B, C \in V$.
2. *Terminal rules with a singleton right part*: $A \to a$ with $a \in \Sigma$.

Syntax trees of this form have internal nodes of degree two and leaf parent nodes of degree one. The procedure to obtain the Chomsky normal form from a grammar G is as follows:

- If the language contains the empty string, add the rule: $S \to \varepsilon$.
- Apply iteratively the following process:
  - For each rule type $A_0 \to A_1 A_2 \dots A_n$.
  - Add the rule type $A \to \langle A_1 \rangle \langle A_2 \dots A_n \rangle$.
  - Also, add another rule $\langle A_2 \dots A_n \rangle \to A_2 \dots A_n$.

  After some iterations, A will be terminal, resulting in $\langle A_1 \rangle \to A_1$.

2.10.7 Real-time normal form

In the real-time normal form, the right part of any rule has a terminal symbol as a prefix:

$$A \to a\alpha \text{ with } a \in \Sigma, \alpha \in \{\Sigma \cup V\}^*$$

The name of this form derives from a property of syntax analysis: every step reads and consumes one terminal symbol. With this normal form, the number of steps for the analysis is proportional to the length of the string.

2.10.8 Greibach normal form

In the Greibach normal form, every right part consists of a terminal followed by zero or more nonterminals:

$$A \to a\alpha \text{ with } a \in \Sigma, \alpha \in V^*$$

2.11 Free grammars extended with regular expressions

The EBNF class is instrumental in constructing grammars that enhance readability through the use of star, cross, and union operators. These grammars also facilitate the creation of syntax diagrams, serving as a blueprint for the syntax analyzer's flowchart. It's important to note that since the context-free family is closed under all regular operations, the generative power of EBNF is equivalent to that of BNF.

Definition (*EBNF grammar*)**.** An EBNF grammar is defined as a four-tuple $\{V, \Sigma, P, S\}$, where there are exactly $|V|$ rules in the form $A \to \eta$ with η being a regular expression over $\Sigma \cup V$.

Compared to BNF, an EBNF grammar is typically shorter and more readable. Additionally, the use of nonterminal symbols in EBNF allows for more intuitive and meaningful names.

Derivation The derivation relation in EBNF is defined by considering an equivalent BNF with infinite rules.

Definition (*Derived string*)**.** Given strings η_1 and η_2 within $(\Sigma \cup V)^*$. The string η_2 is said to be derived immediately in G from η_1, denoted as $\eta_1 \implies \eta_2$, if the two strings can be factorized as:

$$\eta_1 = \alpha A \gamma$$

$$\eta_2 = \alpha \vartheta \gamma$$

and there exists a rule:

$$A \to e$$

such that the regular expression e admits the derivation $e \overset{*}{\implies} \vartheta$.

Note that η_1 and η_2 do not incorporate operators or parentheses typically found in regular expressions. The only element that qualifies as a regular expression is the string e, and it only appears in the derivation if it is terminal. When using EBNF, the node degree is unbounded, leading to a generally wider tree with reduced depth.

2.12 Comparison of regular and context-free languages

Regular languages constitute a specific category within free languages, distinguished by stringent constraints on rule formulation. These constraints result in unavoidable repetitions within the sentences of regular languages. The transformation rules employed to convert a regular expression into a grammar generating the same regular language are as follows:

Regular expression	**Corresponding grammar**
$r = r_1 r_2 \dots r_k$	$E = E_1 E_2 \dots E_k$
$E = r_1 \cup r_2 \cup \dots \cup r_k$	$E = E_1 \cup E_2 \cup \dots \cup E_k$
$r = (r_1)^*$	$E = EE_1 \vert \varepsilon$ or $E = E_1 E \vert \varepsilon$
$r = (r_1)^+$	$E = EE_1 \vert E_1$ or $E = E_1 E \vert E_1$
$r = b \in \Sigma$	$E = b$
$r = \varepsilon$	$E = \varepsilon$

In general, regular expressions form a subset of context-free languages:

$$\text{REG} \subset \text{CF}$$

Definition (*Unilinear grammar*)**.** A grammar is unilinear if and only if its rules are either all right-linear or all left-linear.

We can impose certain constraints on a unilinear grammar:

- Strictly unilinear rules: each rule contains at most one terminal, expressed as $A \to aB$ with $A \in (\Sigma \cup \varepsilon)$ and $B \in (V \cup \varepsilon)$.
- All terminal rules are empty.

Thus, we can assume rules of the form $A \to aB|\varepsilon$ for the right case and $A \to Ba|\varepsilon$ for the left case.

Strictly unilinear grammars It is demonstrable that regular expressions can be translated into strictly unilinear grammars, establishing the regular language set as a subset of unilinear grammars:

$$\text{REG} \subseteq \text{UNILIN}$$

Conversely, from any unilinear grammar, an equivalent regular expression can be obtained:

$$\text{UNILIN} \subseteq \text{REG}$$

Consequently, it holds that

$$\text{UNILIN} = \text{REG}$$

This property allows viewing the rules of the unilinear right grammar as equations, where the unknowns represent the languages generated by each nonterminal. Let G be a strictly unilinear right grammar with all terminal rules empty. A string $x \in \Sigma^*$ is in L_A in the following cases:

1. x is the empty string: governed by a rule $P : A \to \varepsilon$.
2. $x = ay$: controlled by a rule $P : A \to aB$ and $y \in L_B$.

For every nonterminal A_0 defined by $A_0 \to a_1A_1|a_2A_2|\dots|a_kA_k|\varepsilon$ we have $L_A = a_1L_{a_1} \cup a_2L_{a_2} \cup \dots \cup a_kL_{a_k} \cup \varepsilon$. This yields a system of $n = |V|$ equations in n unknowns, solvable through the method of substitution and the application of the Arden identity.

Definition (*Arden identity*)**.** An equation $KX \cup L$, with K being a nonempty language and L any language, possesses a unique solution:

$$X = K^*L = KK^*L \cup L$$

Example:
Consider the grammar:

$$\begin{cases} S \to sS|eA \\ A \to sS|\varepsilon \end{cases}$$

Transforming this grammar into a system of equations, we get:

$$\begin{cases} L_S \to sL_S \cup eL_A \\ L_A \to sL_S \cup \varepsilon \end{cases}$$

Substituting the second equation into the first one and applying the concatenation operation, we obtain:

$$\begin{cases} L_S \to (s \cup es)L_S \cup e \\ A \to sL_S \cup \varepsilon \end{cases}$$

Applying the Arden identity, we arrive at:

$$\begin{cases} L_S \to (s \cup es)^*e \\ A \to s(s \cup es)^*e \cup \varepsilon \end{cases}$$

It is noteworthy that regular languages exhibit inevitable repetitions.

Property 2.12.1. Let G be a unilinear grammar. Every sufficiently long sentence x (i.e., longer than a grammar-dependent constant k) can be factorized as $x = tuv$ (with u non-empty), such that, for all $i \geq 1$, the string $tu^nv \in L(G)$.

In simpler terms, the sentence can be pumped by injecting string u an arbitrary number of times.

Proof. Consider a strictly right-linear grammar G with k nonterminal symbols. In the derivation of a sentence x whose length is k or more, there is necessarily a nonterminal A that appears at least two times. Consequently, it is possible to derive tv, tuv, $tuuv$, and so on. □

This property aids in determining whether a grammar generates a regular language or not. A grammar generates a regular language only if it lacks self-nested derivations. Notably, the inverse is not necessarily true: a regular language may be generated by a grammar with self-nested derivations. The absence of self-nested derivations facilitates solving language equations of unilinear grammars.

In the context-free languages, all sufficiently long sentences necessarily contain two substrings that can be repeated arbitrarily many times, giving rise to self-nested structures. This impedes the derivation of strings with three or more parts that are repeated the same number of times. Consequently, the language of three or more powers is not context-free. Therefore, the language of copies is also not context-free.

2.12.1 Closure properties

Regular languages exhibit closure under operations such as reverse, star, complement, union, and intersection. Conversely, context-free languages demonstrate closure under reverse, star, and union operators. It can be established that the intersection of a context-free language with a regular language remains within the realm of context-free languages. Introducing a filter through a regular language can enhance the selectivity of a grammar, and notably, the outcome of this filtration consistently falls within the category of context-free languages.

CHAPTER 3

Finite state automata

3.1 Introduction

We address the task of determining whether a string belongs to a specified language. Automata serve as abstract machines employed to apply a procedure for string recognition. In this context, the input domain consists of a set of strings from an alphabet Σ. The application of a recognition algorithm α to a given string x is expressed as $\alpha(x)$.

Definition (*Accepted string*)**.** A string x is deemed accepted if $\alpha(x) = \text{yes}$; otherwise, it is rejected.

Definition (*Recognized language*)**.** The language recognized, $L(\alpha)$, is the set of accepted strings:

$$L(\alpha) = \{x \in \Sigma^* | \alpha(x) = \text{yes}\}$$

In cases where the language is semidecidable, there is a possibility that the algorithm may not terminate for some incorrect string x. However, in the practical realm of language processing, concerns about decidability issues are alleviated, as the relevant language families are typically decidable.

Definition (*Computation step*)**.** In both theoretical and practical considerations of formal languages, a computation step is defined as a singular atomic operation of the automaton, capable of manipulating only one symbol at a time.

Consequently, it is customary to articulate the recognition algorithm through an automaton, be it a recognizer or a transducer machine. This convention serves two primary purposes:

1. Establishing a clear connection between different language families and their respective generative devices.

2. Deferring unnecessary and premature references to the concrete implementation of the algorithm in a specific programming language.

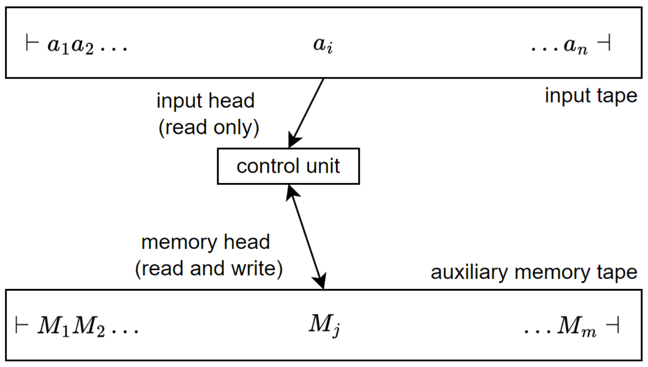


Figure 3.1: General model of a recognizer automaton

Moves The automata scrutinize the input string and carry out a sequence of moves, with each move contingent upon the symbols currently indicated by the heads and the ongoing state. These moves may have various effects, including:

- Shifting the input head one position to the left or right.
- Replacing the current memory symbol with a new one and shifting the memory head one position to the left or right.
- Changing the current state.

Depending on the type of automaton, certain characteristics can emerge:

- *One-way automaton*: the input head is capable of shifting only to the right.
- *No auxiliary memory*: represented by finite state automata, this machine model recognizes regular languages.
- *Auxiliary memory*: illustrated by pushdown automata, this machine model recognizes free languages.

Definition (*Automaton configuration*)**.** A configuration comprises three components determining the automaton's behavior: the unread segment of the input tape, the contents and position of the memory tape and head, and the current state of the control unit.

In the initial configuration, the input head is positioned on the symbol immediately following the start-marker, the control unit is in a specific state (initial state), and the memory tape contains only a special initial symbol. Transitions, driven by moves, prompt changes in the automaton configuration. The entire sequence of transitions constitutes the computation of the automaton.

Definition (*Deterministic automaton*)**.** An automaton displays deterministic behavior if, in every instantaneous configuration, at most one move is possible. Otherwise, the automaton is termed non-deterministic.

In the final configuration, the control unit occupies a designated final state, and the input head is positioned on the end-marker of the string to be recognized. Sometimes, the final configuration is characterized by a condition for the memory tape: either empty or containing only one special final symbol.

Definition (*Acceptance condition*)**.** A source string x is accepted by the automaton when it initiates from the initial input configuration $\vdash x \dashv$, undergoes a series of transitions, and arrives at a final configuration.

The collection of all strings accepted by the automaton forms the language recognized by it. In the case of non-deterministic automata, it might achieve the same final configuration through different sequences of transitions, or it could even reach multiple distinct final configurations. he computation concludes either when the automaton reaches a final configuration or when it cannot perform any further transition steps because there are no possible moves left in the current instantaneous configuration. In the former scenario, the input string is accepted; in the latter, it is rejected.

Definition (*Equivalence*)**.** Two automata that recognize the same language are termed equivalent.

Regular languages, recognized by finite state automata, constitute a subset of languages recognizable in real time by a Turing machine. Conversely, context-free languages form a subset of languages recognized by a Turing machine with polynomial time complexity. Numerous applications in computer science and engineering leverage finite state automata, including digital design, control theory, communication protocols, the exploration of system reliability and security, and more.

3.2 Finite state automata

Finite state automaton A Finite State Automaton (FSA) comprises three fundamental components:

1. The input tape, containing the input string $x \in \Sigma^*$.
2. The control unit, equipped with finite memory containing the state table.
3. The input head, initially positioned at the start marker of string x, which advances rightward with each move until it reaches the end-marker of the string or encounters an error.

Upon reading an input character, the automaton updates its current state in the control unit. After scanning the entire input string x, the automaton either recognizes or rejects the string based on its current state.

State-transition graph The state-transition graph is a directed graph that represents the automaton and consists of the following elements:

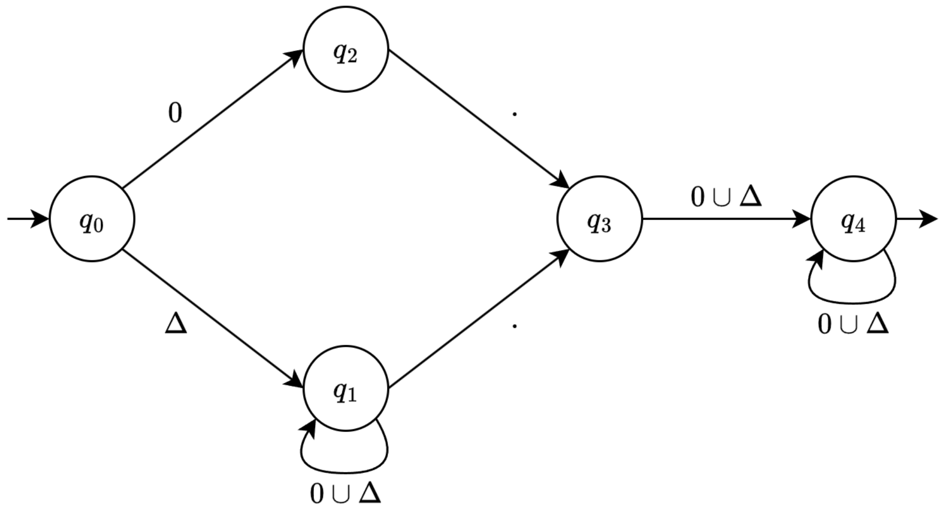
- *Nodes*: represent the states of the control unit.
- *Arcs*: represent the moves of the automaton.

Each arc is labeled with an input symbol and signifies a valid move when the current state matches the source state of the arc, and the current input symbol matches the arc label. The state-transition graph features a unique initial state but may have none, one, or more final states. It can be depicted using an incidence matrix, where each entry, indexed by the current state and input symbol, indicates the next state. This matrix is commonly known as the state table. Alternatively, a syntax diagram, the dual of the state-transition graph, can be employed.

Example:
Consider the language over the alphabet $\Sigma = \delta \cup \{0, \cdot\}$, where $\delta = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ generates decimal numbers. The corresponding regular expression is:

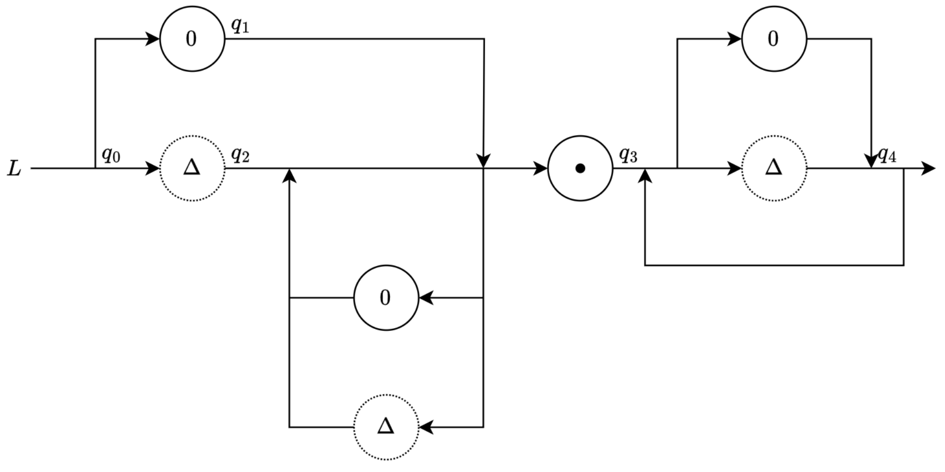
$$e = (0 \cup \delta(0 \cup \delta)^*) \cdot (0 \cup \delta)^+$$

The associated state-transition graph and state-transition table are depicted below:



Current state	**Current character**				
	0	1	...	9	$\cdot$
$\rightarrow q_0$	q_2	q_1	...	q_1	-
q_1	q_1	q_1	...	q_1	q_3
q_2	-	-	...	-	q_3
q_3	q_4	q_4	...	q_4	-
$q_4 \rightarrow$	q_4	q_4	...	q_4	-

The syntax diagram is:



3.3 Deterministic finite state automata

Deterministic finite automaton A finite deterministic automaton M consist of five elements:

1. Q, the state set (finite and not empty).
2. Σ, the input or terminal alphabet
3. $\delta : (Q \times \Sigma) \to Q$, the transition function.
4. $q_0 \in Q$, the initial state.
5. $F \subseteq Q$, the set of final states.

Moves The transition function captures the moves of the automaton as follows:

$$\delta(q_i, a) = q_j$$

This notation signifies that when the automaton, denoted as M, is in the current state q_i and encounters the input symbol a, it transitions the current state to q_j. If $\delta(q_i, a)$ is undefined, the automaton enters an error state, leading to the rejection of the input string. The general transition function operates over the domain $Q \times \Sigma^*$ and is defined as:

$$\delta(q, ya) = \delta(\delta(q, y), a) \qquad \text{where } a \in \Sigma \text{ and } y \in \Sigma^*$$

Definition (*Recognized string*)**.** A string is recognized if, and only if, as the automaton traverses a path labeled by x, it commences from the initial state and concludes at one of the final states:

$$\delta(q_0, x) \in F$$

It is noteworthy that the empty string is accepted only when the initial state is also a final state.

Definition (*Finite-state recognizable automata*)**.** The languages accepted by such automata are termed finite-state recognizable:

$$L(M) = \{x \in \Sigma^* | x \text{ is recognized by } M\}$$

Definition (*Equivalent automata*)**.** Two automata are considered equivalent if they accept the same language.

The time complexity of finite state automata is optimal: the input string x is either accepted or rejected in real-time. Since scanning the string from left to right requires precisely as many steps, the recognition time complexity cannot be further reduced.

3.3.1 Error state and total automata

In case the move is not specified in state q when processing character a, the automaton enters the error state q_{err}:

$$\forall q \in Q \forall a \in \Sigma \text{ if } \delta(q, a) \text{ is undefined then set } \delta(q, a) = q_{\text{err}}$$

Augmenting the deterministic automaton by introducing the error state is always feasible without altering the accepted language.

3.3.2 Clean automata

An automaton might contain components that do not contribute to any accepting computation and are therefore best removed.

Definition (*Reachable state*)**.** A state q is considered reachable from state p if there exists a computation that transitions from p to q.

Definition (*Accessible state*)**.** A state is accessible if it can be reached from the initial state.

Definition (*Post-accessible state*)**.** A state is post-accessible if a final state can be reached from it.

Definition (*Useful state*)**.** A state is called useful if it is both accessible and post-accessible.

Definition (*Clean automaton*)**.** An automaton is deemed clean if every state within it is useful.

Property 3.3.1. Every finite state automaton has an equivalent clean form.

To reduce an automaton, the process involves first identifying all the useless states and subsequently removing them from the automaton along with all their incoming and outgoing arcs.

3.3.3 Minimal automata

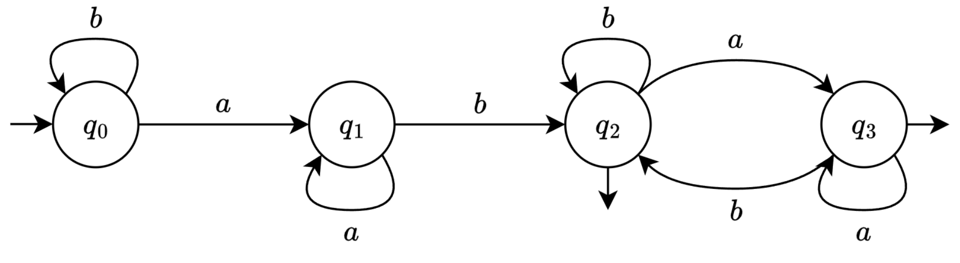
Property 3.3.2. Every finite state language possesses a unique deterministic finite state recognizer with the minimum possible number of states, known as the minimal automaton.

Definition (*Undistinguishable states*)**.** The states p and q are undistinguishable if, and only if, for every string $x \in \Sigma^*$, either both states $\delta(p, x)$ and $\delta(q, x)$ are final, or neither is final.

The merging of two indistinguishable states allows for a reduction in the number of states in the automaton without altering the recognized language. Undistinguishability is a relation that is symmetric, reflexive, and transitive.

