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

Regular Languages

1.1 Formal language theory

A formal language is composed 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

> An *alphabet* Σ is a finite collection of elements referred to as *characters*, denoted as $\{a_1, a_2, \ldots, a_k\}$.
>
> The *cardinality* of an alphabet $\Sigma = \{a_1, a_2, \ldots, a_k\}$ represents the number of characters it encompasses, denoted as $|\Sigma| = k$.
>
> A *string* is a sequential arrangement of elements from the alphabet, potentially with repetitions.

Example: The alphabet $\Sigma = \{a, b\}$ consists of two distinct characters. From this alphabet, various languages can be generated, including:

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

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

Definition

> The strings of a language are called *sentences* or *phrases.*
>
> The *cardinality* of a language is the number of sentence it contains.

Example: Considering the language $L_2 = \{bc, bbc\}$, it is evident that its cardinality is two.

Definition

> The count of times a specific letter appears in a word is referred to as the *number of occurrences*.
>
> The *length* of a string corresponds to the total number of elements it contains.
>
> Two strings are considered *equal* if and only if the following conditions are met:
>
> - They possess the same length.
>
> - Their elements match from left to right, sequentially.

Example : In the string aab, the number of occurrences of the letters a and c is denoted as follows:

$$|aab|_a = 2$$

$$|aab|_c = 0$$

The length of the string aab is determined as:

$$|aab| = 3$$

1.2   Operations on strings

Concatenation

When you have two strings, $x = a_1 a_2 \ldots a_h$ and $y = b_1 b_2 \ldots b_k$, concatenation is defined as:

$$x \cdot y = a_1 a_2 \ldots a_h b_1 b_2 \ldots b_k$$

Concatenation exhibits non-commutative and associative properties $(x(yz) = (xy)z)$. The length of the resulting concatenated string is equal to the sum of the lengths of the individual strings:

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

Empty string

The empty string, denoted as ε serves 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, namely $x, y,$ 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 that is not identical to the entire string x is referred to as a proper non-empty substring.

Reflection

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

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

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

Operations on a language are usually defined by applying the string operations to all of its phrases.

Reflection

The reflection L^R of a language L consists of a finite set of strings that are reversals of sentences 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 follows:

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

A language is considered prefix-free if it contains none of the proper prefixes of its sentences:

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

Example : The language $L_1 = \{x | x = a^n b^n \wedge n \geq 1\}$ is prefix-free.

The language $L_2 = \{x | x = a^m b^n \wedge m > n \geq 1\}$ is not prefix-free.

Concatenation

When dealing with languages L' and L'', the concatenation operation is defined as:

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

Repetition

The definition of repetition for languages is as follows:

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

The corresponding identities are:

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

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

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

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

Example: The language $L = \{\varepsilon, a, b\}^k$ with $k = 3$ can be represented as follows:

$$L = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, \ldots, bbb\}$$

Set operations

As a language is a set, it supports the standard set operations, including 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 the strings, over an alphabet Σ, of any length including zero:

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

Complement

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

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

In other words, it comprises the strings over the alphabet Σ that do not belong to the language L. It's important to note that:

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

The complement of a finite language is always infinite. However, the complement of an infinite language is not necessarily finite.

Reflexive and transitive closures

Given a set A and a relation $R \subseteq A \times A$, the pair $(a_1, a_2) \in R$ is often represented as $a_1 R a_2$. The relation R^* is a relation defined by the following properties:

- Reflexive property:

$$xR^*x \quad \forall x \in A$$

- Transitive property:

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

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 a relation defined by the following property:

- Transitive property:

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

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

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

Star operator

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

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

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

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

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

Every string within the star language L^* can be divided into substrings belonging to the base language L. Consequently, the star language L^* can be equivalent to the base language L. If we take the alphabet Σ as the base language, then Σ^* contains 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 we observe that $\varnothing^* = \{\varepsilon\}$ and $\{\varepsilon\}^* = \{\varepsilon\}$.

Cross operator

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

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

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

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

Quotient

The quotient operator reduces the phrases in L_1 by removing a suffix that belongs to L_2 and is defined as follows:

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

Example: 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 family of regular languages is the most basic among formal language families and can be defined in three different ways: algebraically, through generative grammars, and by using recognizer automata.

Definition

> A *regular expression* is a string denoted as r, constructed over the alphabet $\Sigma = \{a_1, a_2, \ldots, a_k\}$ and featuring metasymbols: union ($\cup$), concatenation ($\cdot$), star (*), empty string (ε), subject 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: $r = s^*$.
>
> Here, the symbols s and t represent regular expressions.

The operator precedence is as follows: star has the highest precedence, followed by concatenation, and then union.

In addition to these operators, we often make use of derived operators:

- 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 $s\vert t$	$L_s \cup L_t$
$s \cdot t$ or st	$L_s \cdot L_t$
s^*	L_s^*

Definition

A *regular language* is a language that is represented by a regular expression.

Example: The regular expression $e = (111)^*$ represents the language $L_e = \{\varepsilon, 111, 111111, \dots\}$.

The regular expression $e_1 = 11(1)^*$ represents the language $L_e = \{11, 111, 1111, 11111, \dots\}$.

Definition

The *family of regular languages*, denoted as REG, is the collection of all regular languages.

The *family of finite languages*, denoted as FIN, is the collection of all languages with finite cardinality.

Every finite language is considered regular because it can be expressed as the union of a finite number of strings, each of which is 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's important to note that the family of regular languages includes languages with infinite cardinality as well. Therefore, we can conclude that $\text{FIN} \subset \text{REG}$.

The union and repetition operators in regular expressions correspond to possible choices, allowing for the creation of sub-expressions that identify specific sub-languages.

Expression r	Choice of r
$e_1 \cup \cdots \cup e_n$ or $e_1\vert \dots \vert e_n$	e_k for every $1 \le k \le n$
e^*	ε or e^n for every $n \ge 1$
e^+	e^n for every $n \ge 1$

When working with a regular expression, it's possible to derive a new one by replacing any outermost sub-expression with another that represents a choice of it.

Definition

We state that a regular expression e' *derives* a regular expression e'', denoted as $e' \implies e''$, when the two regular expressions can be factorized as:

$$e' = \alpha\beta\gamma$$

$$e'' = \alpha\delta\gamma$$

Here, δ represents a choice involving β.

The derivation relation can be applied iteratively, resulting in 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: The expression $e_0 \overset{n}{\implies} e_n$ implies that e_n is derived from e_0 in n steps.

The expression $e_0 \overset{+}{\implies} e_n$ implies that e_n is derived from e_0 in $n \geq 1$ steps.

The expression $e_0 \overset{*}{\implies} e_n$ implies 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 latter define the language specified by the regular expression.

It's essential to note that in derivations, operators must be selected from the external to the internal layers. Making a premature choice could eliminate valid sentences from consideration.

Definition

Two regular expressions are considered *equivalent* if they define the same language.

Ambiguity

A phrase from a regular language can be derived through different equivalent derivations. These derivations may vary in the order of the choices made during the derivation process. To determine the expression that can be derived in multiple ways, we need to establish the numbered subexpressions of a regular expression. To achieve this, follow these steps:

- Begin with a regular expression and consider all possible parentheses.