Definition (*Distinguishable states*)**.** The states p and q are distinguishable if, and only if p is final and q is not, and $\delta(p, a)$ is distinguishable from $\delta(q, a)$.

Example:
Consider the following deterministic automaton:



The corresponding undistinguishability table is as follows:

	q_1	q_2	q_3
q_1	(1,1) (0,2)		
q_2	×	×	
q_1	×	×	(3,3) (2,2)

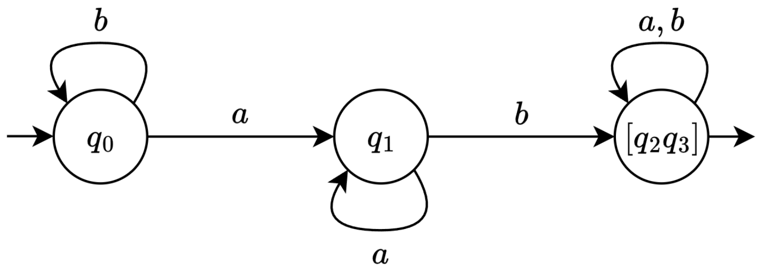
From the table, it is evident that the only indistinguishable states are q_2 and q_3.

Minimization The minimal automaton M', equivalent to the given automaton M has states corresponding to the equivalence classes of the undistinguishability relation. Defining the transition function of M' involves stating that there exists an arc from class $C_1 = [\dots, p_r, \dots]$ to class $C_2 = [\dots, q_s, \dots]$ if and only if in M, there is an arc from state p_r to q_s with the same label:

$$p_r \xrightarrow{b} q_s \Leftrightarrow C_1 = [\dots, p_r, \dots] \xrightarrow{b} C_2 = [\dots, q_s, \dots]$$

In other words, there is an arc between two states belonging to the two classes.

Example:
Consider the automaton from the previous example; it can be minimized by merging the two indistinguishable states found in the undistinguishability table. The resulting minimized automaton is as follows:



3.4 Nondeterministic automata

A right-linear grammar may include two alternative rules starting with the same character. This implies that in state A, upon reading the character, the machine can decide which of the subsequent states to enter, introducing non-determinism in its behavior. A machine move that does not read an input character is referred to as a spontaneous or epsilon move. These spontaneous moves also contribute to the nondeterministic nature of the machine. The primary advantages of nondeterminism are:

- The correspondence between grammars and automata suggests the inclusion of:
  - Moves with two or more destination states.
  - Spontaneous moves (or ε-moves).
  - Two or more initial states.
- Conciseness: defining a language using a nondeterministic automaton can often be more readable and compact compared to using a deterministic one.

Nondeterministic finite state automaton A nondeterministic finite automaton N, excluding spontaneous moves, is characterized by the following components:

- The state set Q.
- The terminal alphabet Σ.
- Two subsets of Q: the set I of the initial states and the set F of final states.
- The transition relation δ, a subset of the Cartesian product $Q \times \Sigma \times Q$.

A computation of length n initiates at state q_0 and concludes at state q_n, with the labeling $a_1 a_2 \dots a_n$.

Definition (*Acceptance condition*)**.** An input string x is accepted by the automaton if it represents the labeling of a path that begins at an initial state and concludes at a final state:

$$L(N) = \{x \in \Sigma^* | q \xrightarrow{x} \text{ with } q \in I \text{ and } r \in F\}$$

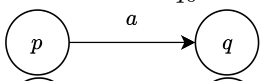
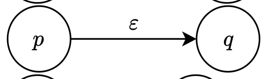
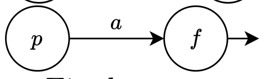
The moves of the nondeterministic automaton can be defined through a many-valued transition function. For a machine $N = (Q, \Sigma, \delta, I, F)$, without spontaneous moves, the transition function δ is defined to have the domain and image:

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \wp(Q)$$

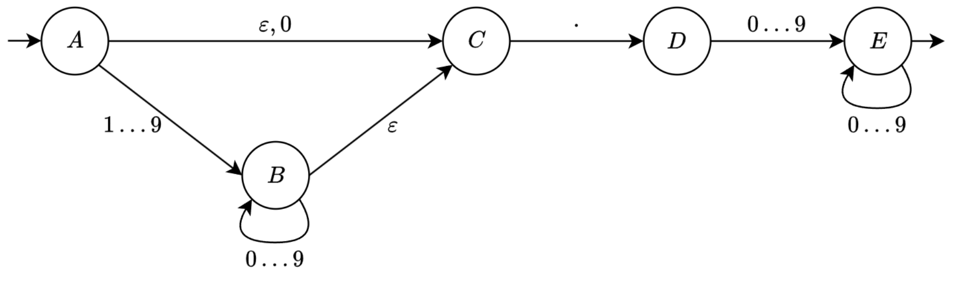
where the symbol $\wp(Q)$ denotes the power set of the set Q. A nondeterministic automaton may possess two or more initial states. However, constructing an equivalent nondeterministic automaton with only one initial state is straightforward. Introduce a new initial state q_0, connect it to the existing initial states through ε-arcs, designate these states as non-initial, and retain only q_0 as the initial state.

3.4.1 Correspondence between automata and grammars

Consider a strictly right-linear grammar $G = (V, \Sigma, P, S)$ and a nondeterministic automaton $N = (Q, \Sigma, \delta, q_0, F)$ with a unique initial state. The following equivalences hold:

Right-linear grammar	**Finite state automaton**
Nonterminal set V	Set of states $Q = V$
Axiom $S = q_0$	Initial state $q_0 = S$
$p \rightarrow aq$, where $a \in \Sigma$ and $p, q \in V$	
$p \rightarrow q$, where $p, q \in V$	
$p \rightarrow a$, where $p, a \in V$ (terminal rule)	
$p \rightarrow \varepsilon$	Final state p

Example:
Consider the following nondeterministic automaton:



It can be easily translated into a grammar following the rules in the table above:

$$\begin{cases} A \to 0C|C|1B|\ldots|9B \\ B \to 0B|\ldots|9B|C \\ C \to \cdot D \\ D \to 0E|\ldots|9E \\ E \to 0E|\ldots|9E|\varepsilon \end{cases}$$

As a result, a grammar derivation corresponds to an automaton computation, and vice versa.

Proposition. A language is generated by a right-linear grammar if and only if it is recognized by a finite automaton.

3.4.2 Ambiguity

Grammar derivations have a one-to-one correspondence with automaton computations, extending ambiguity from grammars to automata.

Definition (*Ambiguous automaton*)**.** An automaton is considered ambiguous if, and only if, the corresponding grammar exhibits ambiguity.

In other words, if a string x labels two or more accepting paths in the automaton. It is evident from the definition that a deterministic automaton is never ambiguous. Regular language families can also be defined using left-linear grammars. By substituting left for right, the mapping between such grammars and automata can be easily established.

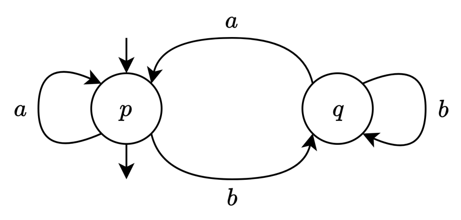
3.5 From automaton to regular expression

Consider a scenario where, for simplicity, the initial state i is unique, having no incoming arcs, and similarly, the final state t is unique without outgoing arcs. Any state other than i and t is referred to as internal. To establish an equivalent automaton, denoted as a generalized finite automaton, we extend the capability of arc labels to encompass regular languages.

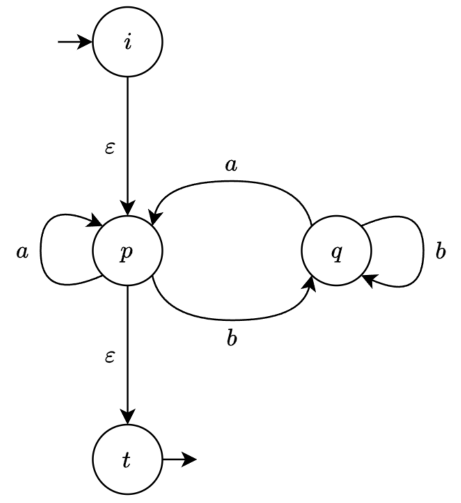
The process involves eliminating internal nodes one by one. After each elimination, one or more compensation arcs are added to maintain the equivalence of the automaton. These new arcs are labeled by regular expressions. Eventually, only the nodes i and t remain, connected by a single arc from i to t. The regular expression labeling this arc generates the complete language recognized by the original finite automaton.

The order in which eliminations are performed is not crucial. However, different orders may yield distinct regular expressions, all equivalent to each other but varying in complexity.

Example:
Consider the given automaton:



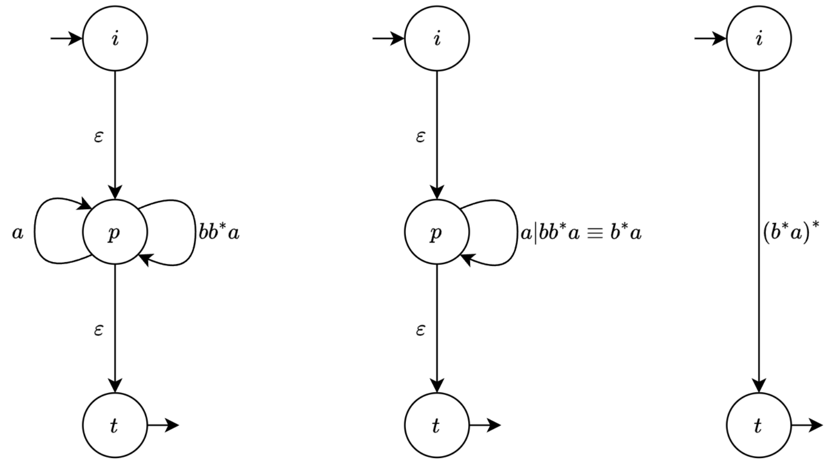
To normalize it and designate the initial and final states, we obtain:



Finally, we apply the Brzozowski and McCluskey method to the normalized automaton in three steps:

1. Eliminate the node q, replacing it with the regular expression bb^*a.
2. Merge the two cycles on node p with the choice operator, resulting in $a|bb^*a = b^*a$.
3. Remove the node p by replacing the label of the arc with $(b^*a)^*$.

The resulting automaton after these steps is as follows:

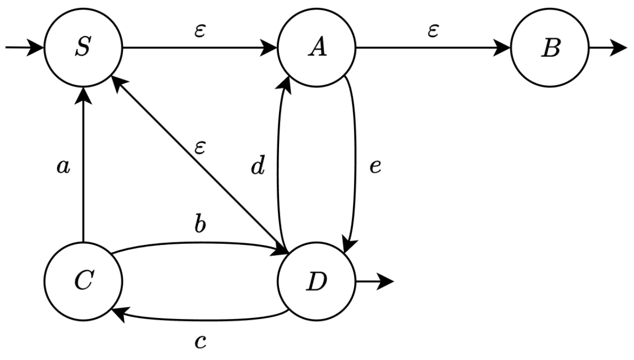


3.6 Elimination of nondeterminism

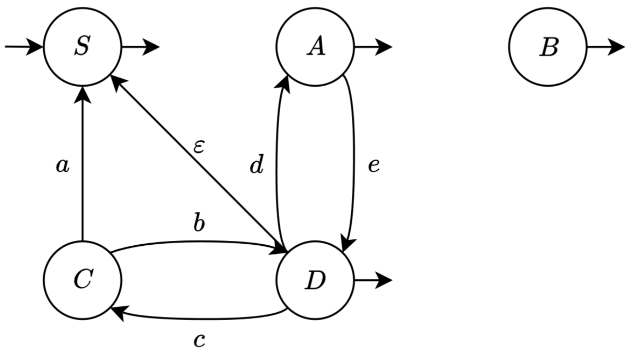
Every nondeterministic finite automaton can be transformed into an equivalent deterministic one. Consequently, every right-linear grammar has an equivalent non-ambiguous right-linear grammar. Therefore, any ambiguous regular expression can be transformed into a non-ambiguous one. The algorithm for converting a nondeterministic automaton into a deterministic one consists of two phases:

1. Elimination of spontaneous moves: as these moves correspond to copy rules, it suffices to apply the algorithm for removing copy rules.
2. Replacement of the non-deterministic multiple transitions by changing the automaton state set: this is the well-known subset construction.

Example:
Consider the following example:



After applying the algorithm, we obtain:



If, after eliminating all the ε-arcs, the automaton is still nondeterministic, proceed to the second phase.

3.7 From a regular expression to a finite state automaton

Several algorithms can transform a regular expression into an automaton, differing in their characteristics. The three main methods are:

1. *Thompson (structural) method.*

2. *Glushkov McNaughton Yamada method.*
3. *Berry-Sethi method.*

3.7.1 Locally testable language

Definition (*Initials set*)**.** The set of initials is defined as:

$$\text{Ini}(L) = \{a \in \Sigma | a\Sigma^* \cap L \neq \varnothing\}$$

To compute the sets of initials we use the following rules:

- $\text{Ini}(\varnothing) = \varnothing$.
- $\text{Ini}(\varepsilon) = \varnothing$.
- $\text{Ini}(a) = \{a\}$ for every character a.
- $\text{Ini}(e \cup e') = \text{Ini}(e) \cup \text{Ini}(e')$.
- $\text{Ini}(e \cdot e') =$ if $\text{Null}(e)$ then $\text{Ini}(e) \cup \text{Ini}(e')$ else $\text{Ini}(e)$.
- $\text{Ini}(e^*) = \text{Ini}(e^+) = \text{Ini}(e)$.

Definition (*Finals set*)**.** The set of finals is defined as:

$$\text{Fin}(L) = \{a \in \Sigma | \Sigma^* a \cap L \neq \varnothing\}$$

To compute the sets of finals we use the following rules:

- $\text{Fin}(\varnothing) = \varnothing$.
- $\text{Fin}(\varepsilon) = \varnothing$.
- $\text{Fin}(a) = \{a\}$ for every character a.
- $\text{Fin}(e \cup e') = \text{Fin}(e) \cup \text{Fin}(e')$.
- $\text{Fin}(e \cdot e') =$ if $\text{Null}(e')$ then $\text{Fin}(e) \cup \text{Fin}(e')$ else $\text{Fin}(e')$.
- $\text{Fin}(e^*) = \text{Fin}(e^+) = \text{Fin}(e)$.

Definition (*Digrams set*)**.** The set of digrams is defined as:

$$\text{Dig}(L) = \{x \in \Sigma^2 | \Sigma^* x \Sigma^* \cap L \neq \varnothing\}$$

To compute the sets of digrams we use the following rules:

- $\text{Dig}(\varnothing) = \varnothing$.
- $\text{Dig}(\varepsilon) = \varnothing$.
- $\text{Dig}(a) = \varnothing$ for every character a.
- $\text{Dig}(e \cup e') = \text{Dig}(e) \cup \text{Dig}(e')$.
- $\text{Dig}(e \cdot e') = \text{Dig}(e) \cup \text{Dig}(e') \cup \text{Fin}(e) \cdot \text{Ini}(e')$.

- $\text{Dig}(e^*) = \text{Dig}(e^+) = \text{Dig}(e) \cup \text{Fin}(e) \cdot \text{Ini}(e)$.

Definition (*Forbidden digrams set*)**.** The set of forbidden digrams is defined as:

$$\overline{\text{Dig}(L)} = \Sigma^2 \backslash \text{Dig}(L)$$

Definition (*Locally testable language*)**.** The language L is called locally testable, if and only if it satisfies the following identity:

$$L \backslash \{\varepsilon\} = \{x | \text{Ini}(x) \in \text{Ini}(L) \land \text{Fin}(x) \in \text{Fin}(L) \land \text{Dig}(x) \subseteq \text{Dig}(L)\}$$

Example:
Consider the language $L_1 = (abc)^*$. The sets defined for L_1 in this case are:

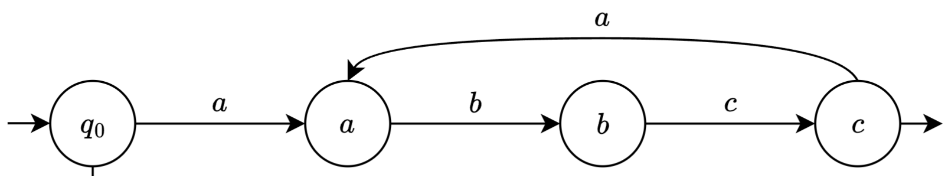
- $\text{Ini}(L_1) = \{a\}$.
- $\text{Fin}(L_1) = \{c\}$.
- $\text{Dig}(L_1) = \{ab, bc, ca\}$.
- $\overline{\text{Dig}(L)} = \{aa, ac, ba, bb, cb, cc\}$

Local language recognizer To create a recognizer for a local language, we examine the input string sequentially from left to right, verifying the following conditions: the starting character belongs to the Ini set, every digram is a member of Dig, and the concluding character is part of the Fin set. The acceptance of the string hinges on the success of all these checks. This recognition process can be implemented using a sliding window approach with a width of two characters, sliding over the input string from left to right. At each step of the sliding window, the contents are inspected. If the window reaches the end of the string and all checks succeed, the string is accepted; otherwise, it is rejected. This sliding window algorithm lends itself to straightforward implementation through a nondeterministic automaton. The recognizer associated with the sets Ini, Fin, and Dig possesses the following characteristics:

- *Initial states*: $q_0 \cup \Sigma$.
- *Final states*: Fin.
- *Transitions*: $q_0 \xrightarrow{a} a$ if $a \in \text{Ini}$, and $b \xrightarrow{a} b$ if $ab \in \text{Dig}$.

If the language includes the empty string, the initial state q_0 is also designated as final.

Example:
The automaton designed to recognize the language $L_1 = (abc)^*$ is depicted below:



Definition (*Linear regular expression*)**.** A regular expression is termed linear if it does not contain any repeated generators.

Property 3.7.1. The languages generated by linear regular expressions are local.

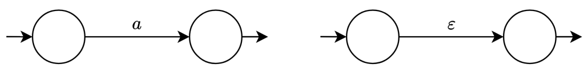
Linearity ensures that the subexpressions of a regular expression are defined over distinct alphabets. Since a regular expression is the composition of its subexpressions, the language generated by a linear regular expression becomes local due to the closures of local languages over disjoint alphabets. It's important to note that the converse does not hold. This observation implies that constructing a recognizer for a general regular language simplifies to determining the characteristic local sets (Ini, Fin, Dig) for such a language, provided the alphabet undergoes slight modification.

Empty string generation To verify whether a regular expression e generates the empty string, the Null(e) operator can be employed. This operator evaluates to true when the empty string is part of the regular expression and false otherwise. The functioning of Null(e) is outlined as follows:

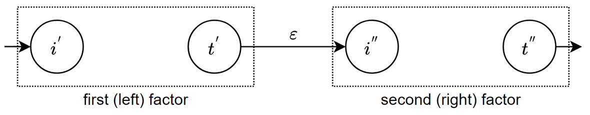
- Null$(\varnothing)$ = false.
- Null(ε) = true.
- Null(a) = false for every character a.
- Null$(e \cup e^{'})$ = Null$(e) \vee$ Null$(e^{'})$.
- Null$(e \cdot e^{'})$ = Null$(e) \wedge$ Null$(e^{'})$.
- Null(e^{*}) = true.
- Null(e^{+}) = Null(e).

3.7.2 Thompson structural method

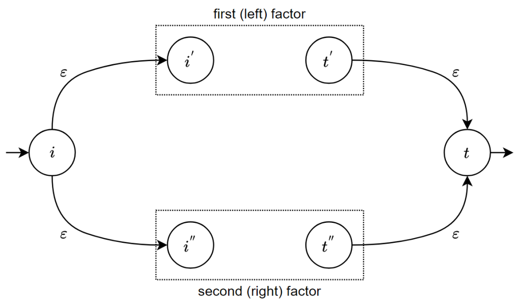
The Thompson structural method modifies the original automaton to have unique initial and final states. It is is based on the correspondence of regular expression and recognizer automaton. The rules used to find the automaton are the following:



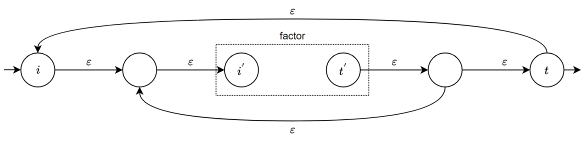
Atomic expressions



Concatenation



Union



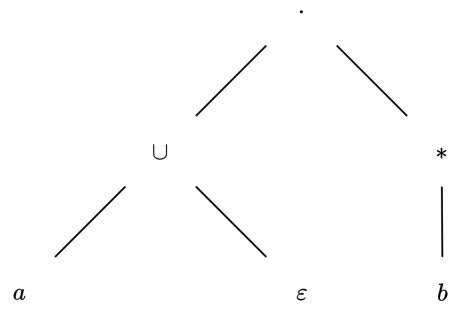
Star closure

In general, the outcome of the Thompson method is a nondeterministic automaton with spontaneous moves. This method is an application of the closure properties of regular languages under the operations of union, concatenation, and star.

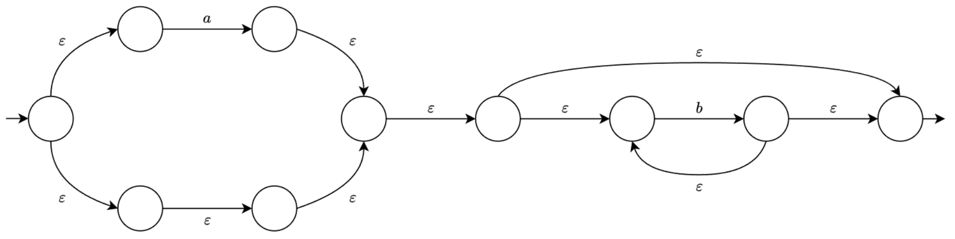
Example:
Consider the regular expression $(a \cup \varepsilon)b^*$. Rewriting the same regular expression with symbols and subexpression indexing yields:

$$\big(_1 \, (_2 \, (_3 a)_4 \cup (_5 \varepsilon)_6)_7 \, (_8 \, (_9 b)_{10})^*_{11}\big)_{12}$$

The corresponding structure tree is as follows:



By applying the rules from the table, the resulting automaton is:



The found automaton can be optimized to avoid redundant states.

3.7.3 Glushkov McNaughton Yamada method

The Glushkov-McNaughton-Yamada (GMY) algorithm is employed to construct an automaton equivalent to a given regular expression. The algorithm assigns states in a one-to-one correspondence with the generators occurring in the regular expression. The GMY algorithm, grounded in the linearity of regular expressions, follows these steps:

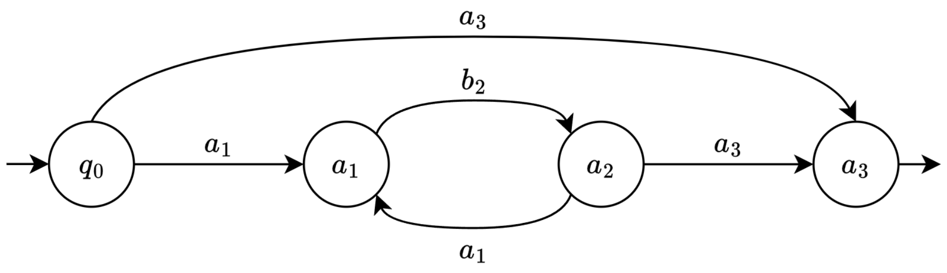
1. Enumerate the regular expression e and derive the linear regular expression $e_\#$.
2. Compute the three characteristic local sets (Ini, Fin, Dig) of $e_\#$.
3. Design the recognizer for the local language generated by $e_\#$.
4. Remove the indexing, resulting in the recognizer for e.

Example:
Consider the regular expression $e = (ab)^*a$. Applying the GMY algorithm involves the following steps:

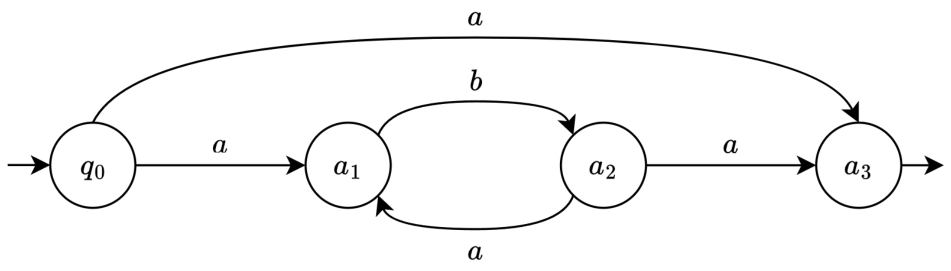
1. Enumerate the regular expression, obtaining:

$$e_\# = (a_1 b_2)^* a_3$$

2. Compute the sets:
   - $\text{Ini}(e) = \{a\}$.
   - $\text{Fin}(e) = \{a\}$.
   - $\text{Dig}(e) = \{ab, ba\}$.
3. Construct the recognizer for the numbered expression:



4. Remove the enumeration:



The result is a non-deterministic automaton without spontaneous moves, featuring as many states as there are occurrences of generators in the regular expression, plus one additional state.

3.7.4 Berry-Sethi method

To obtain the deterministic recognizer, we can apply the subset construction to the non-deterministic recognizer generated by the GMY algorithm. However, a more direct algorithm known as Berry-Sethi exists. The underlying idea of the Berry-Sethi algorithm is as follows:

1. Consider the end-marked regular expression $e \dashv$ instead of the original regular expression e.
2. Let e be a regular expression over the alphabet Σ, and let $e_\#$ be the numbered version of e over $\Sigma_\#$ with the Null predicate and local sets Ini, Fin, and Dig.
3. Define the set Fol as follows:

   (a) $\text{Fol}(c_\#) \in \wp(\Sigma_\# \cup \{\dashv\})$.

(b) $\text{Fol}(\dashv) = \varphi$.

(c) $\text{Fol}(a_i) = \{b_j | a_i b_j \in \text{Dig}(e_\# \dashv)\}$ where a_i and b_j may coincide.

4. Apply the subsequent algorithm.

Algorithm 1 Berry-Sethi algorithm

```
 1: $q_0 \leftarrow Ini(e_\# \dashv)$
 2: $Q \leftarrow \{q_0\}$
 3: $\delta \leftarrow \varnothing$
 4: while $\exists q \in Q$ such that $q$ is unmarked do
 5:     mark state $q$ as visited
 6:     for each character $c \in \Sigma$ do
 7:         $q^{'} \leftarrow \bigcup_{\forall c_\# \in \Sigma_{c_\#}} Fol(c_\#)$
 8:         if $q^{'} \neq \varnothing$ then
 9:             if $q^{'} \notin Q$ then
10:                 set $q^{'}$ as a new unmarked state
11:                 $Q \leftarrow Q \cup \{q^{'}\}$
12:             end if
13:             $\delta \leftarrow Q \cup \{q^{'}\}$
14:         end if
15:     end for
16: end while
```

Example:
Apply the Berry-Sethi (BS) algorithm to the language, starting with the enumeration of the string:

$$e_\# = (a_1 | b_2 b_3)^* (a_4 c_5)^+ \dashv$$

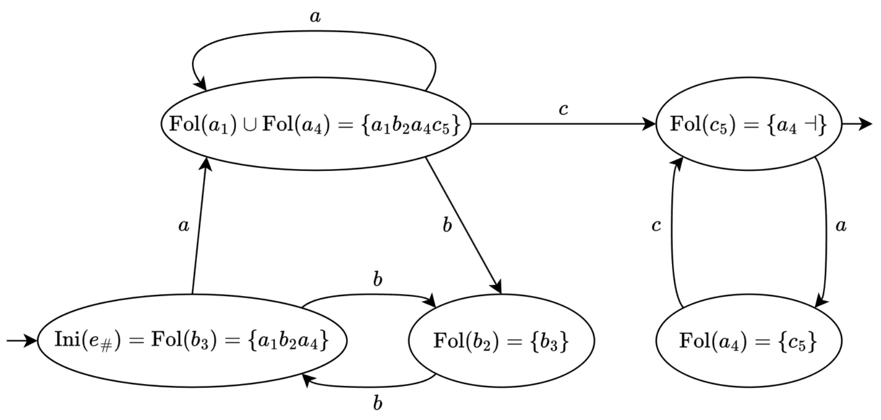
The characteristic sets are defined as follows:

- $\text{Ini}(e_\#) = \{a_1, b_2, a_4\}$.
- $\text{Fin}(e_\#) = \{\dashv\}$.
- $\text{Dig}(e_\#) = \{a_1a_1, a_1b_2, a_1a_4, b_2b_3, b_3a_1, b_3b_2, b_3a_4, a_4c_5, c_5a_4, c_5 \dashv\}$.

The table of followers is given by:

$\mathbf{c_\#}$	$\mathbf{Fol(c_\#)}$
a_1	$a_1b_2a_4$
b_2	b_3
b_3	$a_1b_2a_4$
a_4	c_5
c_5	$a_4 \dashv$

The resulting automaton is depicted in the following figure:

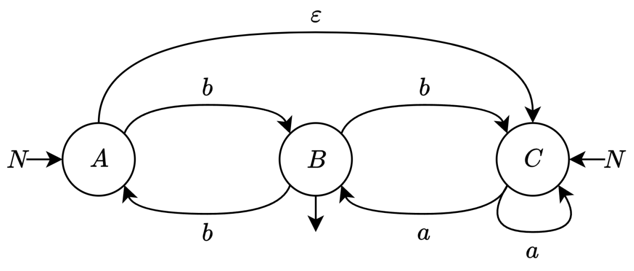


$\mathrm{Fol}(a_1) \cup \mathrm{Fol}(a_4) = \{a_1 b_2 a_4 c_5\}$
$\mathrm{Fol}(c_5) = \{a_4 \dashv\}$
$\mathrm{Ini}(e_\#) = \mathrm{Fol}(b_3) = \{a_1 b_2 a_4\}$
$\mathrm{Fol}(b_2) = \{b_3\}$
$\mathrm{Fol}(a_4) = \{c_5\}$
a
c
b

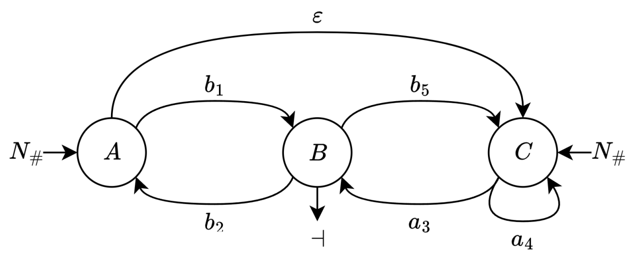
The Berry-Sethi algorithm provides a method for converting a nondeterministic automaton into a deterministic one. The procedure involves the following steps:

1. Enumerate the elements present on the arcs.
2. Generate the followers table based on the enumerated elements.
3. Reconstruct the automaton using the information from the followers table.

Example:
Consider the given automaton:



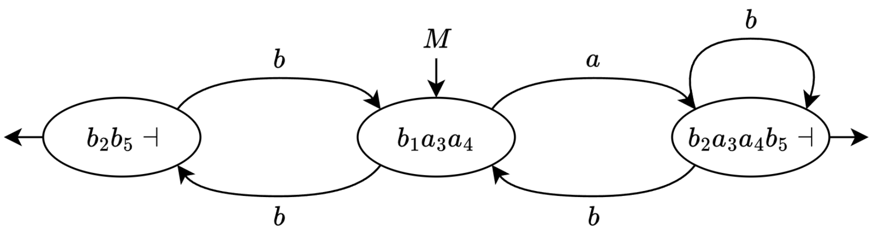
Its numbered version is as follows:



The corresponding follower table is provided below:

$c_\#$	$\mathrm{Fol}(c_\#)$
b_1	$b_2 b_5 \dashv$
b_2	$b_1 a_3 a_4$
a_3	$b_2 b_5 \dashv$
a_4	$a_3 a_4$
b_5	$a_3 a_4$

The resulting deterministic automaton is depicted below:



3.8 Complement and intersection of regular languages

Regular expressions can incorporate operators such as complement, intersection, and set difference, enhancing their conciseness. These operators are valuable tools for refining regular expressions. The REG family exhibits closure properties under complement, intersection, and set difference operations.

Complement The complement of a language L is defined as:

$$\neg L = \Sigma^* \backslash L$$

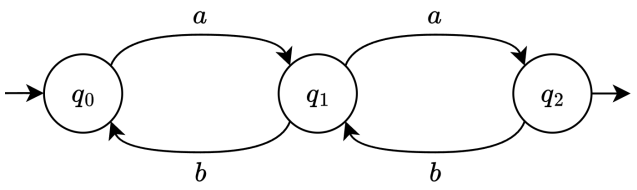
Assuming the recognizer M of L is deterministic, with initial state q_0, state set Q, set of finals states F and transition function δ, constructing a deterministic automaton $\overline{M}$ for the complement language $\neg L$ involves the following steps:

1. Create the error state $p \notin Q$, expanding the states of $\overline{M}$ to $Q \cup \{p\}$.
2. Define the transition function $\overline{\delta}$:
   - $\overline{\delta}(q, a) = \delta(q, a)$ if $\delta(q, a) \in Q$.
   - $\overline{\delta}(q, a) = p$ if $\delta(q, a)$ is undefined.
   - $\overline{\delta}(p, a) = p$ for every character $a \in \Sigma$.
3. Swap the non-final and final states:

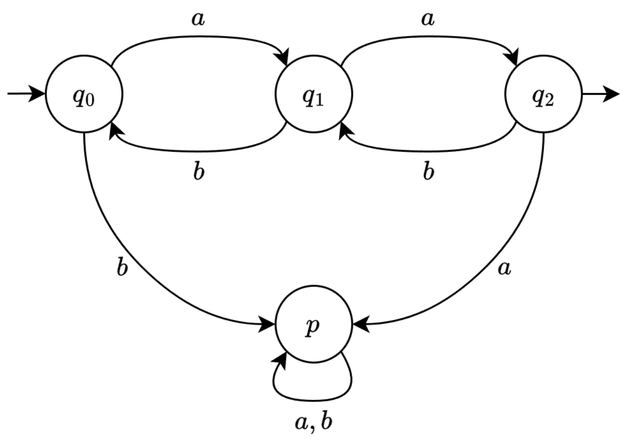
$$\overline{F} = (Q \backslash F) \cup \{p\}$$

It's important to note that a recognizing path of M ($x \in L(M)$) does not terminate in a final state of $\overline{M}$, and conversely, a non-recognizing path of M ($x \notin L(\overline{M})$) does not end in a final state of $\overline{M}$.

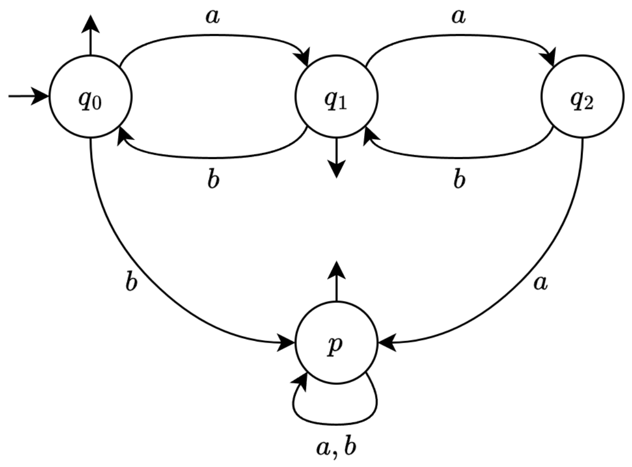
Example:
Consider the deterministic automaton depicted below:



To obtain the complement automaton, introduce the error state p, resulting in the modified automaton:



Finally, interchange final and non-final states to achieve the complement automaton:



For the complement construction to function accurately, it is imperative that the original automaton be deterministic. Failure to meet this criterion may result in non-disjoint original and complement languages, violating the complement definition. Additionally, the complement automaton might encompass redundant states and may not be in its minimal form; hence, it should undergo reduction and minimization as needed.

Cartesian product The product is a commonly used construction in formal languages, where a single automaton simulates the computation of two automata operating in parallel on the same input string. This construction is particularly valuable for building the intersection automaton. The steps to obtain the intersection automaton involve applying the De Morgan theorem:

1. Construct deterministic recognizers for the two languages.
2. Create the respective complement automata.
3. Form their union using the Thompson construction.
4. Transform the union automaton into a deterministic one using the Berry-Sethi construction.
5. Apply complementation again to obtain the intersection automaton.

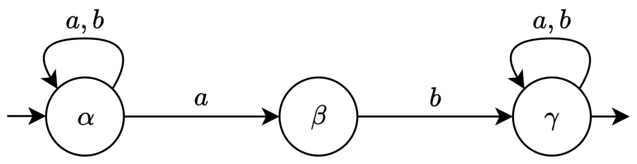
Since the intersection of the two languages is directly recognized by the Cartesian product of their automata, obtaining the intersection automaton can be achieved more directly. Assuming neither automaton contains spontaneous moves, the state set of the product machine is the Cartesian product of the state sets of the two automata. Each product state is a pair $\langle q', q'' \rangle$, where the left (right) member is a state of the first (second) machine. The move is defined as follows:

$$\left\langle q', q'' \right\rangle \xrightarrow{a} \left\langle r', r'' \right\rangle \text{ if and only if } q' \xrightarrow{a} r' \text{ and } q'' \xrightarrow{a} r''$$

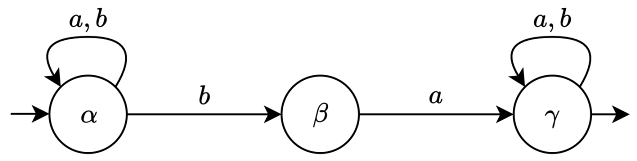
The product machine has a move if and only if the projection of such a move onto the left (right) component is a move of the first (second) automaton. The initial and final state sets are the Cartesian products of the initial and final state sets of the two automata, respectively.

Example:

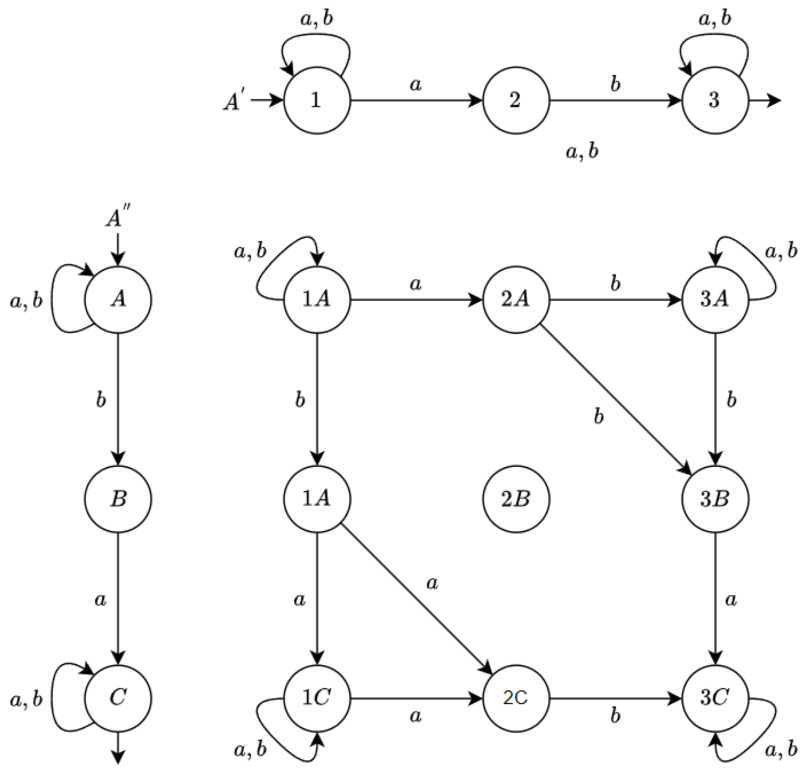
Consider the languages $L' = (a|b)^*ab(a|b)^*$ and $L'' = (a|b)^*ba(a|b)^*$. The deterministic automaton for the language L' is illustrated below:



Similarly, the deterministic automaton for the language L'' is depicted as follows:



The intersection of these two languages is determined using the following table:



CHAPTER 4

Pushdown automata

4.1 Introduction

To recognize context-free languages, an automaton with an auxiliary component structured as an unbounded stack of symbols is necessary. The stack operations include:

- *Push*: adds the symbol(s) onto the top of the stack.
- *Pop*: removes the symbol from the top of the stack if it's not empty; otherwise, reads Z_0.
- *Stack emptiness test*: returns true if the stack is empty, and false otherwise.

The symbol Z_0 represents the stack bottom and can be read but not removed. Additionally, the symbol $\dashv$ serves as the terminator character for the input string. The configuration is determined by the current state, current character, and stack contents. During a move, the pushdown automaton:

- Reads the current character, shifting the input head or performing a spontaneous move without shifting the input head.
- Reads the stack's top symbol, removing it if the stack is not empty, or reading the stack symbol Z_0 if the stack is empty.
- Depending on the current character, state, and stack top symbol, transitions into the next state and places none, one, or more symbols onto the top of the stack.

Pushdown automaton A pushdown automaton M is defined by:

- Q: a finite set of states for the control unit.
- Σ: a finite input alphabet.
- Γ: a finite stack alphabet.
- δ: a transition function.
- $q_0 \in Q$: the initial state.

- $Z_0 \in \Gamma$: the initial stack symbol.
- $F \subseteq Q$: a set of final states.

The transition function δ is defined as follows:

- *Domain*: $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$.
- *Image*: the set of the subsets of $Q \times \Gamma^*$.

Possible moves include:

- *Reading move*: in state q with symbol Z_0 on the top of the stack, the automaton reads the character a and transitions to one of the states p_i (where $1 \leq i \leq n$), after sequentially performing pop and push operations (γ_i):
$$\delta(q, a, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \dots, (p_n, \gamma_n)\}$$
- *Spontaneous move*: n state q with symbol Z_0 on the top of the stack, the automaton does not read any input character and transitions to one of the states p_i (where $1 \leq i \leq n$), after sequentially executing pop and push operations (γ_i):
$$\delta(q, \varepsilon, Z) = \{(p1, \gamma_1), (p2, \gamma_2), \dots, (p_n, \gamma_n)\}$$

Current configuration	**Next configuration**	**Applied move**
$(q, az, \eta Z)$	$(p, z, \eta\gamma)$	Reading
$(q, az, \eta Z)$	$(p, az, \eta\gamma)$	Spontaneous

Pushdown automaton configurations Non-determinism arises when considering a triple (state, input, stack top), as there exist two or more potential moves that involve consuming either none or one input character.