- Derive a numbered version, denoted as e_N, of the original regular expression, e.

- Identify all the numbered subexpressions within the expression.

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

$$e_N = (a_1 \cup (b_2 b_3))^*(c_4^+ \cup (a_5 \cup (b_6 b_7)))$$

From this expression, we can derive its subexpressions by iteratively removing the parentheses and union symbols.

Definition

> A regular expression is considered *ambiguous* of its numbered version, denoted as f', contains two distinct strings, x and y, that become identical when the numbers are removed.

Example : Taking the regular expression $e = (aa|ba)^*a|b(aa|b)^*$, its corresponding numbered version is $e_N = (a_1a_2|b_3a_4)^*a_5|b_6(a_7a_8|b_9)^*$.

From this expression, we can derive $b_3a_4a_5$ and $b_6a_7a_8$, both of which map to the string baa. Consequently, it can be concluded that the regular expression e is ambiguous.

Ambiguity is often a source of problems.

Extended regular expressions

To define a regular expression, we can introduce the following operators without altering its expressive power:

- Power: $a^h = aa \ldots a$ for h times.

- Repetition: $[a]_k^n = a^k \cup a^{k+1} \cup \cdots \cup a^n$.

- Optionality: $(\varepsilon \cup a)$ or $[a]$.

- Ordered interval: $(0 \ldots 9)(a \ldots z)(A \ldots Z)$.

- Intersection: useful to define languages through conjunction of conditions.

- Complement: $\neg L$.

Closure properties of the REG family

Definition

> Suppose *op* represents a unary or binary operator. A family of languages is said to be closed under *op* if and only if every language obtained by applying the *op* operator to languages within that family remains within the same family.

Property 1.1. The REG family is closed under concatenation, union, star, intersection, and complement operators.

This implies that regular languages can be combined using these operators without going beyond the boundaries of the REG family.

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 define more useful languages, whether regular or not, we transition to the formal framework of generative grammars. Grammars provide a more robust method for defining languages using rewriting rules.

Definition

> A *context-free grammar* G is defined by four entities:
>
> 1. V nonterminal alphabet, is the set of nonterminal symbols.
>
> 2. Σ terminal alphabet, is the set of the symbols of which phrases or sentences are made.
>
> 3. P is the set of rules or productions.
>
> 4. $S \in V$ is the specific nonterminal, called the axiom (S), from which derivations start.

A grammar rule is expressed as:
$$X \to \alpha$$
Here, $X \in V$ and $\alpha \in (V \cup \Sigma)^*$ If multiple rules share the same nonterminal X, we can succinctly represent the rule as:
$$X \to \alpha_1|\alpha_2|\ldots|\alpha_n$$
In this case, we say that the strings $\alpha_1, \alpha_2, \ldots, \alpha_n$ are the alternatives for the nonterminal X.

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

- Lowercase Latin letters $\{a, b, \ldots\}$ for terminal characters.

- Uppercase Latin letters $\{A, B, \ldots\}$ for nonterminal symbols.

- Lowercase Latin letters $\{r, s, \ldots, z\}$ for strings over the alphabet Σ.

- Lowercase Greek letters $\{r, s, \ldots, z\}$ for both terminals and non.

- σ only for nonterminals.

The rules are categorized into the following types:

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

2.2 Derivation and language generation

Definition

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 we have the following conditions:

- $\beta = \delta A \eta$.

- There exists a rule $A \to a$ in the grammar G.

- $\gamma = \delta \alpha \eta$

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

If $A \overset{*}{\Longrightarrow} \alpha$, then $\alpha \in (V \cup \Sigma)$ is called *string form generated by G*.
If $S \overset{*}{\Longrightarrow} \alpha$, then α is called *sentential* or phrase form.
If $A \overset{*}{\Longrightarrow} s$, then $s \in \Sigma^*$ is called *phrase* or sentence.

Example: Let's consider the grammar G_l responsible for generating the structure of a book. This grammar consists of 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:

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

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, making it a context-free language.

Definition

A language is considered *context-free* if there exists a context-free grammar that generates it.

Two grammars, denoted as G and G' are *equivalent* if they both generate the same language.

2.3 Erroneous grammars

Definition

A grammar G is called *clean* (or reduced) if and only if for every nonterminal A:

- A is reachable from the axiom S, and hence contribute to the generation of the language. That is, there exists a derivation:

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

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

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

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

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

1. Establish the set UNDEF, which comprises undefined nonterminals.

2. Identify the set of unreachable nonterminals.

Phase one

We define the set DEF as follows:

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

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

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

During each iteration, two cases may occur:

1. New nonterminals are discovered, and they have all 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.

Phase two

The produce relation, denoted as A produce B, holds if and only if there exists a production rule $(A \to \alpha B \beta) \in P$, where $A \neq B$ and α, β can be any strings.

We can now state that a nonterminal C is reachable from the start symbol S if and only if there exists a path in the graph of the produce relation from S to C. Nonterminals that are not reachable from the start symbol can be eliminated.

Additional requirement

In addition to the above cleanliness conditions, a third requirement is often added:

3. G must not allow for circular deviations because they are non-essential and may introduce ambiguity.

A circular derivation occurs when given $A \overset{+}{\Longrightarrow} A$, the derivation $A \overset{+}{\Longrightarrow} x$ is possible, and also $A \overset{+}{\Longrightarrow} A \overset{+}{\Longrightarrow} x$ (and many others) are possible.

It's important to note that even if a grammar is clean, it can have redundant rules that lead to ambiguity.

Example : Examples of unclean grammars are as follows:

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

In the first case, the axiomatic rule does not produce any phrase. In the second case, A is not reachable. In the third case, the grammar is circular on S and A.

2.4 Recursion and language infinity

Recursive grammars are essential for generating infinite languages.

Definition

> A derivation $A \overset{n}{\Longrightarrow} xAy$ is *recursive* if $n \geq 1$.
> If $n = 1$ the derivation $A \overset{n}{\Longrightarrow} xAy$ is called *immediately recursive*.
> The symbol A in the derivation $A \overset{n}{\Longrightarrow} xAy$ is called *recursive nonterminal*.
> If $x = \varepsilon$, the derivation $A \overset{n}{\Longrightarrow} xAy$ is called *left recursive*.
> If $y = \varepsilon$, the derivation $A \overset{n}{\Longrightarrow} xAy$ is called *right recursive*.

It's important to note that a grammar can be recursive without being circular.

The necessary and sufficient condition for language $L(G)$ to be infinite is that, assuming G is clean and devoid of circular derivations, G allows for recursive derivations.

Proof necessary condition : If no recursive derivation was possible, then every derivation would have a limited length, hence $L(G)$ would be finite. ∎

Proof 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. Furthermore, G clean implies:

- $S \overset{*}{\Longrightarrow} uAv$, which means A is reachable from S.