Definition (*Instantaneous configuration*)**.** The instantaneous configuration of a machine M is a 3-tuple:
$$(q, y, \eta) \in Q \times \Sigma^* \times \Gamma^+$$

Here, q is the current state of the control unit, y is the part of the input string x that still has to be scanned, and η is the current contents of the pushdown stack.

Definition (*Initial configuration*)**.** The *initial* configuration of machine M is:
$$(q_0, x, Z_0)$$

Definition (*Final configuration*)**.** The *final* configuration of machine M is:
$$(q, \varepsilon, \lambda)$$

Executing a move results in a transition from one configuration to another, represented as:
$$(q, y, \eta) \to (p, z, \lambda)$$

It is important to note that a sequence of one or more transitions is denoted by $\xrightarrow{+}$.

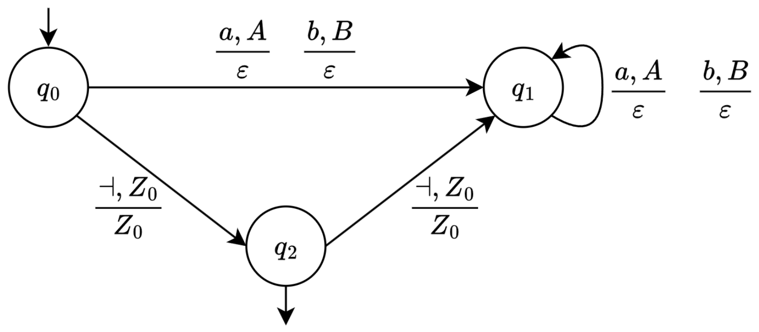
An input string x is deemed accepted by a final state if the following computation exists:
$$(q_0, x, Z_0) \overset{*}{\mapsto} (q, \varepsilon, \lambda)$$

Here, $q \in F$ and $\lambda \in \Gamma^*$. It is worth mentioning that there is no specific condition for λ; sometimes it may be the empty string, but this is not a mandatory requirement.

4.1.1 State-transition graph

The transition function of a finite automaton can be visually depicted. The numerator of the representation denotes the characters read from the input tape and the memory. Meanwhile, the denominator indicates the replacement for the element in the stack memory.

Example:
The language $L = \{uu^R | u \in \{a, b\}^*\}$, consisting of palindromes of even length, is recognized with a final state by the pushdown recognizer. The graphical representation is illustrated in the following figure:


q_0
q_1
q_2
$\frac{a, A}{\varepsilon}$ $\frac{b, B}{\varepsilon}$
$\frac{a, A}{\varepsilon}$ $\frac{b, B}{\varepsilon}$
$\frac{\dashv, Z_0}{Z_0}$
$\frac{\dashv, Z_0}{Z_0}$

4.1.2 From grammar to pushdown automata

Grammar rules can be interpreted as the instructions for a nondeterministic pushdown automaton, which operates in a goal-oriented manner, utilizing the stack as a notebook for the sequence of upcoming actions.

The stack symbols in this automaton can represent both terminals and non-terminals from the grammar. If the stack contains the symbol sequence $A_1 \dots A_k$, the automaton first executes the action associated with A_k. This action aims to determine, from the current position a_i in the input string, if there exists a string w derivable from A_k. If so, the action shifts the input head by $|w|$ positions.

An action may recursively break down into a series of sub-actions, especially when recognizing the non-terminal symbol A_k requires recognizing other non-terminals.

The initial action corresponds to the grammar axiom: the pushdown recognizer checks whether the source string can be derived from the axiom. Initially, the stack contains only the symbol Z_0 and the axiom S, and the input head is positioned at the beginning of the input string. At each step, the automaton nondeterministically chooses an applicable grammar rule and performs the corresponding move. The input string is considered accepted only when it has been entirely scanned and the stack is empty.

Given a grammar $G = (V, \Sigma, P, S)$ with $A, B \in V$, $b \in \Sigma$ and $A_i \in V \cup \Sigma$, the correspondence between a grammar and a pushdown automaton is summarized as follows:

Grammar rule	**Automaton move**
$A \to BA_1 \dots A_n$ with $n \geq 0$	If top $= A$ then pop Push $(A_n \dots A_1 B)$
$A \to bA_1 \dots A_n$ with $n \geq 0$	If $cc = b$ and top $= A$ then pop Push $(A_n \dots A_1)$ Shift reading head
$A \to \varepsilon$	If top $= A$ then pop
For every character $b \in \Sigma$	If $cc = b$ and top $= b$ then pop Shift reading head
—	If $cc = \dashv$ and the stack is empty then accept Halt

Example:
Consider the language $L = \{a^n b^m | n \geq m \geq 1\}$, the corresponding nondeterministic grammar generating strings of this language is presented below:

$$\begin{cases} S \to aS \\ S \to A \\ A \to aAb \\ A \to ab \end{cases}$$

Utilizing the rules from the previous table, we can construct the pushdown automaton as depicted in the following:

#	**Grammar rule**	**Automaton move**
1	$S \to aS$	If $cc = a$ and top $= S$ then pop Push S Shift reading head
2	$S \to A$	If top $= S$ then pop Push(A)
3	$A \to aAb$	If $cc = a$ and top $= A$ then pop Push bA Shift reading head
4	$A \to ab$	If $cc = a$ and top $= A$ then pop Push b Shift reading head
5	—	If $cc = b$ and top $= b$ then pop Shift reading head
6	—	If $cc = \dashv$ and the stack is empty then accept Halt

The automaton recognizes a string if and only if the string is generated by the grammar; each successful automaton computation corresponds to a grammar derivation, and vice versa. Therefore, the automaton simulates the leftmost derivations of the grammar. However, the automaton lacks the ability to guess the correct derivation; it must explore all possibilities. A string is accepted by two or more different computations if and only if the grammar is ambiguous.

Reverse transformation The process of constructing an automaton from grammar can be reversed, yielding the grammar by starting from the pushdown automaton, provided it is a one-state automaton.

Property 4.1.1. The family of free languages generated by free grammars is identical to the family of languages recognized by one-state pushdown automata.

Unfortunately, in general, the resulting pushdown automaton is nondeterministic, as it explores all applicable moves at any point and exhibits an exponential time complexity concerning the length of the source string.

4.1.3 Varieties of pushdown automata

There are three acceptance modes for pushdown automata:

1. *By final state*: accepts when it enters a final state regardless of the stack contents.
2. *By empty stack*: accepts when the stack becomes empty regardless of the current state.
3. *Combined*: accepts by both final state and empty stack.

Property 4.1.2. For the family of nondeterministic pushdown automata with states, the three acceptance modes mentioned above are equivalent.

A generic pushdown automaton may perform an unlimited number of moves without reading any input character, but only if it enters a loop consisting only of spontaneous moves. This behavior either prevents it from fully reading the input string or causes it to execute an unlimited number of moves before deciding to accept or reject the string. Both of these behaviors are undesirable in practice. However, it is always possible to construct an equivalent automaton without spontaneous loops.

On-line mode A pushdown automaton operates in an on-line mode if it decides to accept or reject the string as soon as it reads the last character of the input string and does not execute any further moves. From a practical perspective, the on-line mode is desirable. It is always feasible to build an equivalent automaton that operates in an on-line mode.

4.1.4 Context-free languages and pushdown automata

The language accepted by a generic pushdown automaton is context free. Moreover, any free language can be recognized by a nondeterministic one-state pushdown automaton.

Property 4.1.3. The family CF of context-free languages is equivalent to the set of languages recognized by unrestricted pushdown automata.

Property 4.1.4. The family CF of context-free languages is identical to the set of languages recognized by nondeterministic one-state pushdown automata.

4.1.5 Intersection of regular and context-free languages

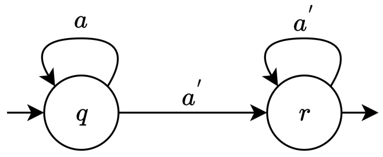
It can be easily demonstrated that the intersection of a context free and a regular language is also context free. To recognize the intersection$L(G) \cap L(A)$ where G is a grammar and A is a finite state automaton, the pushdown automaton M can be obtained as follows:

1. Construct a one-state pushdown automaton N that recognizes $L(G)$ by empty stack.
2. Build the pushdown automaton M whose state-transition graph is the Cartesian product of the state-transition graphs of N and A using the Cartesian product construction. Ensure that the actions of M on the stack are the same as those of N.

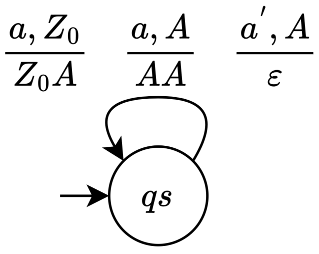
The resulting pushdown automaton M:

1. Has pairs of states from the component machines N and A as its states.
2. Accepts by the combined acceptance mode of final state and empty stack.
3. Consider states containing a final state of A as final states.
4. Is deterministic if both component machines N and A are deterministic.
5. Accepts by final state all and only the strings belonging to the intersection language.

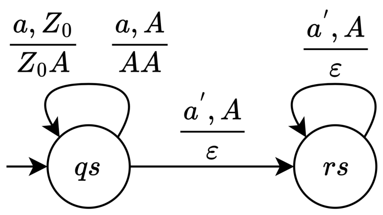
Example:
Consider the Dyck language $L = \{a^n a'^n | n \geq 1\}$. A finite state automaton that recognizes the language by final state can be constructed as follows:



Additionally, a pushdown automaton that recognizes the language by empty stack can be constructed as follows:



Applying the intersection, we obtain a product pushdown automaton that recognizes by empty stack in the final state (r, s):



4.2 Deterministic pushdown automata

Nondeterminism is eliminated when the transition function δ is single-valued. In such cases:

- If $\delta(q, a, A)$ is defined, then $\delta(q, \varepsilon, A)$ is undefined.
- If $\delta(q, \varepsilon, A)$ is defined, then $\delta(q, a, A)$ is undefined for any $a \in \Sigma$.

When the transition function lacks nondeterministic aspects, the automaton becomes deterministic, and correspondingly, the recognized language is deterministic as well. It's important to note that a deterministic pushdown automaton may still incorporate spontaneous moves.

Property 4.2.1. The family DET of the deterministic free languages is properly included in the family CF of all the free languages:

$$\text{DET} \subseteq \text{CF}$$

For any languages L, D, and R belonging to the families CF, DET, and REG, respectively, the following closure properties hold:

Operation	**Property (deterministic set)**	**Property (context-free set)**
Reversal	$D^R \notin \text{DET}$	$D^R \in \text{CF}$
Star	$D^* \notin \text{DET}$	$D^* \in \text{CF}$
Complement	$\neg D \in \text{DET}$	$\neg L \notin \text{CF}$
Union	$D_1 \cup D_2 \notin \text{DET}$ $D \cup R \in \text{DET}$	$D_1 \cup D_2 \in \text{CF}$
Concatenation	$D_1 \cdot D_2 \notin \text{DET}$ $D \cdot R \in \text{DET}$	$D_1 \cdot D_2 \in \text{CF}$
Intersection	$D \cap R \in \text{DET}$	$D_1 \cap D_2 \notin \text{CF}$

CHAPTER 5

Syntax analysis

5.1 Introduction

Given a grammar G, the syntax analyzer reads a source string and:

1. If the string belongs to the language $L(G)$, then exhibits a derivation or builds a syntax tree of the string.
2. Otherwise, it stops and notifies the error, possibly with a diagnostic.

If the source string is ambiguous, the result of the analysis is a set of derivations, also called forest of trees. There are two analyzer classes depending on whether the derivation is rightmost or leftmost, and on the reconstruction order of the derivation.

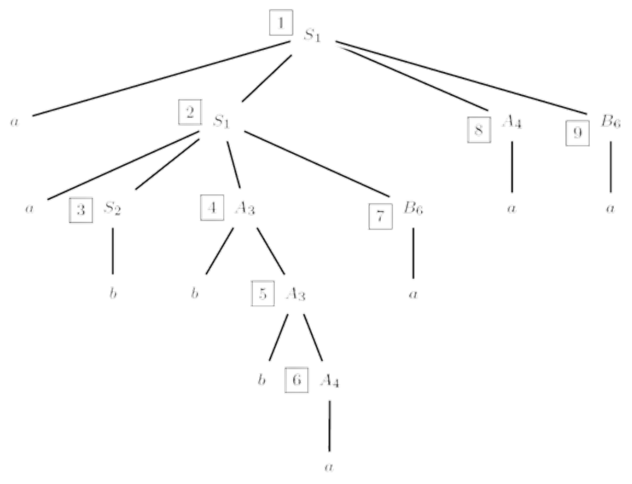
Definition. The *bottom-up analysis* constructs the rightmost derivation in inverse order and analyzes the tree from leaves to root through reductions.

The *top-down analysis* constructs the leftmost derivation in direct order and analyzes the tree from root to leaves through expansions.

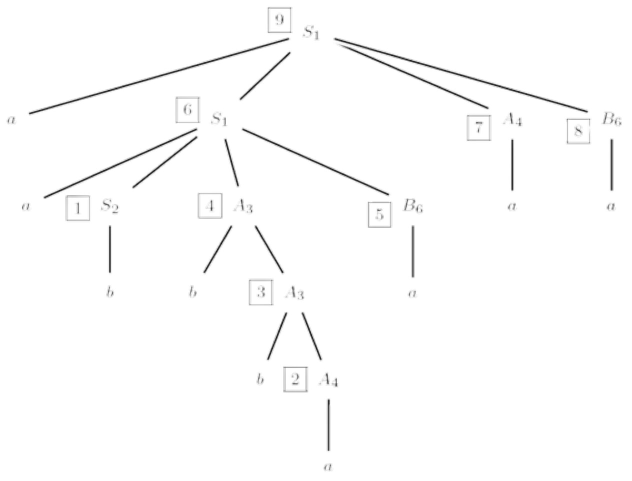
Example:
Consider the string *aabbbaaaa* generated by the grammar:

$$\begin{cases} S \to aSAB \\ S \to b \\ A \to bA \\ B \to cB \\ B \to a \end{cases}$$

The tree corresponding to the top-down analysis of the given string is as follows:



Note that the numbering indicates the application order of the rules.
The tree corresponding to the bottom-up analysis of the given string is as follows:



5.2 Grammar as network of finite automata

Let's G be an EBNF in which each nonterminal has one rule $A \to \alpha$, where α is a regular expression over terminals and nonterminals. The regular expression α defines a regular language and, consequently, there is a finite state automaton M_A that recognizes α.

A transition of the grammar M_A labeled with nonterminal B is interpreted as a call to the automaton M_B. If $B = A$ we the call is termed recursive.

Definition. The finite state automata of the nonterminals of G are called *machines*.
The pushdown machine that analyzes $L(G)$ is called *automaton*.
The set of all the machines of G is called *network*.

The elements used to define a network are:

1. The alphabet of the terminal symbols: Σ
2. The alphabet of the nonterminal symbols: $V = \{S, A, B, \dots\}$.
3. The grammar rules: $S \to \sigma$, $A \to \alpha$, $B \to \beta$, etc.

4. The regular languages over $\Sigma \cup V$ defined by σ, α, β, etc.: R_S, R_A, R_B, etc.
5. The deterministic finite machine recognizing R_S, R_A, R_B, etc.: M_S, M_A, M_B, etc.
6. The machine network: $\mathcal{M} = \{M_S, M_A, M_B, \dots\}$

We need to consider also the terminal language defined by a generic machine M_A, when starting from a state possibly other than the initial one. For any state q_A,not necessarily initial, we write as:

$$L(M_A, q_A) = \{y \in \Sigma^* | \exists \eta \in R(M_A, q) \wedge \overset{*}{\underset{G}{\Longrightarrow}} y\}$$

The formula above contains a string η over terminals and non-terminals, accepted by machine M_A when starting in the state q. The derivations originating from η produce all the terminal strings of language $L(q)$. In particular, it follows that:

$$L(M_A, 0_A) = L(0_A) = L_A(G)$$

and for the axiom it is:

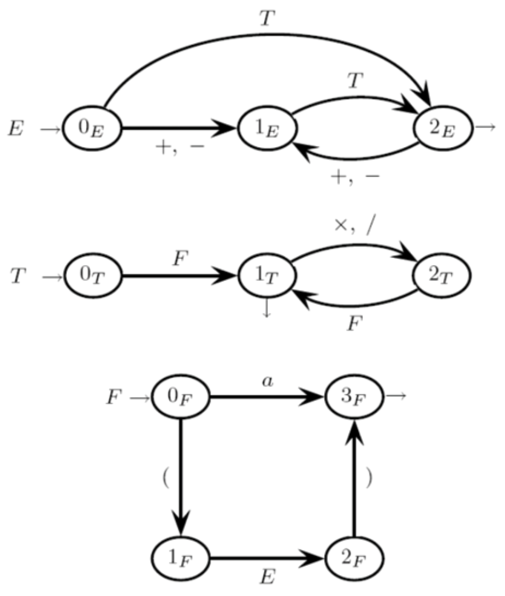
$$L(M_S, 0_S) = L(0_S) = L_S(G) = L(\mathcal{M})$$

We have to set an additional constraint, that is: the initial state 0_A of machine A is never re-entered after the start of the computation. This constraint is easily fulfilled, at worst the machine needs a new initial state. We say that the automata that satisfy such a condition are normalized.

Example:
Consider the grammar for arithmetic expressions:

$$\begin{cases} E \to [+|-]T((+|-)T)^* \\ T \to F((\times|/)F)^* \\ F \to a|'('E')' \end{cases}$$

The corresponding network is as follows:



Note that the machine T has been normalized.

These networks can be translated into programs that uses recursion. The code of the program reflects the machine transitions. When the finite automaton has a bifurcation state, to decide which direction to take we must watch all the symbols that appear on the arcs that leave the state, included the final darts of the machine.

Example:
The machine net from the previous example can be transformed into three code procedures. The procedure for the grammar E is:

```
call $T$
**while** $cc = +$ **do**
    $cc := next$
    call $T$
**end while**
**if** $cc \in \{(\dashv\}$ **then**
    **return**
**else**
    error
**end if**
```

The procedure for the grammar T is:

```
call $F$
**while** $cc = \times$ **do**
    $cc := next$
    call $F$
**end while**
**if** $cc \in \{(+\ \dashv\}$ **then**
    **return**
**else**
    error
**end if**
```

The procedure for the grammar F is:

```
**if** $cc = a$ **then**
    $cc := next$
**else if** $cc =$ ( **then**
    $cc := next$
    call $E$
    **if** $cc =)$ **then**
        $cc := next$
        **if** $cc \in \{)\times\ \dashv\}$ **then**
            **return**
        **else**
            error
        **end if**
    **else**
        error
    **end if**
**else**
    error
**end if**
```

5.3 Bottom-up syntax analysis

To systematically construct a bottom-up syntax analyzer we have:

1. Construction of the pilot graph: the pilot drives the bottom-up syntax analyzer. In each macro-state the pilot incorporates all the information about any possible phrase form that reaches the m-state (with look-ahead). Each m-state contains machine states with look-ahead, which are the characters we expect to see in the input at reduction time.
2. The m states are used to build a few analysis threads in the stack, which correspond to possible derivations: computations of the machine network, or paths with ε-arcs at each machine change, labeled with the scanned string.
3. Verification of determinism conditions on the pilot graph: shift-reduce conflicts, reduce-reduce conflicts, and convergence conflicts.
4. If the determinism test is passed, the bottom-up syntax analyzer can analyze the string deterministically.
5. The bottom-up syntax analyzer uses the information stored in the pilot graph and in the stack.

Definition. The *set of initials* is the set of chars found starting from state q_A of machine M_A of the net M:

$$\text{Ini}(q_A) = \text{Ini}(L(q_A)) = \{a \in \Sigma | a\Sigma^* \cap L(q_A) \neq \varnothing\}$$

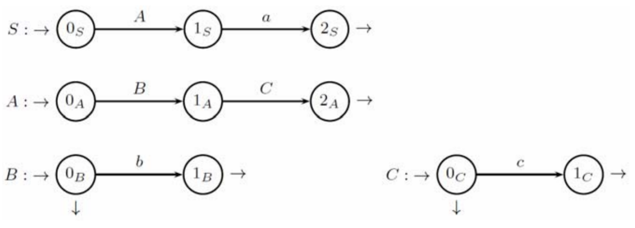
The elements of the initial set are defined in three possible cases:

- $a \in \text{Ini}(q_A)$ if exists an arc $q_A \xrightarrow{a} r_A$.
- $a \in \text{Ini}(q_A)$ if exists an arc $q_A \xrightarrow{B} r_A$ and $a \in \text{Ini}(0_B)$.
- $a \in \text{Ini}(q_A)$ if exists an arc $q_A \xrightarrow{B} r_A$ and $L(0_B)$ is nullable and $a \in \text{Ini}(r_A)$.

Example:
Consider the following grammar:

$$\begin{cases} S \to Aa \\ A \to BC \\ B \to b|\varepsilon \\ C \to c|\varepsilon \end{cases}$$

The corresponding machine net is:



To find the set of initials for S_0 we have to check the following states:

- $\text{Ini}(0_S) = \text{Ini}(0_A) \cup \text{Ini}(1_S)$ because $L(0_A)$ is nullable.

- $\text{Ini}(0_A) = \text{Ini}(0_B) \cup \text{Ini}(1_A)$ because $L(0_B)$ is nullable.
- $\text{Ini}(1_A) = \text{Ini}(0_C) \cup \text{Ini}(2_A)$ because $L(0_C)$ is nullable.

The final result is:

$$\text{Ini}(0_S) = \{b\} \cup \{c\} \cup \{a\}$$

Definition. An *item* is:

$$\langle q_B, a \rangle \text{ in } Q \times (\Sigma \cup \{\dashv\})$$

Two or more items with the same state can be grouped into one item. An item with a machine final state is said to be a reduction item.