- $A \overset{+}{\Longrightarrow} w$, which means derivation from A terminates successfully.
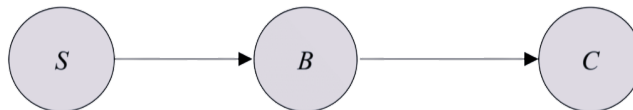
Therefore, there exist nonterminals that generate an infinite language. ∎

Property 2.1. 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 shown below:

This graph is acyclic, which indicates that the grammar is not recursive.

2.5   Syntax trees and canonical derivations

Definition

> A *syntax tree* is a directed, ordered graph with no cycles, in which nodes are arranged from left to right, and for any pair of nodes, there is only one path connecting them.

The key features of a syntax tree include:

- It visually represents the derivation process.

- It has relationships such as parent-child, descendants, root node, and leaf (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.

From a syntax tree, various subtrees can be defined 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 the sum and the product, as follows:
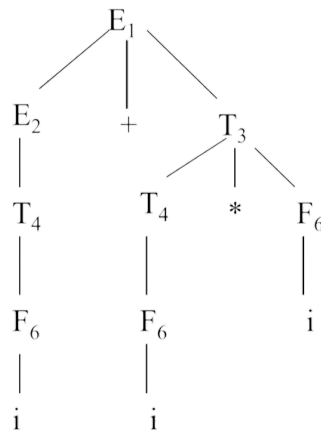
It can also be written in a linear form:

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

We can have right (expands at each step the rightmost non-terminal) and left derivation (expands at each step the leftmost non-terminal). For a given syntax tree of a sentence, there exists a unique right derivation and a unique left derivation that correspond to that tree. Both right and left derivations are valuable for defining parsing algorithms.

The ambiguity of a grammar is determined by examining whether a given sentence has a unique syntax tree or not.

To construct a correct syntax tree, it's important to keep in mind the following:

- Nonterminals for low-precedence operators are derived first.

- Nonterminals for high-precedence operators are derived later.

Definition

A *skeleton tree* is a syntax tree that preserves only the frontier and the structure.

A *condensed skeleton tree* is a syntax tree where the internal nodes on a non-branching paths are merged.
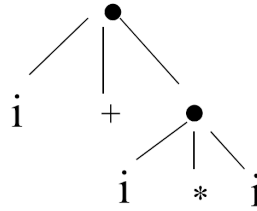
Example : The syntax tree from the previous example can be transformed into a skeleton tree:
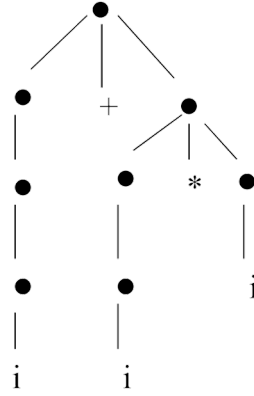
With the corresponding linear form:

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

It can also be transformed into a condensed skeleton tree:

With the relative linear form:

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

2.6   Parenthesis languages

Many artificial languages include parenthesized or nested structures, formed by matching pairs of opening and closing marks. These parentheses can be nested, meaning that inside a pair, there can be other parenthesized structures (recursion). Nested structures can also be placed in sequences at the same level of nesting. This paradigm, abstracted from concrete representation and content, is known as a Dyck language.

Example : For example, an alphabet of a Dyck language could be $\Sigma = \{'(',')','[',']'\}$, and a valid sentence over this alphabet is ()[[()[]]().

2.7   Regular composition of context-free languages

The context-free languages are closed under union, concatenation, and star.

Let $G_1 = (\Sigma_1, V_1, P_1, S_1)$ and $G_2 = (\Sigma_2, V_2, P_2, S_2)$ be the grammars defining languages L_1 and L_2. Le's also suppose that $V_{N_1} \cap V_{N_2} = \varnothing$ and $S \notin (V_{N_1} \cup V_{N_2})$.

Union

The union $L_1 \cup L_2$ is defined by the grammar containing the rules of both grammars, plus the initial rules $S \to S_1|S_2$. In formulas, the grammar is:

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

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

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

Those two languages are defined by the following two grammars:

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

The union language is defined with the rule:

$$S \to S_1|S_2$$

It's worth noting that the nonterminal sets of grammars G_1 and G_2 are distinct.

If the nonterminals in the grammars are not disjoint, it means that they have some common nonterminals. In this case, the grammar generates a superset of the union language, which results in spurious additional sentences being generated.

Concatenation

The concatenation $L_1 L_2$ is defined by the grammar containing the rules of both grammars, plus the initial rule $S \to S_1 S_2$. The grammar is:

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

Star

The grammar G of the star language $(L_1)^*$ is obtained by adding to G_1 and rules $S \to SS_1|\varepsilon$.

Cross

The grammar G of language $(L1)^+$ is obtained by adding to G_1 and 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, which is obtained by reversing the right-hand side of the rules.

2.8   Ambiguity

Definition

> A sentence x defined by grammar G is *ambiguous* if it admits several distinct syntax trees. In such cases, we say that the grammar G is ambiguous.
>
> The *degree of ambiguity* of a sentence x of a language $L(G)$ is the number of distinct syntax trees compatible with G. For a grammar the degree of ambiguity is the maximum among the degree of ambiguity of its sentences.

The problem of determining whether a grammar is ambiguous or not is undecidable because there is no general algorithm that, for any context-free grammar, can guarantee a termination with the correct answer in a finite number of steps. As a result, proving the absence of ambiguity in a specific grammar typically requires a case-by-case analysis, often done manually through inductive reasoning, which involves analyzing a finite number of cases. To demonstrate that a grammar is ambiguous, one can provide a witness, which is an example of an ambiguous sentence generated by the grammar. Therefore, it is advisable to strive for unambiguous grammar designs from the outset to avoid potential issues related to ambiguity.

Ambiguity can be categorized into various classes as outlined below.

Ambiguity from bilateral recursion

A non-terminal symbol A exhibits bilateral recursion when it 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. It's worth noting that 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, which is $E \to i + E | i$.
- A left-recursive grammar, which is $E \to E + i | i$

Example : Let's examine grammar G_2:

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

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

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:

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

Ambiguity from language union

If $L_1 = L(G_1)$ and $L_2 = L(G_2)$ share some sentences, and if a grammar G is constructed for their union language, it becomes ambiguous.

For any sentence $x \in L_1 \cap L_2$, it allows two distinct derivations: one following the rules of G_1 and the other following the rules of G_2. This ambiguity persists when using a single grammar G that includes all the rules.

However, for sentences belonging exclusively to $L_1 \setminus L_2$ and $L_2 \setminus L_1$, they are nonambiguous. To resolve this ambiguity, a solution is to provide separate sets of rules for $L_1 \cap L_2$, $L_1 \setminus L_2$ and $L_2 \setminus L_1$.

Inherent ambiguity

A language is considered inherently ambiguous when all of its grammars are ambiguous.

Example: Let's 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 defined by two grammars:

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

The union grammar of these two grammars is ambiguous. This observation leads to the intuitive conclusion that any grammar for the language L is also ambiguous due to the ambiguity of the language itself.