Definition. The function *closure* computes a kind of closure of a set C of items with look-ahead.

To find the closure of C we have to apply this recursive clause until a fixed point is reached (the initial setting is $\text{closure}(C) = C$):

$$\langle 0_B, b \rangle \in \text{closure}(C) \text{ if } \begin{cases} \exists \text{ candidate } \langle q, a \rangle \in C \text{ and} \\ \exists \text{ arc } q \xrightarrow{B} r \text{ in } \mathcal{M} \text{ and} \\ b \in \text{Ini}(L(r)a) \end{cases}$$

Definition. The *shift operation* is defined as:

$$\begin{cases} \theta(\langle p_A, \rho \rangle, X) = \langle q_A, \rho \rangle \text{ if the arc } p_a \xrightarrow{X} q_a \text{ exists} \\ \text{The empty set otherwise} \end{cases}$$

A shift corresponds to a transition in a machine Y:

- if $X = c$ is a terminal symbol, then shift is a bottom-up syntax analyzer move that reads a char c in the input.
- If X is a non-terminal symbol, then shift is a bottom-up syntax analyzer ε-move after a reduction $z \to X$, and it does not read any input.
- Machine Y runs a transition with nonterminal label X.
- The analysis goes on after recognizing an input substring $z \in L(X)$ derivable from the nonterminal X.

The shift operation extends to sets of items (denoted as m-state):

$$\vartheta(C, X) = \bigcup_{\forall \gamma \in C} \vartheta(\gamma, x)$$

Pilot graph

Definition. The *pilot graph* is a deterministic finite state automaton, named $\mathcal{P}$, defined by the following entities:

- The set R of m-states.
- The pilot alphabet is the union $\Sigma \cup V$ of the terminal and non-terminal alphabets, to be also named the grammar symbols.
- The initial m-state, I_0, is the set $I_0 = closure(\langle 0_S, \dashv \rangle)$.
- The m-state set $R = I_0, I_1, \dots$ and the state-transition function $\theta : R \times (\Sigma \cup V) \rightarrow R$ are computed starting from I_0.

The construction of the pilot graph is incremental: it ends when nothing changes any longer. It has no final states since it does not recognize strings. The item in each m-state I of the pilot are parted into two groups:

- Base: contains the items obtained after a shift, which are non-initial states.
- Closure: contains the items obtained after a closure, which are initial states.

Definition. The *kernel* of an m-state I is the set of the m-states of I without look-ahead.

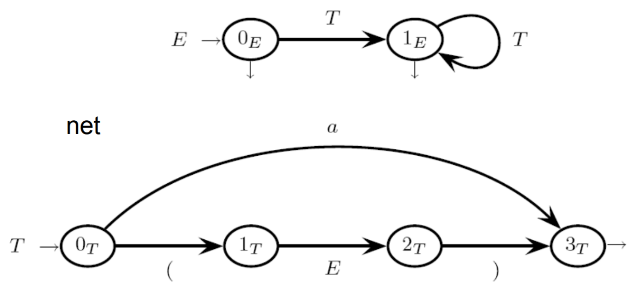
Algorithm 2 Pilot graph construction algorithm

```
$R' \leftarrow \{I_0\}$
while $R \neq R'$ do
    $R \leftarrow R'$
    for each m-state $I \in R$ and symbol $X \in \Sigma \cup V$ do
        $I' \leftarrow$ closure($\vartheta(I, X)$)
        if $I' \neq \varnothing$ then
            add arc $I \xrightarrow{X} I'$ to the graph of $\vartheta$
            if $I' \notin R$ then
                add m-state $I'$ to the set $R'$
            end if
        end if
    end for
end while
```

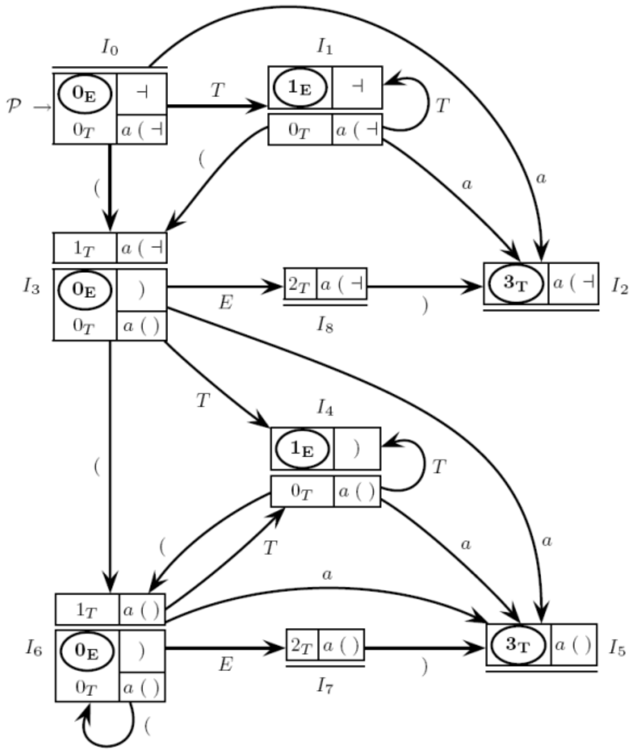
Example:
Consider the grammar:

$$\begin{cases} E \rightarrow T^* \\ T \rightarrow' ('E')'|a \end{cases}$$

The corresponding machine net is:



By applying the previous algorithm we obtain the following pilot graph:



It is possible to note that the kernel-equivalent m-states are:

$$(I_3, I_6), (I_1, I_4), (I_2, I_5), (I_7, I_8)$$

If an m-state contains an item with a final state, then the bottom-up syntax analyzer makes a reduction move. The look-ahead of the reduction item indicates the input characters expected at reduction time. The bottom-up syntax analyzer verifies the current input char *cc*, and:

- If $cc \in$ look-ahead, bottom-up syntax analyzer makes a reduction.
- Else bottom-up syntax analyzer reads input char *cc*, and makes a shift on an arc with label *cc*.
- Otherwise, bottom-up syntax analyzer stops and rejects.

ELR(1) method

The condition for allying the ELR(1) method are to not have:

- Shift-reduce conflicts: exists a reduction item with look-ahead that overlaps with the terminal symbols on the outgoing arcs. If there are some conflicts of this type the bottom-up syntax analyzer is unable to choose between shift and reduction.
- Reduce-reduce conflicts: exists two or more reduction items with look-ahead that overlap. If there are some conflicts of this type the bottom-up syntax analyzer is unable between the two possible reductions.
- Convergence conflicts: an m-state contains two or more items such that their two or more next states are defined for a symbol X (terminal or not). If there are some conflicts of this type the bottom-up syntax analyzer is unable the reduction between the reductions of the converging paths.

Definition. A multiple transition is *convergent* if:

$$\delta(p, X) = \delta(r, X)$$

A multiple convergent transition has a *conflict* if:

$$\pi \cap \rho \neq \varnothing$$

Bottom-up syntax analyzer's workflow

The bottom-up syntax analyzer follows this workflow:

1. The bottom-up syntax analyzer scans a string and executes a sequence of shift and reduction moves.
2. The bottom-up syntax analyzer pushes groups of items and starts from those in the initial pilot m-state.
3. Each m-state item becomes a 3-tuple by adding to it a backward-directed pointer that helps to reconstruct the different analysis threads constructed in parallel.
4. The bottom-up syntax analyzer decides whether to scan or reduce basing on the look-ahead in the pilot.
5. If the condition ELR (1) is satisfied, then the bottom-up syntax analyzer is deterministic.

Note that the length of the reduction handle is not fixed and so a rule may generate phrases of unbounded length. The reduction handle length is determined at reduction time by using the pointers. The pointer chain is followed backwards unto where the analysis thread started. A pointer value $\perp$ identifies a thread start point and all the closure items have a pointer value $\perp$, so these items are the start points of new threads. A 3-tuple with pointer different from $\perp$ continues an already started thread; the pointers different from $\perp$ are displayed as $\#i$; a pointer value $\#i$ means that the pointer is targeted to the i-th item (from top to bottom) in the previous stack element.

Computational complexity

When analyzing a string x of length $n = |x|$, the number of elements in the stack is greater or equal to $(n+1)$. To count the number of bottom-up syntax analyzer moves, we consider this contributes:

- Number of terminal shift, denoted as n_T.
- Number of nonterminal shift, denoted as n_N.
- Number of reductions, denoted as n_R.

These variables are linked in the following ways: the terminal shift corresponds to the length of the string and the number of nonterminal shift is the same as the number of reductions. As a result, we have that the total number of bottom-up syntax analyzer moves is:

$$n_T + n_N + n_R = n + 2n_R$$

Furthermore:

- The number of reductions with one or more terminals $(A \to a)$ is less or equal to n.
- The number of reductions of type null $(A \to \varepsilon)$ and copy $(A \to B)$ is linearly bounded by n.
- The number of reductions without any terminals $(A \to BC)$ is linearly bounded by n.

As a result we have that the final time complexity is $O(n) \leq kn + c$ for some integer constants k and c. Furthermore, the space complexity is the max stack size, which is $n_T + n_N \leq kn + c$. It is important to note that the space complexity is always upper-bounded by time complexity.

Bottom-up syntax analyzer implemented with a vectored stack

In an implementation of the bottom-up syntax analyzer with some programming language we can always mine the stack elements underneath the top one and directly look deep inside the stack. The third field of a stack item can be an integer that directly points back to the position of the stack element where the analysis thread begins.

Actually the analyzer is no longer a true bottom-up syntax analyzer as the stack alphabet becomes infinite. Such a variation is possible in every analyzer of practical interest and is not costly. In a closure item, write the current stack element index instead of the initial value $\perp$. In a base item, copy the same index value as that in the item before. So the bottom-up syntax analyzer does not scan back the reduction handle and goes directly to the origin.

5.4 Top-down syntax analysis

ELL(1) method

A grammar G represented as a machine net is ELL(1) if:

- It does not have any leftmost recursive derivations.
- Its pilot graph satisfies the ELR(1) condition.

- Its pilot graph does not have any multiple transitions (single transition property).

The ELL(1) condition above implies that the family of the ELL(1) grammars is contained in that of the ELR(1) grammars. Furthermore, also the family of ELL(1) languages is strictly contained in that of the ELR(1) languages:

$$\text{ELL}(1) \subset \text{ELR}(1)$$

Example:
Consider the grammar:

$$\begin{cases} E \rightarrow T^* \\ T \rightarrow' ('E')'|a \end{cases}$$

The pilot is ELR(1) as found before, it has the single transition property. Furthermore, it has no left recursion. As a result the grammar is also ELL(1).

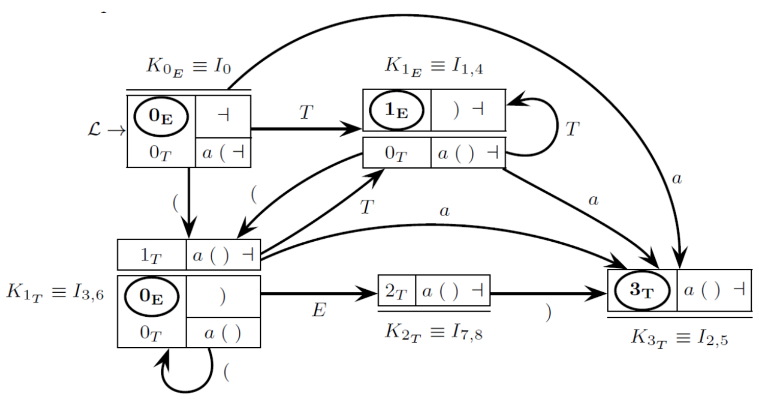
The ELL(1) analysis is more simple than the ELR(1). The main properties are:

- Predictive decision since we have only one item in each m-state base. As a result we know immediately which rule to apply.
- Stack pointer unnecessary once the rule to be applied is known. As a result we don't need to carry on more than one analysis thread, and it is unnecessary to keep the items corresponding to other analysis hypotheses.
- Contraction of the stack because we have only one analysis thread. As a result we don't need to push on stack the state path followed, and it suffices to push on stack the sequence of machines followed.
- Simplification of the pilot: m-states with same kernel are unified (unify look-ahead). Transitions with same label from kernel-identical m-states go into kernel-identical m-states The m-state bases (with non-initial states) contain only one item (consequence of STP); thus, they are in a one-to-one correspondence with the non-initial states of the machine net.

Example:
Consider the grammar:

$$\begin{cases} E \rightarrow T^* \\ T \rightarrow' ('E')'|a \end{cases}$$

We have shown that it is both ELR(1) and ELL(1), so we can construct the simplified version of the pilot which is:

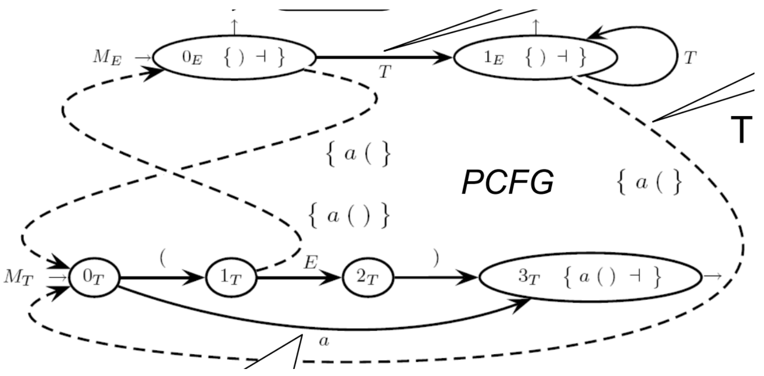


Parser control-flow graph

The parser control-flow graph (PCFG) is the control unit of the ELL(1) syntax analyzer. The prospect sets are included only in the final states to choose whether to exit the machine or to continue with some more moves. The call arcs are dashed and labeled with a guide set, that is the set of characters that are expected in the input soon after calling the machine. The guide set allows choosing whether:

- Executing one call move.
- Scanning a terminal symbol.
- Executing one of two or more call moves.
- Exiting the machine (if final state).

Example:
Consider the pilot of the previous example. The corresponding parser control-flow graph is as follows:



The character b is in the guide set, denoted as $b \in \text{Gui}(q_A \dashrightarrow 0_{A_1})$, if one of the following properties holds:

- $b \in \text{Ini}(L(0_{A_1}))$.
- A_1 is nullable and $b \in \text{Ini}(L(r_A))$.
- A_1 and $L(r_A)$ are both nullable and $b \in \pi_{r_A}$.

- Exists in $\mathcal{F}$ a call arc $0_{A_1} \overset{\gamma_2}{\dashrightarrow} 0_{A_2}$ and $b \in \gamma_2$.

The guide sets of the call arcs that depart from the same state have to be disjoint from one another, and be disjoint from all the scan arcs from the same state.

In a PCFG almost all the arcs (except the non-terminal shift) are interpreted as conditional instructions:

- Terminal arcs $p \xrightarrow{a} q$ run if an only if the current character $cc = a$.
- Call arcs $q_A \to 0_B$ run if an only if the current character is $cc \in \text{Gui}(q_A \to 0_B)$.
- Exit arcs (darts) $f_A \to$ from a state with an item $\langle f_A, \pi \rangle$ run if and only if the current character is $cc \in \pi$.
- The non-terminal arcs $p \xrightarrow{a} q$ are interpreted as (unconditioned) return instructions from a machine.

Property 5.4.1. If the guide sets are disjoint, then the condition for ELL(1) is satisfied.

Algorithm

The stack elements are the states of the PCFG. The stack is initialized with element $\langle 0_E \rangle$. Suppose $\langle q_A \rangle$ is the stack top (it means that machine M, is active and in the state q_A). The ELL syntax analyzer has four move types:

- Scan move: if the shift arc $q_A \xrightarrow{cc} r_A$ exists, then scan the next token and replace the stack top by $\langle r_A \rangle$ (the active machine does not change).
- Call move: if there exists a call arc $q_A \overset{\gamma}{\dashrightarrow} 0_B$ such that $cc \in \gamma$, let $q_A \xrightarrow{B} r_A$ be the corresponding nonterminal shift arc; then pop, push element $\langle r_A \rangle$ and push element $\langle 0_B \rangle$.
- Return move: if q_A is a final state and token cc is in the prospect set associated with q_A, then pop.
- Recognition move: if M_A is the axiom machine, q_A is a final state and $cc = \dashv$, then accept and halt.

In any other case the analyzer stops and rejects the input string.

Parser implementation by means of recursive procedures

With this implementation the machine becomes a procedure without parameters and, consequently, we have one procedure per each nonterminal. A call move become a procedure call, a can move become a call procedure next, and a return move become a return from procedure. Parser Control Flow Graph becomes the control graph of the procedure.

Use the guide sets and the prospect sets to choose which move to execute. The analysis starts by calling the axiomatic procedure.

Direct construction of the parser control-flow graph

It is possible to construct the PCFG without building the full ELR pilot graph. To do this we simply put call arcs on the machine net and annotate the net with the prospect and guide sets.

To build the prospect set we have to distinguish between initial states and other states. For the initial states 0_A we have that:

$$\pi_{0_A} := \pi_{0_A} \cup \bigcup_{q_i \xrightarrow{A} r_i} \left(\mathrm{Ini}(L(r_i)) \cup \text{ if } \mathrm{Null}(L(r_i)) \text{ then } \pi_{q_i} \text{ else } \varnothing\right)$$

For every non-initial state we have:

$$\pi_q := \bigcup_{p_i \xrightarrow{X_i} q} \pi_{p_i}$$

To build the guide set we have to apply iteratively this formula:

$$\mathrm{Gui}(q_A \dashrightarrow 0_{A_1}) := \bigcup \begin{cases} \mathrm{Ini}(L(A_1)) \\ \text{If } \mathrm{Null}(A_1) \text{ then } \mathrm{Ini}(L(r_A)) \text{ else } \varnothing \\ \text{If } \mathrm{Null}(A_1) \wedge \mathrm{Null}(L(r_A)) \text{ then } \mathrm{Ini}(\pi_(r_A)) \text{ else } \varnothing \\ \bigcup_{0_{A_1} \dashrightarrow 0_{B_i}} \mathrm{Gui}(0_{A_1} \dashrightarrow 0_{B_i}) \end{cases}$$

Furthermore (for the final darts and the terminal shift arcs):

$$\mathrm{Gui}(f_A \to) := \pi_{f_A}$$

$$\mathrm{Gui}(q_A \xrightarrow{a} r_A) := \{a\}$$

Modification of grammar not in ELL(1)

If the ELL(1) condition is not verified, we can try to modify the grammar and make it of type ELL(1). This approach may take a long time and be a hard work.

The alternative approach consists in using a longer look-ahead: the analyzer looks at a number $k > 1$ of consecutive characters in the input. If the guide sets of length k on alternative moves are disjoint, then the ELL(k) analysis is possible.

5.5 Syntax analysis of nondeterministic grammars

5.6 Syntax analysis of nondeterministic grammars

The Earley method, also known as the **tabular** method, is applicable to grammars of any type, including those that are ambiguous and non-deterministic. This technique constructs all potential derivations of the scanned string prefix concurrently, eliminating the need for explicit look-ahead implementation. Additionally, it operates without a stack, utilizing a vector of sets to maintain the current parsing state.

When analyzing a string $x = x_1 x_2, \dots, x_n$ or $x = \varepsilon$ with a length $|x| = n \geq 0$, the algorithm employs a vector $E\,[0, \dots, n]$ consisting of $n+1$ elements. Every element $E[i]$ is a set of pairs $\langle q_\oplus, j\rangle$, where:

- $q_\oplus$ is a state of machine $M_\oplus$.

- j is an integer pointer indicating the element $E[i]$ (with $0 \leq i \leq j \leq n$) that precedes or corresponds to $E[j]$. It contains a pair $\langle 0_{\oplus}, j \rangle$:
  - $0_{\oplus}$ belongs to the same machine as $q_{\oplus}$.
  - j marks the position in the input string x from which the current derivation of $\oplus$ started, represented by $\uparrow$.
  - If $j = i$, the string is empty ε.

A pair $\langle q_{\oplus}, j \rangle$ is classified as:

- *Initial* if $q_{\oplus} = 0_{\oplus}$.
- *Final* if $q_{\oplus} \in F_{\oplus}$.
- *Axiomatic* if $\oplus = S$.

5.6.1 Earley's Method

To begin, the Earley vector undergoes initialization, wherein all elements $E[1], \ldots, E[n]$ are set to the empty set ($E[i] = \emptyset \, \forall 1 \leq i < n$), and the first element $E[0]$ is assigned the set containing the initial pair $\langle 0_S, 0 \rangle$. In this pair, the state 0_S corresponds to the initial state of the machine M_S associated with the start symbol S, and the integer pointer 0 denotes the position in the input string from which the current derivation of S originates. As parsing advances, with the current character being x_i, the current element $E[i]$ becomes populated with one or more pairs. The final Earley vector encompasses all conceivable derivations of the input string. Three distinct operations are applicable to the current element of the vector $E[i]$: closure, terminal shift and nonterminal shift. These operations mirror those of the analogous ELR(1) parser.

Closure This operation is applicable to a pair originating from a state with an arc labeled with a nonterminal $\oplus$. Let $\langle p, j \rangle$ be a pair in the element $E[i]$, and suppose there exists an arc $p \xrightarrow{\oplus} q$ with a nonterminal label $\oplus$ and a (non-relevant) destination state q. The operation adds a new pair $\langle 0_{\oplus}, i \rangle$ to the same element $E[i]$: the state of this pair is the initial one $0_{\oplus}$ of machine $M_{\oplus}$ of that nonterminal $\oplus$, and the pointer has value i, which means that the pair is created at step i starting from a pair already in the element $E[i]$. The effect of this operation is to add the current element $E[i]$ to all the pairs with the initial states of the machines that can recognize a substring starting from the next character x_{i+1} and ending at the current character x_i.

Terminal Shift Applies to a pair with a state from where a terminal shift arc originates. Suppose that arc $\langle p, j \rangle$ is in element $E[i-1]$ and that the net has arc $p \xrightarrow{x_i} q$ labelled by the current token x_i. The operation writes into element $E[i]$ the pair $\langle q, j \rangle$, where the state is the destination of the arc and the point equals that of the pair in $E[i-1]$, to which the terminal shift arc is attached. The next token will be x_{i+1} (the first one after the current).

Nonterminal Shift This operation is triggered by the presence of a final pair $\langle f_\oplus, j\rangle$ in the current element $E[i]$, where $f_\oplus \in F_\oplus$ is a final state of machine $M_\oplus$ of nonterminal $\oplus$; such a pair is called enabling. In order to shift, it's necessary to locate the element $E[j]$ and shift the corresponding nonterminal: the parser searches for a pair $\langle p, l\rangle$ such that the net contains an arc $p \xrightarrow{\oplus} q$, with a label that matches the machine of state $f_\oplus$ in the enabling pair. The pointer l is in the interval $[0, j]$. The operation will certainly find at least one such pair and the nonterminal shift applies to it. Then the operation writes the pair $\langle q, l\rangle$ into $E[i]$; if more than one pair is found, the operation is applied to all of them.

Earley's Algorithm The algorithm for EBNF grammars makes use of two procedures, called `completion` and `terminalshift`. The input string is denoted by $x = x_1 x_2 \ldots x_n$, where $|x| = n \geq 0$ *(if* $n = 0$*, then* $x = \varepsilon$*)*.

Algorithm 3 Completion(E, i)

```
1: while some pair has been added do
2:     for each pair $\langle p, j\rangle \in E[i]$ and $X, q \in V$ such that $p \xrightarrow{X} q$ do
3:         add pair $\langle 0_X, i\rangle$ to element $E[i]$
4:     end for
5:     for each pair $\langle f, j\rangle \in E[i]$ and $X \in V$ such that $f \in F_X$ do
6:         for each pair $\langle p, l\rangle \in E[j]$ and $q \in Q$ such that $p \xrightarrow{X} q$ do
7:             add pair $\langle q, l\rangle$ to element $E[i]$
8:         end for
9:     end for
10: end while
```

The completion procedure adds new pairs to the current vector element $E[i]$ by applying the closure and nonterminal shifts as long as new pairs are added. The outer loop (`do-while`) is executed at least once because the closure operation is always applied. Finally, note that this operation processes the nullable nonterminals by applying to them a combination of closures and nonterminal shifts.

Algorithm 4 TerminalShift(E, i)

```
1: for each pair $\langle p, j\rangle \in E[i-1]$ and $q \in Q$ such that $p \xrightarrow{x_i} q$ do
2:     add pair $\langle q, j\rangle$ to element $E[i]$
3: end for
```

The `terminalshift` procedure adds to the current vector element $E[i]$ all the pairs that can be reached from the pairs in $E[i-1]$ by a terminal shift, scanning token x_i, $1 \leq 1 \leq n$. It may fail to add any pair to the element, that will remain empty; a nonterminal that exclusively generates the empty string ε never undergoes a terminal shift. Finally, notice that the procedure works correctly even when the element $E[i]$, or its predecessor $E[i-1]$, is empty.

Algorithm 5 Early method algorithm

```
1: $E[0] \leftarrow \{\langle 0_S, 0\rangle\}$
2: for $i = 1$ to $n$ do
3:     $E[i] \leftarrow \varnothing$
4: end for
5: $Completion(E, 0)$
6: $i \leftarrow 1$
7: while $i \leq n \wedge E[i-1] \neq \varnothing$ do
8:     $TerminalShift(E, i)$
9:     $Completion(E, i)$
10:    $i++$
11: end while
```

The algorithm can be summarized into the following steps:

1. The initial pair $\langle 0_S, 0\rangle$ is added to the first element $E[0]$ of the vector.
2. The elements $E[1]$ to $E[n]$ (if present) are initialized to the empty set.
3. $E[0]$ is completed.
4. If $n \geq 1$ (if the string x is not empty), the algorithm puts pairs in the current element $E[i]$ through `terminalshift` and finishes element $E[i]$ through `completion`. If `terminalshift` fails to add any pair to $E[i]$, the element remains empty.
5. The loop iterates as far as the last element $E[n]$, terminating when the vector is finished or the previous element $E[i-1]$ is empty.

Property 5.6.1 (Acceptance condition)**.** When the Earley algorithm terminates, the string x is accepted if and only if the last element $E[n]$ of vector E contains a final axiomatic pair $\langle f_S, 0\rangle$, with $f_S \in F_S$

Complexity of Earley's Algorithm Assuming that each basic operation has cost $O(1)$, that the grammar is fixed and x is a string, the overall complexity of the algorithm can be calculated by considering the following contributes:

1. A vector element $E[i]$ contains several pairs $\langle q, j\rangle$ that are linearly limited by i, as the number of states in the machine net is constant and $j \leq i$. As such, the number of pairs in $E[i]$ is bounded by n:
$$|E(i)| = O(n)$$
2. For a pair $\langle p, j\rangle$ checked in the element $E[i-1]$, the terminal shift operation adds one pair to $E[i]$. As such, for the whole $E[i-1]$, the `terminalshift` operation needs no more than n steps:
$$\texttt{terminalshift} = O(n)$$
3. The `completion` procedure iterates the operations of closure and nonterminal shift as long as they can add some new pair. Two operations can be performed on the whole set $E[i]$:

(a) For a pair $\langle q, j \rangle$ checked in $E[i]$, the closure adds to $E[i]$ a number of pairs limited by the number $|Q|$ of states in the machine net, or $O(1)$. For the whole $E[i]$, the closure operation needs no more than n steps:

$$\texttt{closure} = O(n) \times O(1) = O(n)$$

(b) For a final pair $\langle f, j \rangle$ checked in $E[i]$, the nonterminal shift first searches pairs $\langle p, l \rangle$ for a certain p through $E[j]$, with size $O(n)$, and then adds to $E[i]$ as many pairs as it found, which are no more than $E[j] = O(n)$. For the whole set $E[i]$, the `completion` procedure needs no more than $O(n^2)$ steps:

$$\texttt{completion} = O(n) + O(n^2) = O(n^2)$$

4. By summing up the numbers of basic operations performed in the outer loop for $i = 1 \ldots n$, the overall complexity of the algorithm is:

$$\texttt{terminalshift} \times n + \texttt{completion} \times (n+1) = O(n) \times n + O(n^2) \times (n+1) = O(n^3)$$

As such, the following property holds:

Property 5.6.2 (Complexity of Earley's algorithm)**.** The asymptotic time complexity of the Earley algorithm in the worst case is $O(n^3)$, where n is the length of the string analyzed.

5.6.2 Syntax tree construction

The next procedure, `buildtree` *(or BT)* builds the syntax tree of an accepted string x by using the Earley vector E as a guide, under the assumption that the grammar is unambiguous. The tree is represented by a parenthesized string, where two matching parentheses delimit a subtree rooted at some nonterminal node. Given an EBNF grammar $G = (V, \Sigma, P, S)$, machine net $\mathcal{M}$, and a string x of length $n \geq 0$ that belongs to language $L(G)$, suppose that its Earley vector E with $n+1$ elements is available. Function `buildtree` is recursive and has four formal parameters:

- Nonterminal $\oplus \in V$, root of the tree to be built.
- State f, final for the machine $M_\oplus \in \mathcal{M}$. f is the end of the computation path in $M_\oplus$ that corresponds to analyzing the substring generated by $\oplus$.
- two non-negative integers i and j. i and j always respect the condition $0 \leq i \leq j \leq n$. They respectively represent the start and end of the substring generated by $\oplus$:

$$\begin{cases} \oplus \underset{G}{\overset{+}{\Rightarrow}} x_{j+1} \ \ldots \ x_i & \text{if } j < i \\ \oplus \underset{G}{\overset{+}{\Rightarrow}} \varepsilon & \text{if } j = i \end{cases}$$

Grammar G admits derivation $S \overset{+}{\Rightarrow} x_1, \ldots, x_2$ or $S \overset{+}{\rightarrow} \varepsilon$ and the Earley algorithm accepts E; as such, the element $E[n]$ of the vector E contains a final axiomatic pair $\langle f_S, 0 \rangle$, with $f_S \in F_S$.

To build the tree of string x with root node S, procedure `buildtree` is called with parameters $(S, f_S, 0, n)$; then it builds recursively all the subtrees and will assemble them in the final tree.

Algorithm 6 buildtree($\oplus, f, i, j$)

```
1: $C := \varepsilon$
2: $q := f$
3: $k := i$
4: **while** $q \neq 0_\oplus$ **do**
5:     **if** $\exists$ $h = k - 1$ and $\exists$ p $\in Q_\oplus$ such that $\langle p, j \rangle \in E[h]$ and net has p $\xrightarrow{x_k}$ q **then**
6:         $C := C \cup x_k$
7:     **end if**
8:     **if** $\exists Y \in V$ and $\exists e \in F_Y$ and $\exists h, j (j \leq h \leq k \leq i)$ and $\exists p \in Q_\oplus$ such that $(\langle p, j \rangle \in E[k]$
   and $\langle e, h \rangle \in E[h]$ and net has $p \xrightarrow{Y} q)$ **then**
9:         $C := C \cup$ buildtree($Y, q, k, j$)
10:    **end if**
11:    $q := p$
12:    $k := h$
13: **end while**
14: **return** $C(x)$
```

Essentially, `buildtree` walks back on a computation path in machine $M_\oplus$ and jointly scans back the Earley vector E from $E[n]$ to $E[0]$; during the walk, it recovers the terminal and nonterminal shift operations to identify the **children of the same node** $\oplus$. In this way, the procedure reconstructs in reverse order the shift operations performed by the Earley parser. The `while` loop runs zero or more times, recovering **.one shift per iteration** The **first** condition in the loop recovers a **terminal shift** appending the related leaf to the tree, while the **second** one recovers a **nonterminal shift** and recursively calls itself to build the subtree of the related nonterminal node. State `e` is final for machine M_Y, and inequality $0 \leq h \leq k \leq i$ is guaranteed by the definition of the Earley vector E. If the parent nonterminal $\oplus$ immediately generates the **empty string** *(as there exists a rule $\oplus \rightarrow \varepsilon$)*, the **leaf** ε **is the only child** and the loop does not run again. Function `buildtree` uses two local variables in the `while` loop the current state q of the machine $M_\oplus$ and the current index `k` of the Earley vector element $E[k]$, both updated at each iteration: initially, q is **final**; at the end of the algorithm, q is the **initial** state $0_\oplus$ of the machine $M_\oplus$. At each iteration, the current state q is shifted to the previous state p and the current index k is shifted from i to j, through jumps of different lengths. Sometimes k may stay in the same position: this happens if and only if the function processes a series of nonterminals that end up generating the **empty string** ε. The two `if` conditions are **mutually exclusive** if the grammar is **not ambiguous**: the first one is true if the child is a leaf; the other is true if the child has its own subtree.

Computation complexity of `buildtree` Assuming that the grammar is unambiguous, clean and devoid of circular derivations, for a string of length n the number of tree nodes is linearly bounded by n. The basic operations are those of checking the state or the pointer of a pair, and of concatenating a leaf or a node to the tree; both of them are executed in constant time. The total complexity can be estimated as follows:

1. A vector element $E[k]$ contains a number of pairs of magnitude $O(n)$
2. There are between 0 and k elements of E
3. Checking the condition of the first `if` statement requires a constant time *($O(1)$)*; the possible enlisting of one leaf takes a constant time *($O(1)$)*. Processing the whole $E[k-1]$

takes a time of magnitude

$$O(n) \times O(1) + O(1) = O(n)$$

4. Checking the condition of the second `if` statement requires a linear time *($O(n)$)*, due to the search process; the possible enlisting of one node takes a constant time *($O(1)$)*. Processing the whole $E[k]$ takes a time of magnitude

$$O(n) \times O(n) + O(1) = O\left(n^2\right)$$

Finally, since the total number of terminal plus nonterminal shifts to be recovered is linearly bounded by the numbers of nodes to be built, the total complexity of the algorithm is:

$$\left(O(n) + O\left(n^2\right)\right) \times O(n) = O\left(n^3\right)$$

Computational complexity reduction via Earley vector ordering Since the `buildtree` procedure does not write the Early vector, it's possible to reduce its complexity by reordering the vector E in a way that the `while` loop runs fewer times. Suppose that each element $E[k]$ is ranked according to the value of its pointer: this operation is done in $(n+1)O\left(n \log(n)\right) = O\left(n^2 \log(n)\right)$. Now the second `if` statement requires a time $O(n)$ to find a pair with final state e and pointer h in the stack, while searching the related pair with a fixed pointer j takes $O(n)$ time. Similarly, the first `if` statement will only require a time $O\left(\log(n)\right)$. The total time complexity of the algorithm is:

$$O\left(n^2 \log(n)\right)$$

Optimization via look-ahead The items in the sets $E[k]$, $\forall\, k$ of the Earley vector can be extended by including a look-ahead, computed in the same way as ELR(1) parsers: by siding a look-ahead each time, the Earley algorithm avoids putting into each set the items that correspond to choices that cannot succeed. This technique may cause an increase in the number of items in the vector itself for some grammars, and as such its use is not always beneficial.

Application of the algorithm to ambiguous grammars Ambiguous grammars deserve interest in the processing of natural languages: the difficult part is representing all the possible syntax trees related to a string, the number of which can grow exponentially with respect to its length. This can be done by using a **Shared Packed Parse Forest** *(SPPF)*, a graph type more general than the tree built that still takes a worst-case cubic time for building.

CHAPTER 6

Syntax and semantic translation

6.1 Introduction

In a general sense, a translation refers to a function (or mapping) from a source language to a target language. Typically, two approaches are employed: through coupled grammars or using a transducer (an automaton similar to an acceptor with the ability to produce output). These methods fall under the category of purely syntactic translations, expanding and completing the language definition and parsing methods studied so far. However, they do not inherently provide a semantic interpretation (meaning) of the input string. In contrast, an alternative method known as the attribute grammar model capitalizes on the syntactic modularity of grammar rules. The distinction between syntax and semantics lies in the former dealing with the form of a sentence, while the latter is concerned with its meaning. In computer science, this difference is reflected in the domain of entities and operations permitted by each:

- Syntax employs concepts and operations from formal language theory, representing algorithms as automata. Entities include alphabets, strings, and syntax trees, with operations such as concatenation, union, intersection, and complementation.
- Semantics utilizes concepts and operations from logic, representing algorithms as programs. Entities in semantics are not as limited as in syntax, encompassing numbers, strings, and any data structures. The complexity level is higher, with operations extending beyond formal language theory.

6.2 Syntactic translation

Definition (*Transducer*)**.** The transducer is an automaton that can produce output via an output function.

Definition (*Translation*)**.** The translation is the correspondence between two texts that have the same meaning in two different languages: the given text language is called source, and it's denoted by Σ while the other language is called target, and it's denoted by Δ.

Definition (*Purely syntactic method*)**.** A purely syntactic method is a method that applies local transformations to the source text, without considering its meaning.

The translation grammar is a purely syntactic translation method that uses pushdown transducers on top of a parsing method. The translation regular expression is a purely syntactic translation method that uses a finite transducer enriched with an output function.

Definition (*Syntax directed method*)**.** A syntax directed method is a semiformal approach based on a combination of syntax rules and semantic functions.

Definition (*Syntax directed semantic translation*)**.** The syntax directed semantic translation is a semiformal approach based on a combination of syntax rules and semantic functions

Definition (*Syntactic representation*)**.** The syntax directed semantic translation is a semiformal approach based on a combination of syntax rules and semantic functions

The procedure for syntax-directed semantic translation involves the following steps:

1. Define a set of attributes for nonterminals in the program.
2. Establish a set of semantic equations governing how attributes are evaluated.
3. Specify the order in which equations should be evaluated.
4. Construct a parse tree that captures the syntactic structure of the program.
5. Traverse the tree in the order of attribute evaluation.
6. Use equations to compute the specified attributes.

Analogies of Formal Language and translation Theory The set of language phrases aligns with the set of pairs (comprising source and destination strings) that characterize the translation relation.

The language grammar transforms into a translation grammar, responsible for generating pairs of phrases.

Similarly, the finite state automaton or pushdown automaton transforms into a transducer automaton or a syntax analyzer, computing the translation relation.

6.2.1 Translation relation and function

Translation can be defined formally as a binary relation between the source and the target universal languages, denoted as Σ^* and Δ^*, respectively. This relation is essentially a function whose domain is a subset of the Cartesian product $\Sigma^* \times \Delta^*$. A translation relation ρ is a set of pairs of strings (x, y) where $x \in \Sigma^*$, $y \in \Delta^*$, satisfying:

$$\rho = \{(x, y), \dots\} \subseteq \Sigma^* \times \Delta^*$$

Here, y is the image (or translation, destination) of the source string x, and they correspond to each other. Given a translation relation ρ, the source and target languages L_1 and L_2 are defined as:

$$L_1 = \{x \in \Sigma^* | \exists y \in \Delta^* \text{ such that } (x, y) \in \rho\}$$
$$L_2 = \{y \in \Delta^* | \exists x \in \Sigma^* \text{ such that } (x, y) \in \rho\}$$

Alternatively, translation can be formalized using the set of all images of a source string, defining a function τ that maps each source string to the set of corresponding target strings:

$$\tau : \Sigma^* \to \wp(\Delta^*)$$

$$\tau(x) = \{y \in \Delta^* \,|\, (x, y) \in \rho\}$$

The union of the application of the function τ to all source strings is the target language L_2:

$$L_2 = \bigcup_{x \in L_1} \tau(x)$$

The translation relation ρ is partially defined, as some strings in the source alphabet might lack a translation in the target alphabet. To make the function total, a special value, `error`, is introduced where the application of the function τ to a string x is undefined. In cases where each source string has at most one image, the translation function is defined as:

$$\tau : \Sigma^* \to \Delta^*$$

This case is crucial for defining the inverse translation τ^{-1}:

$$\tau^{-1} : \Delta^* \to \wp(\Sigma^*) \qquad \tau^{-1}(y) = \{x \in \Sigma^* | y \in \tau(x)\}$$

Similarly, the inverse translation relation ρ^{-1} is obtained by swapping the corresponding source and target strings:

$$\rho^{-1} = \{(y, x) \,|\, (x, y) \in \rho\}$$

The mathematical properties of τ give rise to different types of translations:

1. *Total*: every source string has one or more images.
2. *Partial*: one or more source strings lack any image.
3. *Single-valued*: no string has two distinct images.
4. *Multivalued*: one or more source strings have more than one image.
5. *Injective*: distinct source strings have distinct images. An alternative definition is that every target string corresponds to at most one source string, making the inverse translation τ^{-1} single-valued.
6. *Surjective*: the image of the translation coincides with the range; every string over the target alphabet is the image of at least one source string. In this case, the inverse translation τ^{-1} is total.
7. *Bijective*: the translation is both injective and surjective. Only in this case is the inverse translation bijective as well.

6.2.2 Transliteration

Transliteration, also known as alphabetic homomorphism, represents the most straightforward type of translation. In this process, each source character is transliterated into a target character or string. The translation defined by an alphabetic homomorphism is single-valued, although this property may not necessarily extend to the inverse function. Consider any Greek letter, denoted as x. Then, for the inverse transliteration:

$$h^{-1}(x) = \{\alpha, \ldots, \omega\}$$

If the homomorphism erases a letter (mapping it to the empty string), the inverse translation becomes multivalued. This is because any string composed of erasable characters can be inserted at any position in the text. If the inverse function is also single-valued, the transliteration becomes bijective, allowing the reconstruction of the source string from a given target string. Transliteration transforms a letter into another without considering the occurrence context.

Purely Syntactic Translation In the context of a source language defined by a grammar, every syntactic component (e.g., a subtree) of the source language is individually mapped to an equivalent component in the target language.

Definition (*Translation grammar*)**.** A translation grammar $G_T = (V, \Sigma, \Delta, P, S)$ is a context-free grammar with a terminal alphabet $C \subseteq \Sigma^* \Delta^*$ of pairs (u, v) of source/target strings, also expressed as fractions $\frac{u}{v}$.

Definition (*Translation relation*)**.** The translation relation ρ_G defined by grammar G_τ is:

$$\rho_{G_\tau} = \{(x, y) \, | \exists, z \in L\left(G_\tau\right) \wedge x = h_\Sigma\left(z\right) \wedge y = h_\Delta\left(z\right)\}$$

where $h_\epsilon : C \to \Sigma$ and $h_\Delta : \to \Delta$ are the homomorphism that map each pair (u, v) to the corresponding source/target string.

6.2.3 Ambiguity of source grammar and translation

As previously observed, when the source grammar is ambiguous, a sentence may have two distinct syntax trees, each corresponding to a different target syntax tree. Even in cases where the source grammar is unambiguous, the translation can still be multivalued if different target rules are associated with the same source rule in the translation scheme.