Ambiguity from concatenation of languages

Ambiguity arises in the concatenation of languages when there exists a situation where a suffix of a sentence in the first language also serves as a prefix of a sentence in the second language.

To eliminate this ambiguity, one must avoid situations where a substring from the end of a sentence in the first language is seamlessly connected to the beginning of a sentence in the second language. An effective solution to this problem is to introduce 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 the concatenation introduces ambiguity, we can resolve it by adding a new terminal symbol, denoted as $\#$. The axiomatic rule can then be transformed as follows:

$$S \to S_1 \# S_2$$

However, it's essential to note that this modification also alters the language itself.

Other cases of ambiguity

There are other, less significant cases of ambiguity, including:

- Ambiguity in regular expressions: to resolve this, eliminate redundant productions from the rule.

- Lack of order in derivations: address this problem by introducing a new rule that enforces the desired order.

2.9 Strong and weak equivalence

Definition

> Two grammars are *weakly equivalent* if they generate the same language, expressed as:
>
> $$L(G) = L(G')$$

It's important to note that with weak equivalence, two grammars can generate the same language but still produce different syntax trees. The structural aspect is crucial, as it is utilized by translators and interpreters.

Definition

> Two grammars are *strongly equivalent* if they not only generate the same language but also produce identical condensed skeleton trees.

Consequently, it follows that strong equivalence encompasses weak equivalence.

Furthermore, it is worth noting that the problem of strong equivalence is decidable, whereas the problem of weak equivalence is undecidable.

2.10 Grammar normal forms and transformations

Grammars normal forms constrain the rules without reducing the family of generated languages. They are useful for both proving properties and for language design.

Let us see some transformations useful both to obtain an equivalent normal form and design the syntax analyzers.

Nonterminal expansion

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

Example : Consider the grammar:

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

With the expansion of the nonterminal B we obtain:

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

Elimination of the axiom from right parts

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

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. It is composed of the following logical clauses, to be applied until a fixed point is reached:

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

The construction of the non-nullable form consist in:

1. Compute the Null set.

2. For each rule within P add as alternatives those obtained by deleting, in the right part, the nullable non-terminals.

3. Remove all empty rules, except for $A = S$.

4. Clean the grammar and remove any circularity.

The normal form without nullable nonterminals needs 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.

Copy rules and their elimination

The copy rules are used to factorize common parts, and they reduce the size of the grammar. However, copy elimination shortens derivations and reduces the height of the syntax trees.

The typical trade off is to define $\text{Copy}(A) \subseteq V$ set on nonterminal 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 the copy rules we have to:

1. Computation of Copy (assume a grammar with non-empty rules) by applying logical clauses until a fixed point is reached. That is the reflexive transitive closure of the copy relation defined by the copy rules:

$$C \in \text{Copy}(A) \text{ if } (B \in \text{Copy}(A)) \wedge (B \rightarrow C \in P)$$

2. Definition of the rules of a grammar G', equivalent to G but without copy rules. We remove the copy rules:

$$P' := P \backslash \{A \rightarrow B | A, B \in V\}$$

And we add the compensating rules:

$$P' := P' \cup \{A \rightarrow \alpha | \exists B (B \in \text{Copy}(A) \wedge (B \rightarrow \alpha) \in P)\}$$

The set of rule may increase considerably in size.

Conversion of left recursion to right recursion

Grammars with no left recursion are necessary for designing top-down parser.

To change from left recursion to right recursion we can have multiple possibility. The main case is the conversion of immediate left recursion:

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

Where $\beta_i \neq \varepsilon \; \forall i$. We can transform this grammar 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 we have right recursion since the string is generated from the left.

Chomsky normal form

The Chomsky normal form consist of two types of rules:

1. Homogeneous binary rules: $A \to BC$ with $B, C \in V$.

2. Terminal rules with singleton right part: $A \to a$ with $a \in \Sigma$.

Syntax tree 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: $S \to \varepsilon$.

- Then apply iteratively the following process:

  - For each rule type $A_0 \to A_1 A_2 \dots A_n$.
  - Add the rule type $A \to \langle A_1 \rangle \langle A_2 \dots A_n \rangle$.
  - And also another rule $\langle A_2 \dots A_n \rangle \to A_2 \dots A_n$.

  After some iterations A will be terminal, which means that we obtain $\langle A_1 \rangle \to A_1$.

Real-time normal form

In the real-time normal form we have that 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 a syntax analysis: every step reads and consumes one terminal symbol. With this normal form we have that the number of steps for the analysis is proportional to the length of the string.

Greibach normal form

In the Greibach normal form we have that 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 class of EBNF is useful to construct grammars that are more readable thanks to star, cross and union operators.

These grammars also allow for the definition of syntax diagrams which can be viewed as a blueprint of the syntax analyzer flowchart.

Note that since the context-free family is closed under all regular operations, therefore the generative power of EBNF is the same as that of BNF.

Definition

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

The BNF grammar is longer and less readable than an EBNF. Furthermore, the choice of nonterminal symbols names can be arbitrary.

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

Definition

Given string 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 does not contain regular expressions' operators nor parenthesis. Only string e is a regular expression, but it does not appear in the derivation if it is not terminal.

With EBNF we have unbounded node degree. As a result, the tree is in general wider and reduced in depth.

2.12   Comparison of regular and context-free languages

Regular languages are a special case of free languages that are generated with strong constraints on the form of rules. Due to these constraints the sentences of regular languages present inevitable repetitions. The rules used to transform a regular expression into a grammar that generates the same regular language are the following:

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

In general, we have that the regular expressions are a subset of the context-free language:

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

Definition

> A grammar is *unilinear* if and only if its rules are either all right-linear or all left-linear.

We can require that a unilinear grammar follows these constraints:

- Strictly unilinear rules: with at most one terminal $A \rightarrow aB$ with $A \in (\Sigma \cup \varepsilon)$ and $B \in (V \cup \varepsilon)$.

- All terminal rules are empty.

Therefore, we can assume only rules $A \rightarrow aB|\varepsilon$ for the right case and $A \rightarrow Ba|\varepsilon$ for the left case.

It is possible to demonstrate that the regular expressions can be translated into strictly unilinear grammars. Therefore, the regular language set is a subset of unilinear grammars: $\text{REG} \subseteq \text{UNILIN}$. We can also show that from any unilinear grammar one can obtain an equivalent regular expression: $\text{UNILIN} \subseteq \text{REG}$. As a result we have that:

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

Due to this property we can see the rules of the unilinear right grammar as equations, where the unknowns are the languages generated by every 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: we have a rule $P : A \rightarrow \varepsilon$.

2. $x = ay$: we have a rule $P : A \rightarrow aB$ and $y \in L_B$.

For every nonterminal A_0 defined by $A_0 \rightarrow a_1A_1|a_2A_2|\ldots|a_kA_k|\varepsilon$ we have $L_A = a_1L_{a_1} \cup a_2L_{a_2} \cup \cdots \cup a_kL_{a_k} \cup \varepsilon$. Therefore, we obtain a system of $n = |V|$ equations in n unknowns to be solved with the method with substitution and by applying the Arden identity.