Property 6.2.1 (Unambiguity conditions for translations)**.** Let $G_\tau = (G_1, G_2)$ be a translation grammar such that the source grammar G_1 is unambiguous, and that no two rules of target grammar G_2 correspond to the same rule of source grammar G_1.

When these conditions are satisfied, the translation specified by grammar G_τ is single-valued, ensuring the definition of a valid translation function.

6.2.4 Translation grammar and pushdown automata

Similar to the recognizer for context-free grammar, a transducer implementing a translation grammar necessitates a stack of unbounded length. A pushdown transducer, also known as an IO automaton, extends the concept of a pushdown automaton by allowing the output of zero or more characters at each move.

A IO automaton is represented by a 8-tuple $(Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F)$:

- Q is a finite set of states.
- Σ is the source alphabet (input).
- Γ is the pushdown stack alphabet.
- Δ is the target alphabet (output).
- δ is the state transition and output function:

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to Q \times \Gamma^* \times \Delta^*$$

- $q_0 \in Q$ is the initial state.
- $Z_0 \in \Gamma$ is the initial stack symbol.

- $F \subseteq Q$ is the set of final states.

The meaning of the function is as follows: if the current state, input character, and stack top are respectively:

$$q^{'}, a, Z$$

and it holds:

$$\delta\left(q^{'}, a, Z\right) = \left(q^{''}, \gamma, y\right)$$

then the automaton reads a from the input and Z from the stack top, enters the state $q^{''}$, pushes the string γ onto the stack, and outputs the string y. If the automaton recognizes the string by an empty stack, then the set of final states coincides with Q. The underlying automaton of the translator is obtained by removing the target alphabet symbols and the output string actions from the definitions.

Instantaneous configuration The instantaneous configuration of the pushdown transducer is defined as a 4-tuple $(q, y, \eta, z) \in (Q \times \Sigma^* \times \Gamma^* \times \Delta^*)$ where:

- q is the current state.
- y is the remaining portion (suffix) of the source string x to be read.
- η is the stack content.
- z is the string written to the output tape, up to the current configuration.

Moves When a move is executed, a transition from a configuration to the next one occurs, denoted as:

$$(q, x, \eta, \omega) \mapsto (p, y, \lambda, z)$$

A computation is a chain of zero or more transitions, denoted by $\overset{*}{\mapsto}$.

Current configuration	Next configuration	Applied move
$(q, ax, \eta Z, z)$	$(p, x, \eta\gamma, zy)$	reading move $\delta(q, a, Z) = (p, \gamma, y)$
$(q, ax, \eta Z, z)$	$(p, ax, \eta\gamma, zy)$	spontaneous move $\delta(q, \varepsilon, Z) = (p, \gamma, y)$

The initial configuration and string acceptance conditions are the same as described for pushdown automata, and the computed translation τ is defined as follows (assuming that the acceptance condition is by a final state):

$$\tau(x) = z \Leftrightarrow (q_0, x, Z_0, \varepsilon) \overset{*}{\mapsto} (q, \varepsilon, \lambda, z) \quad \text{with } q \in F, \lambda \in \Delta^*$$

6.2.5 From translation grammar to pushdown transducer

Translation schemes and pushdown transducers are two models for representing language transformations: the former is suitable for specifying the translation, while the latter is suitable for its implementation.

Property 6.2.2 (Equivalence of translation grammar and pushdown transducer)**.** A translation relation is defined by a translation grammar or scheme if, and only if, it is computed by a (eventually nondeterministic) pushdown transducer.

Normalization of Translation Rules To simplify the rules, the following hypotheses are made for the source/target pairs $\frac{u}{v}$ occurring in the rules, where $u \in \Sigma^*$ and $v \in \Delta^*$:

1. For any pair $\frac{u}{v}$ it holds $|u| \leq 1$:
   - The source u is a single character $a \in \Sigma$ or the empty string ε.
   - The rule $\frac{a_1 a_2}{v}$ can be replaced by $\frac{a_1}{v}\frac{a_2}{\varepsilon}$.
2. No rule may contain the following substrings:

$$\frac{\varepsilon}{v_1}\frac{a}{v_2} \quad \text{or} \quad \frac{\varepsilon}{v_1}\frac{\varepsilon}{v_2}$$

with $v_1, v_2 \in \Delta^*$. If such combinations are present in a rule, they can be respectively replaced by the equivalent pairs:

$$\frac{a}{v_1 v_2} \quad \text{or} \quad \frac{\varepsilon}{v_1 v_2}$$

Predictive pushdown transducer construction algorithm Let C represent the set of pairs, including those of the form $\frac{\varepsilon}{v}$ with $v \in \Delta^+$ and $\frac{b}{w}$ with $b \in \Sigma$ and $w \in \Delta^*$, occurring in some rule of the translation grammar. Initially, the stack contains the axiom S, and the reading head is positioned at the first character of the source string. At each step, the automaton non-deterministically selects an applicable rule and performs the corresponding move. It's important to note that this automaton doesn't necessarily require states, although they may be introduced later to enhance its execution.

#	*rule*		*comment*
1	$A \to \frac{\varepsilon}{v} B A_1 \ldots A_n$ $n \geq 0$ $v \in \Delta^+$, $B \in V$ $A_i \in (C \cup V)$	if *top* $= A$ then *write* (v) *pop* *push* $A_n \ldots A_1 B$	*emit* the target string v *push* on stack the prediction string $BA_1 \ldots A_n$
2	$A \to \frac{b}{w} B A_1 \ldots A_n$ $n \geq 0$ $b \in \Sigma$, $w \in \Delta^*$ $A_i \in (C \cup V)$	if $cc = b \wedge$ *top* $= A$ then *write*(w) *pop* *push* $(A_n \ldots A_1)$ advance the reading head	char b was next expected and has been read *emit* the target string w *push* the prediction string $A_1 \ldots A_n$
3	$A \to BA_1, \ldots A_n$ $n \geq 0$ $b \in \Sigma$, $w \in \Delta^*$ $A_i \in (C \cup V)$	if *top* $= A$ then *pop* *push* $(A_n \ldots A_1 B)$	*push* the prediction string $BA_1 \ldots A_n$
4	$A \to \frac{\varepsilon}{v}$ $v \in \Delta^+$	if *top* $= A$ then write (v) *pop*	*emit* the target string v
5	$A \to \varepsilon$	if *top* $= A$ then *pop*	
6	for every pair $\frac{\varepsilon}{v} \in C$	if *top* $= \frac{\varepsilon}{v}$ then write (v) *pop*	the past prediction $\frac{\varepsilon}{v}$ is now completed by writing v
7	for every pair $\frac{b}{w} \in C$	if $cc = b \wedge$ *top* $= \frac{b}{w}$ then *write* (w) *pop* *advance* the reading head	the past prediction $\frac{b}{w}$ is now completed by writing w
8		if $cc = \dashv \ \wedge$ stack is empty then accept `halt`	the string has been translated

Details about the rules

- Rows $1, 2, 3, 4, 5$ are applicable when the stack top is a nonterminal symbol. In case 2, if the right part begins with a source terminal, the move is conditioned by its presence in the input string.
- Rows $1, 3, 4, 5$ initiate spontaneous moves, which do not shift the reading head.
- Rows $6, 7$ come into play when the stack top is a pair. If the pair contains a source character (row 7), it must match the current input character. If the pair contains a target string (rows $6, 7$), the latter is output.
- Finally, row 8 acknowledges the string if the stack is empty, and the current character indicates the end of the input string.

6.2.6 Syntax analysis with online translation

A pushdown transducer generated using the predictive pushdown transducer algorithm is often nondeterministic and generally not well-suited for a compiler. A more efficient transducer can be directly constructed by transforming a syntactic analyzer into the corresponding syntactic transducer. Given a transduction grammar, assuming that the underlying source grammar allows the construction of a deterministic syntax analyzer, the transduction can be computed by directly translating the syntax tree as it is built. There are two different types of translations depending on the construction of the parser: top-down and bottom-up.

- *Top-down analyzer* (LL): a parser of this type can always be transformed into a transducer.
- *Bottom-up analyzer* (LR): for a bottom-up analyzer to be transformed into a transducer, every grammar rule must satisfy a special restrictive condition (the write action can only occur at the end of the production).

Top-down deterministic transducer Assuming that the source grammar G_1 is $ELL(k)$ with $k = 1$ (ELL(1)), by completing the corresponding parser with write actions, a deterministic transducer can be constructed. This approach represents the simplest method for creating a translator. The transducer can also be designed using a set of recursive procedures.

6.2.7 Regular Translation

A regular translation expression, also known as a regular translation expression in short, is a regular expression equipped with the union, concatenation, star, and cross operators. It operates on pairs of strings (u, v), denoted as $\frac{u}{v}$, where the terms u and v may be empty strings belonging to the source and target alphabets, respectively.

Regular translation expression Let $C \subset \Sigma^* \times \Delta^*$ be the set of pairs (u, v) occurring in the expression. The Regular Translation (also called rational translation) relation defined by the regular translation expression e_τ comprises pairs (x, y) of source and target strings under the following conditions:

- There exists a string $z \in C^*$ in the regular set defined by the regular translation expression e_τ.
- Strings x and y are projections of string z onto the first and second components, respectively.

The sets of source and target strings defined by a regular translation expression are regular languages. However, it's crucial to note that not every translation relation, characterized by two regular sets as its source and target languages, can be expressed with a regular translation expression.

6.2.8 2I Automaton

As the set C of pairs occurring in a regular translation expression can be treated as a new terminal alphabet, recognition of the regular language over C can be achieved through a Finite State Automaton. The concept of a 2I Automaton (two-input automaton) serves as a rigorous method for defining translation and constructing the translation function, enabling the handling of lexical and syntactic translation in a unified manner.

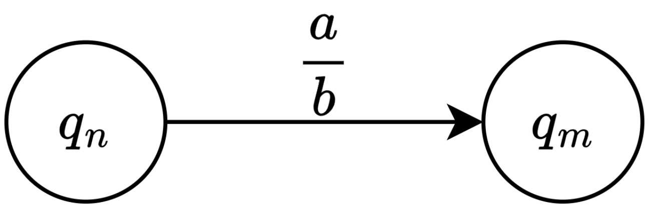
Definition (2I Automaton)**.** A 2I Automaton is a finite automaton with two inputs. It is defined by a set of states Q, an initial state $Q_0 \in Q$, and a set $F \subseteq Q$ of final states. The transition function δ is defined as follows:

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Delta \cup \{\varepsilon\}) \to \wp(Q)$$

An automaton move involves the following actions:

1. The automaton enters state $q^{'}$.
2. If $q^{'} \in \delta(q, a, b)$, the automaton reads a from the source tape and b from the target tape.
3. If $q^{'} \in \delta(q, \varepsilon, b)$, the automaton does not read the source tape and reads character b from the target tape.
4. If $q^{'} \in \delta(q, \varepsilon, \varepsilon)$, the automaton does not read the source tape and does not read the target tape.

The automaton recognizes the source and target strings if the computation concludes in a final state after both tapes have been entirely scanned. A transition from a state q_n to a state q_m while reading a pair (a, b) respectively from the source and from the target tape is denoted as $\xrightarrow{\frac{a}{b}}$



Automaton forms By projecting the arc labels of a 2I automaton onto the first component, a new automaton with one input tape is obtained. The symbols on the tape belong to the source alphabet Σ, and the resulting automaton is subjacent to the original machine. A 2I automaton in normal form is characterized by the fact that each move reads exactly one character either from the source tape or from the target tape, but not from both. Arc labels can be of the following types:

1. Label $\frac{a}{\varepsilon}$ with $a \in \Sigma$ if one character from the source is read.
2. Label $\frac{\varepsilon}{b}$, with $b \in \Delta$, if one character from the target is read.

The families of translation defined by regular translation expressions and by finite (eventually nondeterministic) 2I automata coincide.

Theorem 6.2.1 (Nivat Theorem)**.** *The Nivat Theorem states the following* 4 *conditions are equivalent:*

The translation relation ρ_τ is defined by a right-linear (or left-linear) translation grammar G_τ.

The translation relation ρ_τ is defined by a 2I-automaton.

The translation relation $\rho_\tau \subseteq \Sigma^* \times \Delta^*$ is regular.
There exists and alphabet Ω, a regular language R over Ω and two alphabetic homomorphism:

$$h_1 : \Omega \to \Sigma \cup \{\varepsilon\}$$

$$h_2 : \Omega \to \Delta \cup \{\varepsilon\}$$

such that:

$$\rho_\tau = \{(h_1(z), , h_2(z)) \,|z \in R\}$$

6.2.9 Sequential Transducer

Sequential transducers are automata used to efficiently compute translations in real-time while reading the input tape. Finally, when the input is finished, the automaton may append a finite piece of text that depends on the final state reached.

Definition (*Sequential transducer*)**.** A sequential transducer or IO-automaton T is a deterministic machine defined by a set Q of states, a source alphabet Σ and a target alphabet Δ, an initial state q_0 and a set $F \subseteq Q$ of final states.

Furthermore, there are three single-valued functions:

1. The state transition function δ computes the next state.
2. The output function η computes the string to be emitted by a move.
3. The final function ϕ computes the last suffix to be appended to the target string at termination.

The domains and images of these three functions are as follows:

$$\delta : Q \times \Sigma \to Q \quad \eta : Q \times \Sigma \to \Delta^* \quad \phi : F \times \{\dashv\} \to \Delta^*$$

The graphical representation of the two functions $\delta(q, a) = r$ and $\eta(q, a) = u$ is:

$$q \xrightarrow{\frac{a}{u}} r$$

which means that in the state q, while reading character a, emits string u and moves to the next state r. The ultimate function $\phi(r, \dashv)$ denotes that upon complete reading of the source string, if the final state is r, then the corresponding output string is v. For a source string x, the translation $\tau(x)$ computed by the sequential transducer T results from the combination of two strings generated by the output function and the final function:

$$\left\{ yz \in \Delta^*, |dle|, \exists \text{ a computation labelled } \frac{x}{y} \text{ ending in } r \in F \wedge z = \phi\,(r, \dashv) \right\}$$

The machine is deemed deterministic because the input automaton $\langle Q, \Sigma, \delta, q_0, F\rangle$ associated with T is deterministic, and both the output and final functions, η and ϕ, are single-valued. However, the determinism of the underlying input automaton alone does not guarantee that the translation is uniquely determined, as there could be non-unique outputs between two states of the sequential transducer T, such as when there are two arcs labeled $\frac{a}{b}$ and $\frac{a}{c}$.

A function computable through a sequential state transducer (IO-automaton) is referred to as a sequential function, and the composition of two sequential functions yields another sequential function.

Two opposite passes In cases where a single-valued translation is specified by a regular translation expression or a 2I automaton, it may not always be feasible to implement the translation using a sequential transducer like a deterministic IO-automaton. However, in such scenarios, the translation can be realized through two consecutive (deterministic) sequential passes, each moving in opposite directions while scanning the string:

1. A sequential transducer scans *from left to right*, converting the *source string* into an intermediate string.
2. Another sequential transducer scans the *intermediate string from right to left*, generating the specified target string.

6.3 Semantic Translation

The previously detailed syntactic translation methods prove insufficient for handling slightly more intricate translations, such as converting a number from binary to decimal. These methods rely on tools that are too basic to accomplish such tasks. To address this limitation, a more robust approach, known as syntax-directed translation, becomes necessary. The term ”directed” emphasizes the departure from the purely syntactic methods discussed in preceding sections.

Syntax-directed translation involves semantic techniques, including tree-walking procedures that traverse the syntax tree and compute variables known as semantic attributes. These attributes encapsulate the meaning or semantics of a given source text. It’s crucial to note that a syntax-directed method is not a formal model because the procedures for computing attributes lack formalization.

A syntax-directed compiler carries out two consecutive phases:

1. Parsing or syntax: this phase computes a syntax tree, typically condensed into an abstract syntax tree, containing essential information for the subsequent phase.
2. Semantic evaluation or semantic: the semantic phase involves applying a set of semantic functions to each node of the tree until all attributes are evaluated. The set of evaluated attribute values represents the meaning or translation, and this information is found at the root of the tree.

This two-phase approach is termed two-pass compilation and is the most common and straightforward method of compilation. The separation of parsing and semantic evaluation provides compiler designers with greater flexibility in creating these phases.

6.4 Attribute grammars

The significance of a sentence lies in a collection of attribute values, determined by semantic functions and assigned to the nodes of the syntax tree. Within a syntax-directed translator, the definition of these semantic functions is encapsulated, and they are linked to the grammar rules. The amalgamation of grammar rules and their associated semantic functions is termed an attribute grammar.

For the sake of simplicity, the attribute grammar is formulated in reference to an abstract syntax, which may be less intricate than the actual grammar but is often prone to ambiguity. Nonetheless, this ambiguity does not hinder a single-valued translation, as the parser transmits only one syntax tree to the semantic evaluator.

In instances where compilers are less complex, they might streamline the process by consolidating the two phases into a single pass utilizing a unified syntax, typically that of the language itself.

Attribute grammar An attribute grammar is defined as follows:

1. A context-free syntax $G = (V, \Sigma, P, S)$ where V and Σ represent the terminal and nonterminal sets, P comprises the production rule set, and S serves as the axiom. It is advisable (though not mandatory) to exclude the axiom from any rule RP.
2. A collection of symbols, known as (semantic) attributes, is linked with both nonterminal and terminal syntax symbols. The set of attributes associated with a symbol $\bigoplus$ is represented as $\text{attr}\,(\bigoplus)$. Within the grammar, the attribute set is divided into two distinct sets: the left and right attributes.
3. A set of semantic functions (or rules) is defined with the following characteristics:

   - Each function is linked to a production rule:

     $$p : D_0 \to D_1 D_2 \ldots D_r \quad r \geq 0$$

     where D_0 is a nonterminal, and the other symbols can be either terminal or nonterminal.
   - The production p is referred to as the syntactic support of the function and may be shared among different functions.
   - The attribute σ associated with a symbol D_k is denoted by σ_k or σ_D if the syntactic symbol occurs exactly once in production p.
   - A semantic function is structured as follows:

     $$\sigma_k := f\left(\text{attr}\left(\{D_0, D_1, \ldots, D_k\}\right) \setminus \{\sigma_k\}\right) \quad 0 \leq k \leq r$$

     where function f assigns the computed value to the attribute σ of symbol D_k, and its arguments can include any attributes of the same production p, excluding σ_k itself.
   - Generally, semantic functions cover their entire domain and are expressed in a suitable notation, termed semantic metalanguages, which can be informal.
   - A function $\sigma_0 := f(\ldots)$ defines an attribute, identified as left, of the nonterminal D_0, which serves as the LP (or parent) of the production.
   - A function $\sigma_k := f(\ldots),\ k \geq 1$ defines an attribute, labeled as right, of a symbol (sibling or child) D_k present in the RP.
   - The same attribute cannot serve as left in one function and right in another.
   - Since terminal characters never appear in the left part, their attributes cannot be of the left type.

4. The set fun(p) of functions supported by production p must adhere to the following conditions:

   (a) For each left attribute σ_0 of D_0, there must exist exactly one function in fun(p) defining the attribute.

   (b) No function within fun(p) defining the attribute exists for each right attribute δ_0 of D_0.

   (c) No function within fun(p) defining the attribute exists for each left attribute $\sigma_i, i \geq 1$.

   (d) For each attribute $\delta_i, i \geq 1$, there must exist exactly one function in fun(p) defining the attribute.

   The left attributes σ_0 and the right ones δ_i with $i \geq 1$ are termed internal for production p because functions supported by p define them. Conversely, the right attributes δ_0 and left attributes σ_i with $i \geq 1$ are termed external for production p since functions supported by other productions define them.

5. Certain attributes can be initialized with constant values or values computed by external functions. This is frequently observed in the case of lexical attributes, which are linked to terminal symbols. In such instances, the grammar does not stipulate a specific computation rule.

6.4.1 Dependence graph

If a grammar is specified through a translation, it abstracts from the details of tree-traversing procedures. The attribute evaluation program can be automatically constructed based on the functional dependencies between attributes, assuming the semantic function bodies are provided.

Definition (*Dependence graph of a semantic function*)**.** The nodes of dependence graph of a semantic function are the arguments and results of the function considered, and there is an arc from each argument to the result.

Definition (*Dependence graph of a production p*)**.** The dependence graph of a production p, denoted by dep_p, collects the dependence graphs for all the functions supported by the production considered.

Definition (*Dependence graph of a decorated syntax tree*)**.** The dependence graph of a decorated syntax tree is obtained by pasting together the graphs of the individual productions that are used in the tree nodes.

A grammar is termed acyclic (or loop-free) if the dependence graph of the tree is acyclic for every sentence.

Property 6.4.1 (*Correct attribute grammar*)**.** Given an attribute grammar satisfying the conditions for attribute grammars, the following holds: if the attribute dependence graph of the tree is acyclic, the system of equations corresponding to the semantic function has exactly one solution.

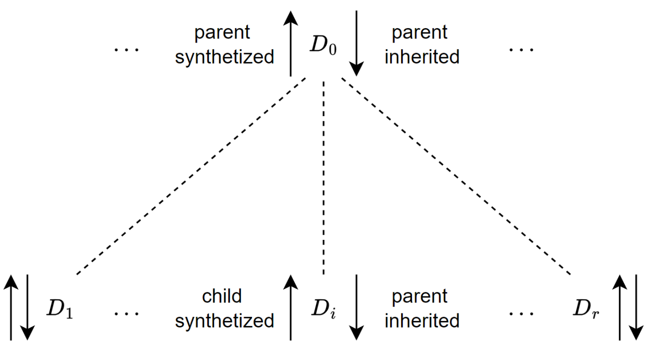


Figure 6.1: One-sweep semantic evaluation

Under the acyclicity condition, the equations can be ordered so that each semantic function is applied after the functions that compute its arguments. This results in a value for the solution, given the totality of the functions. The solution is unique, similar to a system of linear equations. The topological ordering method can be used to provide a total order of nodes, although this is an inefficient way to compute the solution. It would require applying the sorting algorithm even before computing the attribute values.

Checking whether a given grammar is acyclic poses another problem. Since the source language is typically infinite, exhaustive enumeration of all trees for the acyclicity test is impractical. An algorithm determining if an attribute grammar is acyclic exists but is $\mathcal{NP}$-complete and thus not used in practice. It is more convenient to test certain sufficient conditions.

6.4.2 One-sweep semantic evaluation

An efficient evaluator should be capable of computing all attributes of a tree in just one pass. This involves traversing the tree through a depth-first search, allowing for attribute evaluation in a single sweep across the entire tree.

Let N represent a node in the tree, with $N_1, \ldots, N_r$ as its children, and t_i denoting the subtree rooted at node N_i. A depth-first algorithm starts by visiting the tree root, and to visit the subtree t_N rooted at node N, it recursively follows these steps:

1. Conduct a depth-first visit of the subtrees $t_1, \ldots, t_r$ in an order corresponding to a permutation of $1, \ldots, r$.

2. Evaluate the attributes with the following principles:

   - Before entering and evaluating a subtree t_N, compute the right attributes of node N (the root of the subtree). These attributes are then passed as input parameters to the procedure implementing the visit. Procedure calls with input parameter passing constitute the ”descending phase” of the visit.

   - At the end of the visit of subtree t_N, compute the left attributes of node N. These attributes are the output parameters of the procedure implementing the visit. Procedure returns with output parameter passing constitute the ”ascending phase” of the visit.

Not all grammars are compatible with this one-sweep procedure because more intricate functional dependencies may necessitate multiple visits to the same node.

One-Sweep Grammar For each production $p : D_0 \to D_1 D_2 \dots D_r$ with $r \geq 0$, it is necessary to introduce a new relation between the symbols in the right part of the production. This allows the creation of a directed graph, referred to as the sibling graph sibl_p, which captures the relations between the symbols in the right part of the production p. The nodes of sibl_p are the symbols $\{D_1, \dots, D_r\}$ of the production, and has arcs $D_i \to D_j$, with $i \neq j,\ i, j \geq 1$ if in the dependence graph dep_p there is an arc $\sigma_i \to \delta_j$ from an attribute of symbol D_i to an attribute of symbol D_j. Note that the nodes in the sibling graph are syntactical symbols, distinct from the attributes in the dependence graph.

A grammar satisfies the one-sweep condition if, for each production $p : D_0 \to D_1 D_2 \dots D_r, r \geq 0$ that has a dependence graph dep_p, the following clauses hold at the same time:

1. Graph dep_p contains no circuit (is acyclic).
2. Graph dep_p does not contain a path $\lambda_i \to \dots \to \rho_i,\ i \geq 1$ that goes from a left attribute λ_i to a right attribute ρ_i of the same symbol D_i, where D_i is a sibling of D_0.
3. Graph dep_p contains no arc $\lambda_0 \to \rho_i,\ \geq 1$, from a left attribute of the father node D_0 to a right attribute of a sibling node D_i.
4. The sibling graph sibl_p contains no circuit (is acyclic).

Explanation of the conditions

1. Necessary for the grammar to be acyclic.
2. If such a path existed, it would be impossible to compute the right attribute ρ_i before visiting the subtree t_i, because the value of the left attribute λ_i is computed at the end of the visit of t_i (against the order of the depth-first visit).
3. The value of attribute ρ_i would not be available when the visit of the subtree t_i is started.
4. Allows to topologically sort the child nodes.

Construction of the one-sweep evaluator The procedure visits the subtrees, computes, and returns the left attributes of the root of the subtree. For each production $p : D_0 \to D_1 D_2 \dots D_r,\ r \geq 0$:

1. Choose a topological order (TOS) of the nonterminals $D_1, D_2, \dots D_r$ with respect to the sibling graph sibl_p.
2. For each symbol $D_i, 1 \leq i \leq r$, choose a topological order (TOR) of the right attributes of symbol D_i with respect to the dependence graph dep_p.
3. Choose a topological order (TOL) of the left attributes of symbol D_0 with respect to the dependence graph dep_p.

The three orders TOS, TOR and TOL prescribe how to arrange the instructions of the procedure that implements the visit of the subtree t_N.

6.4.3 Combined syntax and semantic analysis

The integration of syntax tree construction and attribute computation allows the parser to handle both responsibilities, invoking semantic functions as needed. In this context, we assume the use of a pure BNF grammar, avoiding the complexity that EBNF productions would introduce in specifying the relationship between syntax symbols and attributes.

Source language		**Tool**
regular	lexical analysis with lexical attributes	flex, lex
LL(k)	recursive descent parser with attributes	
LR(k)	shift-reduce parser with attributes	yacc, bison

Table 6.1: Combined syntax and semantic

Lexical analysis with attribute evaluation The lexical analyzer, or scanner, not only divides the source text into lexemes (or tokens) but also assigns semantic attributes to these lexemes. Lexemes, representing the smallest meaningful substrings, are categorized into lexical classes defined by regular formal languages, such as regular expressions or sets of strings. The lexical and syntactic specifications operate at different levels, with the lexical level determining the form of lexemes and the syntactic level assuming these lexemes as terminal symbols in the grammar. Some lexemes may carry additional meaning, serving as semantic attributes computed by the lexical analyzer.

Attributed recursive descent translator In cases where a syntax is suitable for deterministic top-down parsing, attribute evaluation can proceed during parsing, provided that the functional dependencies of the grammar satisfy conditions beyond those of a one-sweep grammar. The one-sweep algorithm traverses the syntax tree in a depth-first order, which may differ from the natural order constructed by a top-down parser. To merge these two procedures, functional dependencies must be free from any conflicts that might result from differing tree traversal orders.

L-condition A grammar satisfies the condition L if, for each production $p : D_0 \to D_1 \dots D_r$, it holds:

1. The one-sweep condition is satisfied.
2. The sibling graph sibl_p contains arc $D_j \to D_i, j > i \geq 1$.

The second condition in the L-condition prevents a right attribute of node D_i from depending on any attribute of a node D_j placed to its right in the production p.

Attribute grammar and deterministic parsing Consider an attribute grammar G, where:

- The syntax satisfies the $LL(k)$ condition.
- The semantic rules satisfy the L-condition.

It's possible to construct a top-down deterministic parser with attribute evaluation able to compute the attributes of G at parsing time.

6.5 Static analysis

Static analysis is a methodology employed by compilers to examine certain properties of the source code prior to the compilation phase. Various analyses can be classified based on their intended objectives, including:

- *Verification*: this involves examining the correctness of the program.
- *Optimization*: the goal is to enhance the efficiency of the program.
- *Scheduling and parallelizing*: this aims to alter the program's execution order to exploit parallelism.

In these scenarios, a control-flow graph is employed, which is a directed graph resembling a program flowchart. This graph can be conveniently interpreted as representing the state-transition function of a finite automaton. Static analysis involves scrutinizing this graph using various techniques. It is crucial to note that the control-flow graph only reflects the source code of an individual program, not the entire source language. Static flow analysis should not be conflated with syntax-driven translation.

6.5.1 Program as an automaton

The control-flow graph serves as an abstraction of a program and can be likened to a Finite State Automaton (FSA). Its key components are detailed below:

- Each node represent an instruction. Instructions are simplified compared to those in the source language. Typical instructions include assignments, jumps, arithmetic operations, and more. Operands consist solely of simple variables and constants.
- Each arc signifies a possible control flow. If an instruction p is followed by q, a directed arc is present from p to q. Unconditional instructions have at most one successor. Conditional instructions have two (or more) successors. An instruction with two or more predecessors creates a confluence of arcs in the graph.
- The first instruction of the program serves as the entry point, corresponding to the initial node.
- The last instruction of the program serves as the exit point, corresponding to the final node.

While a control-flow graph is a valuable representation, it lacks certain details of the program:

- The `true`/`false` value determining the successor of a conditional instruction.
- The node representing a `goto` instruction is absent, as it is simplified to an arc to the successor instruction.
- Any operations executed by an instruction are abstracted. A value assignment is considered to define a variable. If a variable appears in the Right-Hand Part (RHP) of an expression or boolean condition, it is deemed used. A node representing a statement p in the graph is associated with set def(p) of defined variables and set use(p) of used variables.

Definition (*Language of the control-flow graph*)**.** Consider a finite state automaton A, depicted as a control-flow graph. Its terminal alphabet is the set I of program instructions, with each instruction represented by a 3-tuple:

$$\langle \texttt{label}, \texttt{defined variables}, \texttt{used variables} \rangle$$

The language $L(A)$ recognized by the automaton comprises strings over the alphabet I. These strings label the paths from the initial node (the entry point) to the final node (the exit point). Each string in $L(A)$ signifies a sequence of program instructions that the machine can execute when the program runs.

Furthermore, language L is local.

Conservative approximation The automaton provides only an approximation of the valid execution paths of a program. It does not guarantee that all paths specified are executable, as it does not perform syntax analysis on the conditions governing the selection of successor nodes in conditional instructions.

Example:

```
1: if x^2 >= 0 then
2: instruction_2 else
3: instruction_3
```

The formal language accepted by the automaton contains two paths:

$$\{1\,2, 1\,3\}$$

However, the second path is not executable, as the condition is always true.

As a result, static analysis can sometimes lead to pessimistic conclusions (discovering never-executed code paths). Determining whether a path in a control-flow graph will ever be executed by the program is undecidable, reducing the problem to the halting problem.

Analyzing all recognized paths is a conservative approximation to a program. While it may identify non-existing errors or erroneously assign resources, it ensures that real errors will never be missed. Despite its potential inefficiency, this method is deemed **error safe**.

A common assumption in static analysis is that the automaton is clean, meaning every instruction is on a path from the entry point to the exit point. Deviations from this assumption can lead to anomalies:

- Some executions may not reach the exit point.
- Some instructions may remain unexecuted (dead code).

6.5.2 Liveness of a variable

A professional compiler undergoes multiple analysis passes to optimize code, with one of the most crucial being the analysis of variable liveness, determining the duration for which the value of a variable is required.

Variable liveness A variable a is considered live upon exiting a program node p if there exists a path from p to another node q (not necessarily distinct from p) in the program's control-flow graph. This path must satisfy the following conditions:

- The path does not traverse an instruction $r \neq q$ that defines a. If r defines a, then $a \in \text{def}(r)$.
- The instruction q uses a. If q uses a, then $a \in \text{use}(q)$.

In simpler terms, a variable is considered live out of a particular node if some instruction that could be subsequently executed utilizes the value that the variable held in the former node. If the variable is reassigned before its next use, it is not live out of the node. More precisely, a variable is live-out for a node if it is live on any outgoing arc from that node. Similarly, a variable is live-in for a node if it is live on some incoming arc to the node.

Computing liveness intervals Let I denote the instruction set, and let $D(a)(I) \subseteq I$ and $U(a)(I) \subseteq I$ represent the sets of instructions defining and using variable a, respectively. Variable a is considered live out of an instruction p if and only if, for the language $L(A)$ accepted by the automaton, condition is satisfied: the language $L(A)$ contains a sentence $x = upvqw$, where:

- u, w are arbitrary sequences of instructions (possibly empty).
- p is any instruction.
- v is a possibly empty instruction sequence not containing a definition of a.
- q is an instruction that uses a.

These conditions are formalized by the equation:

$$u, w \in I^* \wedge p \in I \wedge v \in (I \setminus D(a)) \wedge q \in U(a)$$

Here, the set difference contains all instructions that do not define a, while q uses a. The set of all strings x satisfying this condition, denoted as L_p is a subset of the language $L(A)$ recognized by the automaton. L_p is regular because it can be defined by the intersection:

$$L_p = L(A) \cap R_p$$

where R_p is the regular language defined by the regular expression:

$$R_p = I^* p \, (I \setminus D(a))^* \, U(a) \, I^*$$

The definition of R_p and L_p specifies that the symbol p must be followed by a symbol q from the set $U(a)$, and all symbols between p and q must not be in set $D(a)$. To check if a variable is live out of node p, it is sufficient to verify if $L_p \neq \emptyset$. This can be achieved by building the recognizer as the product of machine A and the recognizer of R_p. If no path connects the input node to the final node, then L_p is empty.

However, this procedure is not efficient given the large number of variables and instructions in real-world programs. A more efficient technique is data-flow analysis, which examines all paths from any instruction to another instruction using the same variable.

Data-Flow equations The computation of liveness is articulated through a system of data-flow equations. Let's consider a node p in a program A. The equations establish the relationship between variables that are live out (denoted $\text{live}_{\text{out}}(p)$) and those live in (denoted by $\text{live}_{\text{in}}(p)$) for the node. Additionally, they express the connection between variables live out of a node and those live in for its successors. Here, $\text{succ}(p)$ denotes the set of (eventually immediate) successors of node p and $\text{var}(A)$ is the set of all variables in program A.

Definition (*Data-flow equations*)**.** For each final node p:

$$\text{live}_{\text{out}}(p) = \emptyset$$

For any other node p:

$$\text{live}_{\text{in}}(p) = \text{use}(p) \cup (\text{live}_{\text{out}}(p) \setminus \text{def}(p))$$

$$\text{live}_{\text{out}}(p) = \bigcup_{q \in \text{succ}(p)} \text{live}_{\text{in}}(q)$$

Solution of data-flow equations In the context of a control-flow graph with a total of $|I| = n \geq 1$ nodes, the resulting system gives rise to $2 \cdot n$ equations involving $2 \cdot n$ unknown variables: $\text{live}_{\text{in}}(p)$ and $\text{live}_{\text{out}}(p)$ for each node $p \in I$. Each unknown is a set of variables, and the solution sought is a pair of vectors, each containing n sets.

To solve the system of equations, an iterative approach is employed, starting with the empty set as the initial approximation ($i = 0$) for every unknown:

$$\forall\, p \in I \quad \text{live}_{\text{in}}(p) = \emptyset \quad \text{live}_{\text{out}}(p) = \emptyset$$

At each iteration i, for each equation in the system defined, the unknowns on the right-hand side are replaced with the values computed in the previous iteration. The new iteration $i + 1$ is then calculated. The iteration process halts when the values computed in the last iteration are equal to those computed in the previous iteration (a fixed point is reached). This solution is termed the least fixed point solution of the transformation that computes a new vector from one of the preceding iterations.

A finite number of iterations is always sufficient to compute the least fixed point solution because:

- Every set $\text{live}_{\text{in}}(p)$ and $\text{live}_{\text{out}}(p)$ is finite, given the finite number of variables.
- Each iteration either increases the cardinality of the aforementioned sets of variables or leaves them unchanged, as the equation is monotonic.
- When the solution ceases to change, the algorithm terminates.

6.5.3 Application of Liveness Analysis

Memory allocation Liveness Analysis is particularly valuable for determining whether two variables can share the same memory cell. If two variables are live-in at the same program instruction, then they must be stored in different memory cells. This is indicative of the two variables interfering with each other. Conversely, if two variables do not interfere, it implies that they can be stored in the same memory cell.

Useless instructions An instruction that defines a variable is deemed useless if the assigned value to the variable is never utilized by any subsequent instruction; in other words, the value is not live-out for the defining instruction. To ascertain the uselessness of the definition of a variable a by instruction p, it suffices to check if $\text{live}_{\text{out}}(p) \nsubseteq \{a\}$.

6.5.4 Reaching definition analysis

Another fundamental type of static analysis involves the search for a variable definition that reaches a specified instruction. In particular, this is crucial when an instruction assigns a constant value to a variable. The compiler examines the program to determine if this constant can replace the variable in subsequent instructions using it. This transformation offers two advantages:

1. *Reduced memory access and increased speed*: by replacing the variable with a constant value, the number of memory accesses is reduced, potentially leading to a faster program.
2. *Compile-time evaluation*: obtaining an expression where all operands are constants allows for compile-time evaluation.

The second transformation is termed constant propagation.

Definition (*Reaching definition*)**.** The definition of a variable a at instruction q *(denoted as* a_q*)* reaches the input of an instruction p (not necessarily different from q) if there exists a path from q to p that does not contain any redefinition of a.

This implies that instruction p can access and use the value of the variable a defined in instruction q.

Reaching definition and regular expressions The definition a_q reaches instruction p if language $L(A)$ contains a sentence of the form $x = uqvpw$, where:

- u, w are arbitrary sequences of instructions (possibly empty).
- p is any instruction.
- v is a sequence of instructions that do not contain any definition of a (possibly empty).
- q is an instruction that defines a.

The condition is represented by the following regular expression:

$$u, w \in I^* \land q \in D(a) \land v \in (I \setminus D(a))^* \land p \in I$$

where p, q may coincide.

Reaching definition and data-flow equations If node p defines variable a, any other definition a_q of the same variable in another node q, with $q \neq p$ is suppressed by p. The set of definitions suppressed by instruction p is given by:

$$\begin{cases} \text{sup}(p) = \emptyset & \text{if def}(p) = \emptyset \\ \text{sup}(p) = \{a_q \mid q \in I \land q \neq p \land a \in \text{def}(q) \land a \in \text{def}(p)\} & \text{if def}(p) \neq \emptyset \end{cases}$$

Data-flow equations For the initial node 1:

$$\text{in}(1) = \emptyset$$

For any other node $p \in I$:

$$\text{out}(p) = \text{def}(p) \cup (\text{in}(p) \setminus \text{sup}(p))$$

$$\text{in}(p) = \bigcup_{\forall q \in \text{pred}(p)} \text{out}(q)$$

Similar to the liveness equations, the reaching definition system can be solved through iteration until the computed solution converges to a fixed point; initially, all sets are empty. The set out' represents the elements of out reaching the exit node starting from node q, but with their subscripts deleted.

Explanation

- The initial data flow equations presuppose the absence of variables passed as input parameters to the subprogram. If there are input parameters, in(1) would then comprise the set of these
- The subsequent data flow equations incorporate into the exit from node p all local definitions of p and the definitions reaching the entrance to p, unless the latter are overridden by p.
- The final data flow equations assert that any definition reaching the exit of a predecessor node also reaches the entrance to node p.

Constant propagation The constant propagation problem is the search for constant expressions that can be evaluated at compile time. In the instruction p, it is safe to replace with a constant k any variable a used in p if the following conditions hold:

1. There exists and instruction $q : a := k$, such that a_q reaches p
2. No other definition a_r of variable a reaches the entrance of p, with $r \neq q$

Availability of variables and initialization A basic correctness check performed by a compiler is to verify that all the variables are initialized before their first use; more generally, a variable used in some instruction must be available at the entrance of that instruction.

Definition (*Variable availability*)**.** A variable a is available at the entrance of instruction p (just before its execution) if in the program control-flow graph every path from the initial node 1 to the entrance of p contains a statement that defines variable a.

Definition (Badly Initialized Variables)**.** An instruction p is not well initialized if the following predicate holds:

$$\exists q \in \text{pred}(p) \text{ such that } use(p) \not\subseteq \text{out}'(q)$$

The condition says that there exists a node q predecessor of p such that the definition of reaching its exit does not include all the variables used in p. Therefore, when the program execution runs on a path through q, one or more variables used in p don't have a value.

CHAPTER 7

Laboratory

7.1 Regular expression

The POSIX compatible regular expression library is a standard library of the C programming language. It contains an extended set of functions to build and manipulate regular expressions.

Syntax	**Matches**
x	the character x
.	any character except newline
[x,y,z]	any character in the set x, y, z
[^x,y,z]	any character not in the set x, y, z
[a-z]	any character in the range a-z
[^a-z]	any character not in the range a-z

Table 7.1: Basic character sets

Syntax	**Matches**
R	the regular expression R
R S	the concatenation of R and S
R—S	the alternation of R and S
R*	zero or more occurrences of R
R+	one or more occurrences of R
R?	zero or one occurrence of R
R{n}	exactly n occurrences of R
R{n,}	at least n occurrences of R
R{n,m}	at least n and at most m occurrences of R

Table 7.2: Composition of Regular Expressions

Syntax	Matches
`(R)`	capture group or override precedence
`^R`	match at the beginning of the line
`R$`	match at the end of the line
`\t`	tab character
`\n`	newline character
`\w`	a word *(same as `[a-zA-Z0-9_]`)*
`\d`	a digit *(same as `[0-9]`)*
`\s`	a whitespace character (*same as* [`\t\s\n`])
`\W`	a non-word character
`\D`	a non-digit character
`\S`	a non-whitespace character

Table 7.3: Regular expression utilities

7.2 Lexical analysis

The purpose of the lexical analysis is:

1. To recognize the tokens of the language.
2. To (possibly) decorate the tokes with additional information.

Such analysis is performed through a scanner, which basically is just a big FSA. Since coding a scanner is a hard task, scanner generators based on regular expressions such as `flex` are used. In a compiler, the scanner prepares the input for the parser:

1. It *detects* the tokens of the language.
2. It *cleans* the input.
3. It *adds* information to the tokens.

Words Words cannot be enumerated in artificial languages, as there are too many of them (despite being bounded). However, technical words are simpler than natural words:

- Their structure is simple.
- They follow specific rules.
- They are (normally) a regular language.

Property 7.2.1 (`C` identifiers)**.** The first character must be a letter or an underscore. The following characters must be letters, digits or underscores.