Definition (*Arden identity*)

> Equation $KX \cup L$, with K nonempty language and L any language, has exactly one solution, which is.
> $$X = K^*L = KK^*L \cup L$$

Example : Consider the grammar:

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

This grammar can be transformed into a system of equation as follows:

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

By substituting the second equation into the first one, and the applying the concatenation operation of the union operator we obtain:

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

We can now apply the Arden identity, obtaining:

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

We can note that regular languages exhibits inevitable repetitions.

Property 2.2. 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) so that, for all $i \geq 1$, the string $tu^n v \in L(G)$.

In other words, 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. Then, it is also possible to derive tv, tuv, $tuuv$, etc. ■

This property is useful to demonstrate whether a grammar generates a regular language or not.

A grammar generates a regular language only if it has no self-nested derivations. Note that the inverse is not necessarily true: a regular language may be generated by a grammar with self-nested derivations. The lack of self-nested derivations allows solving language of equations of unilinear grammars.

In the context-free languages all sufficiently long sentences necessarily contain two substring that can be repeated arbitrarily many times, thus originating self-nested structures. This hinders the derivation of string with three or more parts that are repeated the same number of times (e.g., $a^n b^n c^n$). As a result, the language of three or more power is not context-free Therefore, the language of copies is also not context-free.

Closure properties

The regular language is closed under reverse, star, complement, union, and intersection operators. On the other hand, the context-free language is closed under reverse, star, and union operators.

We can also prove that the intersection between a context-free language and a regular language is still part of the context-free language.

To make a grammar more selective one can filter it through a regular language. The result of this filtering is always context-free.

Finite state automata

3.1 Introduction

We consider the problem of recognizing whether a string belongs to a given language before doing the semantic analysis. Automata are abstract machines used to describe a string recognition procedure.

For this problem, the input domain is a set of strings of alphabet Σ. The application of a recognition algorithm α to a given string x is denoted as $\alpha(x)$. We say string x is accepted if $\alpha(x) = $ yes, otherwise it is rejected. The language recognized, $L(\alpha)$, is the set of accepted strings:

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

If the language is semidecidable, it may happen that for some incorrect string x the algorithm will not terminate. In practice, we do not have to worry about such decidability issues because in language processing the only language families of concern are decidable.

In the theory and practice of formal languages, one computation step is a single atomic operation of the abstract recognition machine (automaton), which can manipulate only one symbol at a time. Therefore, it is customary to present the recognition algorithm by means of an automaton of some kind, no matter if a recognizer or a transducer machine, mainly for the following reasons:

1. Outlining the correspondence between the various families of languages and the respective generative devices.

2. Skipping any unnecessary and premature reference to the effective implementation of the algorithm in some programming language.
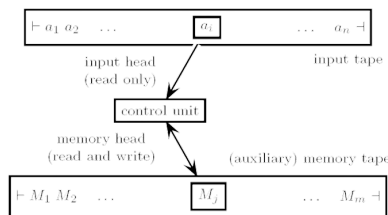

Figure 3.1: General model of a recognizer automaton

The automata analyzes the input string and executes a series of moves. Each move depends on the symbols currently pointed by the heads and also on the current state. The move may have the following effects:

- Shifting the input head of one position to the left or right.

- Replacing the current memory symbol by a new one and shifting the memory head of one position to the left or right.

- Changing the current state.

Depending on the type of automata it is possible to have:

- One-way automaton: the input head can be shifted only to the right.

- No auxiliary memory: this is the finite state automata. It is the machine model that recognizes the regular languages.

- Auxiliary memory: this is the pushdown automata. It is the machine model that recognizes the free languages.

Definition

A *configuration* is the set of the three components that determine the behavior of the automaton:

- The still unread part of the input tape.

- The contents and position of the memory tape and head, respectively.

- 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 only contains a special initial symbol. The automaton configuration changes through a series of transitions, each of which is driven by a move. The whole series of transitions is the computation of the automaton.

Definition

An automaton has a *deterministic* behavior if in every instantaneous configuration, at most one move is possible. Otherwise, the automaton is said to be *non-deterministic*.

In the final configuration, the control unit is in a special state qualified as final 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: to be empty or to contain only one special final symbol.

Definition

A source string x is *accepted* by the automaton if it starts from the initial input configuration $\vdash x \dashv$, executes a series of transitions and reaches a final configuration.

The set of all strings accepted by the automaton constitutes the language recognized by the automaton. If the automaton is non-deterministic, it may reach the same final configuration in two or more different sequences of transitions, or it may even reach two or more different final configurations.

The computation terminates either because the automaton has reached a final configuration or because it cannot execute any more transition steps, due to the fact that in the current instantaneous configuration there is not any possible move left. In the former case the input string is accepted, in the latter case it is rejected.

Definition

> Two automata that accept the same language are said to be *equivalent*.

Regular languages, which are recognized by finite state automata, are a subfamily of the languages recognizable in real time by a Turing machine. On the other hand, context-free languages are a subfamily of the languages recognized by a Turing machine that have a polynomial time complexity. Many applications of computer science and engineering make use of finite state automata: digital design, theory of control, communication protocols, the study of system reliability and security, etc.

3.2 Finite state automata

Definition

> A *finite state automaton* (FSA) consists of the following three elements:
>
> 1. The input tape, which contains the input string $x \in \Sigma^*$.
>
> 2. The control unit and its finite memory, which contains the state table.
>
> 3. The input head, initially positioned at the start marker of string x, which is shifted to the right at every move, as far as it reaches the end-marker of string x or an error happens.

After reading an input character, the automaton updates the current state of the control unit.

After scanning the input string x, the automaton recognizes string x or rejects it, depending on the current state.

Definition

> 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 by an input symbol, and it represents the move enabled when the current state matches the source state of the arc and the current input symbol matches the arc label. The state-transition graph has a unique initial state, but it may have none, one or more final states.

The graph can be represented in the form of an incidence matrix. Each matrix entry is indexed by the current state and by the input symbol, and contains the next state. Such an incidence matrix is often called state table. It is possible to use a syntax diagram, that is the dual of the state-transition graph (nodes are transformed into vertices and the other way around).

Example : Given the language over the alphabet $\Sigma = \delta \cup \{0, \cdot\}$, where $\delta = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ can be used to generate the decimal numbers. The regular expression to do so is as follows:

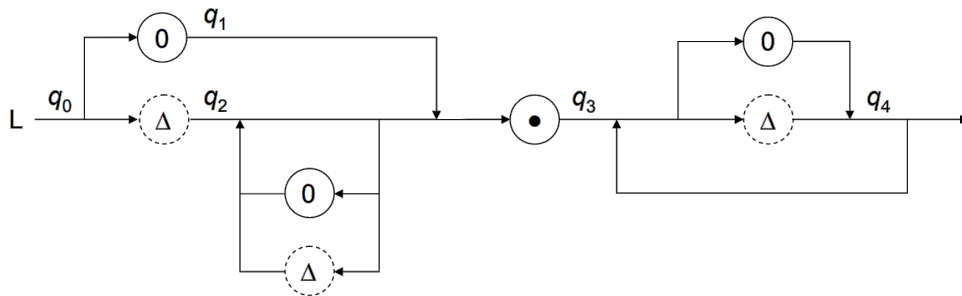$$e = (0 \cup \delta(0 \cup \delta)^*) \cdot (0 \cup \delta)^+$$

The corresponding state-transition graph is as follows:
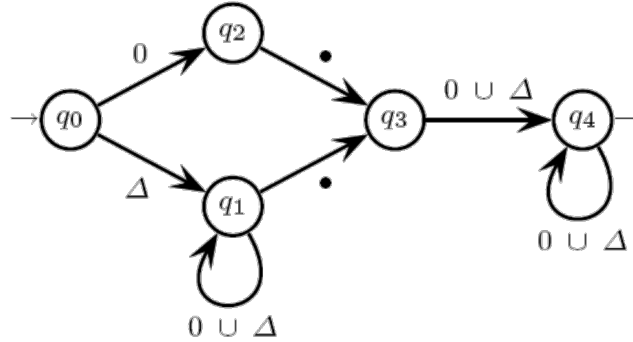
And the state-transition table is:

Current state	Current character				
	0	1	$\ldots$	9	$\cdot$
$\rightarrow q_0$	q_2	q_1	$\cdots$	q_1	–
q_1	q_1	q_1	$\cdots$	q_1	q_3
q_2	–	–	$\cdots$	–	q_3
q_3	q_4	q_4	$\cdots$	q_4	–
$q_4 \rightarrow$	q_4	q_4	$\cdots$	q_4	–

The syntax diagram is:

3.3 Deterministic finite state automata

Definition

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.

The transition function encodes the automaton moves:

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

This notation indicates that when M is in the current state q_i and reads the input symbol a, it switches the current state to q_j. If $\delta(q_i, a)$ is undefined, then M enters an error state and rejects the input string. The general transition function has the following domain $Q \times \Sigma^*$, and it is defined as:

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

Definition

A string is *recognized* if and only if, when the automaton moves through a path labeled by x, it starts from the initial state and ends at one of the final states:

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

Note that the empty string is accepted if and only if the initial state is final as well.

Definition

The languages accepted by such automata are called *finite-state recognizable*:

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

Two automata are *equivalent* if they accept the same language.

The time complexity of finite state automata is optimal: the input string x is accepted or rejected in real-time. Since it takes exactly as many steps to scan the string from left to right, the recognition time complexity could not be lower than this.

Error state and total automata

If the move is not defined in state q when reading character a, we say that the automaton falls into the error state q_{err}:

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

It is always possible to complete the deterministic automaton by adding the error state, without changing the accepted language.

Clean automata

An automaton may contain useless parts not contributing to any accepting computation, which are best eliminated.

Definition

A state q is *reachable* from state p if a computation exists going from p to q.
A state is *accessible* if it can be reached from the initial state.

> A state is *post-accessible* if a final state can be reached from it.
> A state is called *useful* if it is accessible and post-accessible.
> An automaton is *clean* if every state is useful.

Property 3.1. Every finite state automaton has an equivalent clean form.

To reduce an automaton: first identify all the useless states, then strip them off the automaton along with all their incoming and outgoing arcs.

Minimal automata

Property 3.2. For every finite state language there exists one, and only one, deterministic finite state recognizer that has the smallest possible number of states, which is called the minimal automaton.

Definition

> 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 one is.

Two undistinguishable states can be merged and thus the number of states of the automaton can be reduced, without changing the recognized language. Undistinguishability as a relation is symmetric, reflexive, and transitive.

Definition

> The states p and q are *distinguishable* if, and only if:
>
> 1. p is final and q is not or vice versa.
>
> 2. $\delta(p,a)$ is distinguishable from $\delta(q,a)$.

Example : Consider the following deterministic automaton:
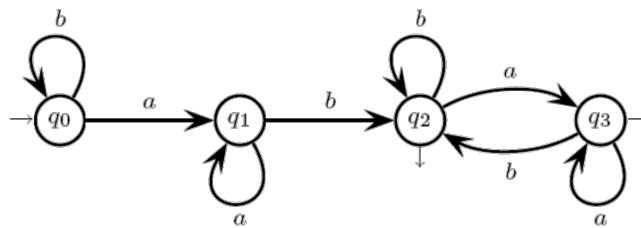
The corresponding undistinguishability table is as follows:

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

From the table is possible to see that the only undistinguishable states are q_2 and q_3.

Minimization

The minimal automaton M', equivalent to the given M, has for states the equivalence classes of the indistinguishability relation. To define the transition function of M', it suffices to state that there is 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 wit the same label:

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

That is, 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 undistinguishable states that are found in the undistinguishability table. The final automaton is the following:
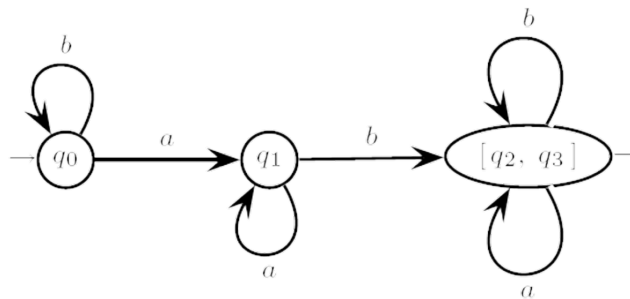
3.4   Nondeterministic automata

A right-linear grammar may contain two alternative rules starting with the same character. This means that in state A, reading the character, the machine can choose which one of the next states to enter: its behavior is not deterministic. A machine move that does not read an input character is termed spontaneous or an epsilon move. Spontaneous moves too cause the machine to be nondeterministic.

The main advantages of having nondeterminism are:

- The correspondence between grammars and automata suggest having:

  - Moves with two or more destination states.
  - Spontaneous moves (or ε-moves).
  - And two or more initial states.

- Concision: defining a language by means of a non-deterministic automaton may be more readable and compact than using a deterministic one.

Nondeterministic finite state automaton

Definition

A *non-deterministic finite automaton* N, without spontaneous moves, is defined by:

- 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 originates at state q_0 and ends at state q_n, and has labeling $a_1 a_2 \ldots a_n$

Definition

> An input string x is *accepted* by the automaton if it is the labeling of a path that starts from an initial state and ends to 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 non-deterministic automaton can be defined by means of a many-valued transition function.

For a machine $N = (Q, \Sigma, \delta, I, F)$, without spontaneous moves, the transition function δ s defined as to have the domain and image:
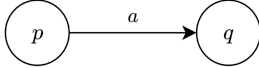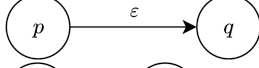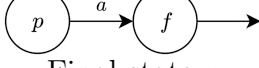
$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$$

where symbol $\mathcal{P}(Q)$ indicates the power set of set Q.

A non-deterministic automaton may have two or more initial states. But it is easy to construct an equivalent non-deterministic automaton with only one initial state. Add a new initial state q_0, connect it to the existing initial states by ε-arcs, make such states non-initial and leave only q_0.

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). We have that the following equivalences holds:

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

Example : Consider the following non-deterministic automaton:
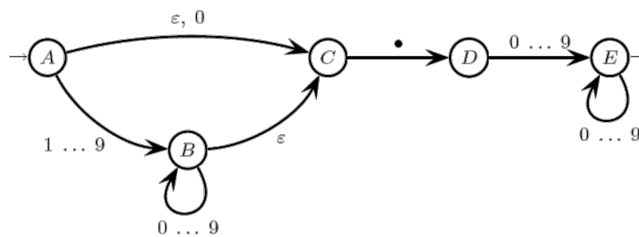
It can be easily translated into a grammar following the rules in the table above:

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

As a result we have that 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.

Ambiguity

As grammar derivations are in one-to-one correspondence with automaton computations, ambiguity is extensible to automata.

Definition

An automaton is *ambiguous* if, and only if, the corresponding grammar is so, i.e., if a string x labels two or more accepting paths.

Clearly it follows from the definition that a deterministic automaton is never ambiguous.

REG families can be defined also using left-linear grammars. By interchanging left with right, it is simple to discover the mapping between such grammars and automata.

## 3.5 From automaton to regular expression

Suppose for simplicity the initial state i is unique, and no arc enters in it; similarly the final state t is unique and without outgoing arcs. Every state other than i and t is called internal. We construct an equivalent automaton, termed generalized finite automaton, that allows arc labels to be also regular languages. Eliminate the internal nodes one by one, and after each elimination add one or more compensation arcs to preserve the equivalence of the automaton. Such new arcs are labeled by regular expressions. At the end only the nodes i and t are left, with only one arc from i to t. The regular expression that labels such an arc generates the complete language recognized by the original finite automaton.

The elimination order is not relevant. However, different orders may generate different regular expressions, all equivalent to one another but of different complexity.

**Example :** Consider the following automaton:
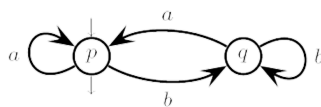
We have to normalize it to obtain the initial state and the final state, obtaining:
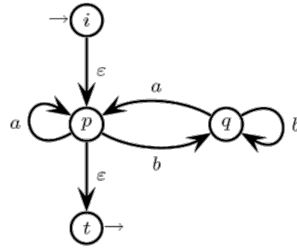
Finally, we can apply the Brzozowski and McCluskey method to the normalized automata. We do this in three steps that consist in:

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, obtaining: $a|bb^*a = b^*a$.

3. Remove the node p by replacing the label of the arc with $(b^*a)^*$
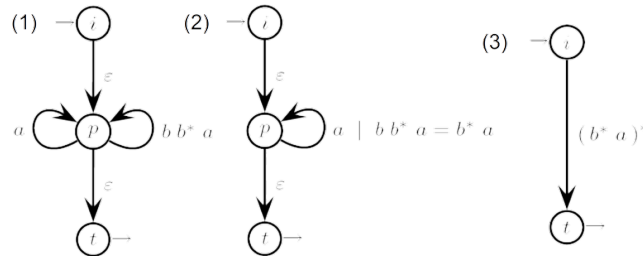
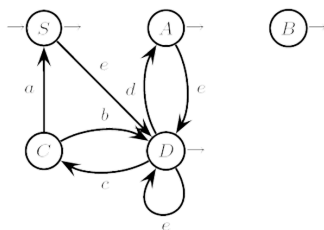## 3.6 Elimination of nondeterminism

Every non-deterministic finite automaton can always be transformed into an equivalent deterministic one. Consequently, every right linear grammar always admits an equivalent non-ambiguous right linear one. Thus, every ambiguous regular expression can always be transformed into a non-ambiguous one. The algorithm to transform a non-deterministic automaton into a deterministic one is structured in two phases:

1. Elimination of the spontaneous moves. As such moves correspond to copy rules, it suffices to apply the algorithm for removing the copy rules.

2. Replacement of the non-deterministic multiple transitions by changing the automaton state set. This is the well known subset construction.

Example: Given the following automaton:
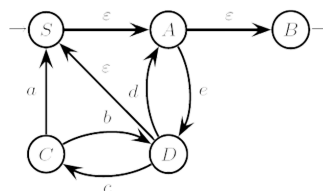
After applying the algorithm we have:

If after eliminating all the ε-arc the automaton is still non-deterministic, then go to the second phase.

3.7 From a regular expression to a finite state automaton

There are a few algorithms to transform a regular expression into an automaton, which differ as for automaton characteristic. The three main methods are:

1. Thompson (or structural method): it decomposes the regular expression into subexpressions, until it reaches the terminal symbols, and then it constructs the subexpression recognizers, connects them and builds up a network of recognizers that implement the union, concatenation and star operators.

2. Glushkov-McNaughton-Yamada: it constructs a nondeterministic recognizer without spontaneous moves, but with multiple transitions.

3. Berry-Sethi method: it constructs a deterministic recognizer without spontaneous moves, but of size often larger than Thompson.

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:
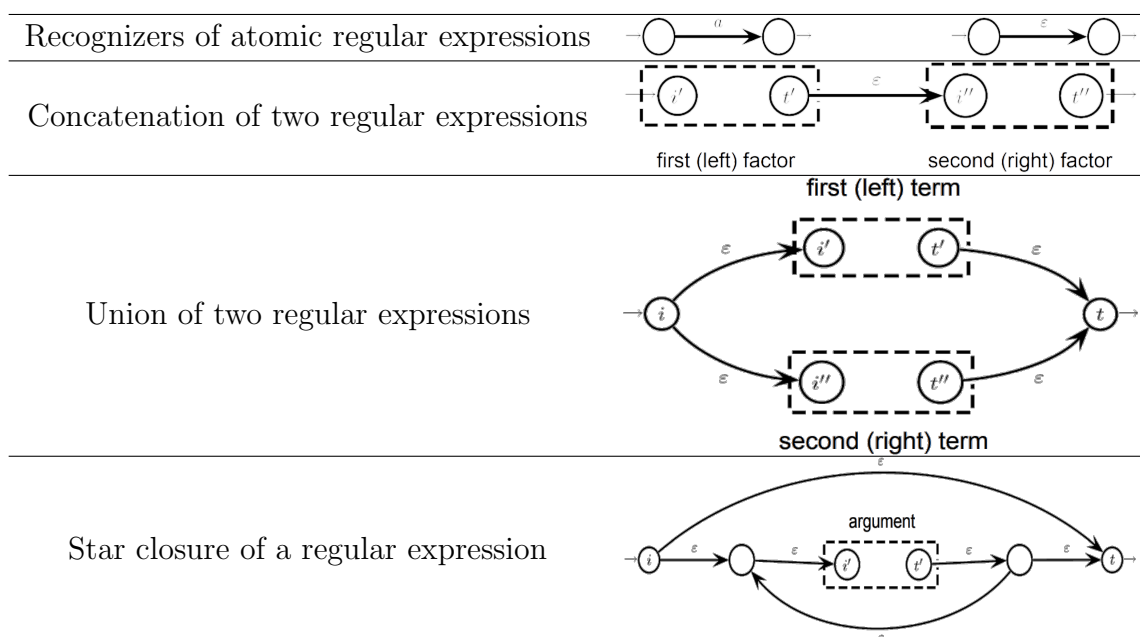
In general the outcome of the Thompson method is a non-deterministic automaton with spontaneous moves. The method is an application of the closure properties of the regular languages under the operations of union, concatenation and star.
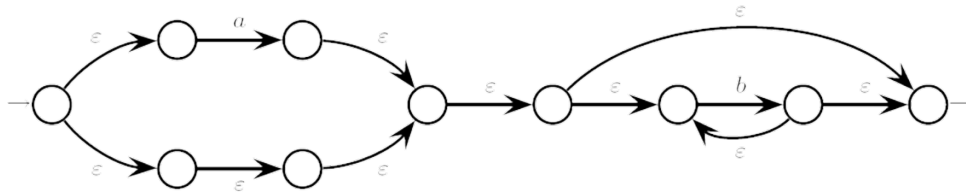
Example: Consider the regular expression $(a \cup \varepsilon)b^*$. It is possible to rewrite the same regular expression with symbols and subexpressions indexing, that is:

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

The corresponding structure tree is as follows:
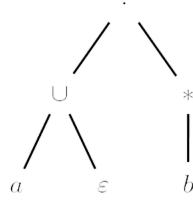
By applying the rules of the previous table we obtain the following automaton:

The automaton found can be optimized to avoid redundant states.

Glushkov-McNaughton-Yamada algorithm

The GMY algorithm constructs the automaton equivalent to a given regular expression, with states that are in a one-to-one correspondence with the generators that occur in the regular expression.

Given a language L over the alphabet Σ we can define:

- The set of initials: $\mathrm{Ini}(L) = \{a \in \Sigma | a\Sigma^* \cap L \neq \varnothing\}$.

- The set of finals: $\mathrm{Fin}(L) = \{a \in \Sigma | \Sigma^* a \cap L \neq \varnothing\}$.

- The set of digrams: $\mathrm{Dig}(L) = \{x \in \Sigma^2 | \Sigma^* x \Sigma^* \cap L \neq \varnothing\}$.

- The set of forbidden digrams: $\overline{\mathrm{Dig}(L)} = \Sigma^2 \backslash \mathrm{Dig}(L)$

To compute the sets of initials, finals, and digrams we use the following rules:

Set of initials
$\mathrm{Ini}(\varnothing) = \varnothing$
$\mathrm{Ini}(\varepsilon) = \varnothing$
$\mathrm{Ini}(a) = \{a\}$ for every character a
$\mathrm{Ini}(e \cup e') = \mathrm{Ini}(e) \cup \mathrm{Ini}(e')$
$\mathrm{Ini}(e \cdot e') = $ if $\mathrm{Null}(e)$ then $\mathrm{Ini}(e) \cup \mathrm{Ini}(e')$ else $\mathrm{Ini}(e)$
$\mathrm{Ini}(e^*) = \mathrm{Ini}(e^+) = \mathrm{Ini}(e)$

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

Set of digrams
$\text{Dig}(\varnothing) = \varnothing$
$\text{Dig}(\varepsilon) = \varnothing$
$\text{Dig}(a) = \varnothing$ 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

The language L is called *locally testable*, if and only if it satisfies the following identity:

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

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

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

To design the recognizer of a local language we scan the input string from left to right and check whether: the initial character belongs to the set Ini, every digram belongs to the set Dig, and the final character belongs to the set Fin. The string is accepted if, and only if, all the above checks succeed.

We can implement the above recognizer by resorting to a sliding window with a width of two characters, which is shifted over the input string from left to right. At each shift step the window contents are checked, and if the window reaches the end of the string and all the checks succeed, then the string is accepted, otherwise it is rejected. This sliding window algorithm is simple to implement by means of a nondeterministic automaton.

Given the sets Ini, Fin and Dig, the corresponding recognizer has:

- 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 contains the empty string, the initial state q_0 is final as well.

Example : The recognizer automaton for the language $L_1 = (abc)^*$ is as follows:
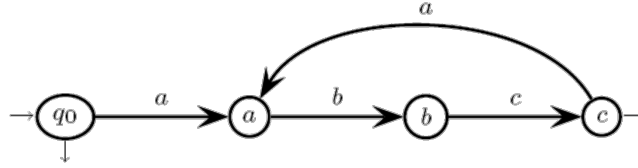
Definition

A regular expression is said to be *linear* if there is not any repeated generator.

Property 3.3. The languages generated by linear regular expressions are local.

Linearity implies the regular expression subexpressions are defined over disjoint alphabets. But a regular expression is the composition of its subexpressions, thus the language of a linear regular expression is local as a consequence of the closures of the local languages over disjoint alphabets. Notice that the opposite implication does not hold This implies that constructing the recognizer for a generic regular language reduces to the problem of finding the characteristic local sets Ini, Fin, Dig of such a generic language, provided the alphabet is slightly modified.

To check whether a regular expression e generates the empty string we can use the Null(e) operator, that is true when the empty string is in the regular expression, false otherwise. It works 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).

The idea of the GMY algorithm, based on the linear regular expressions is the following:

1. Enumerate the regular expression e and obtain the linear regular expression $e_\#$.

2. Compute the three characteristic local sets Ini, Fin and Dig of $e_\#$.

3. Design the recognizer of the local language generated by $e_\#$.

4. Cancel the indexing and thus obtain the recognizer of e.

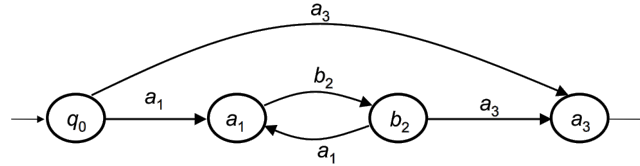**Example :** Consider the regular expression $e = (ab)^*a$. To apply the GMY algorithm we follow these steps:

1. Enumerate the regular expression, obtaining:
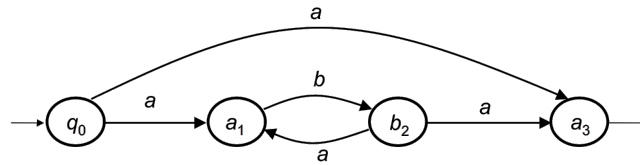
$$e_\# = (a_1 b_2)^* a_3$$

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, with as many states as the occurrences of generators in the regexp are, and one more state.

Berry-Sethi method

In order to obtain the deterministic recognizer, we can just apply the subset construction to the non-deterministic recognizer built by the GMY algorithm. However, there is a more direct algorithm called Berry-Sethi. The idea at the base of this algorithm is the following:

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 predicate Null and local sets Ini, Fin and Dig.

3. Define the set Fol as follows:

   (a) $\text{Fol}(c_\#) \in \mathcal{P}(\Sigma_\# \cup \{\dashv\})$.
   (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 following 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: Given the language $L = (a|bb)^*(ac)^+$ apply the BS algorithm. First we enumerate the string:
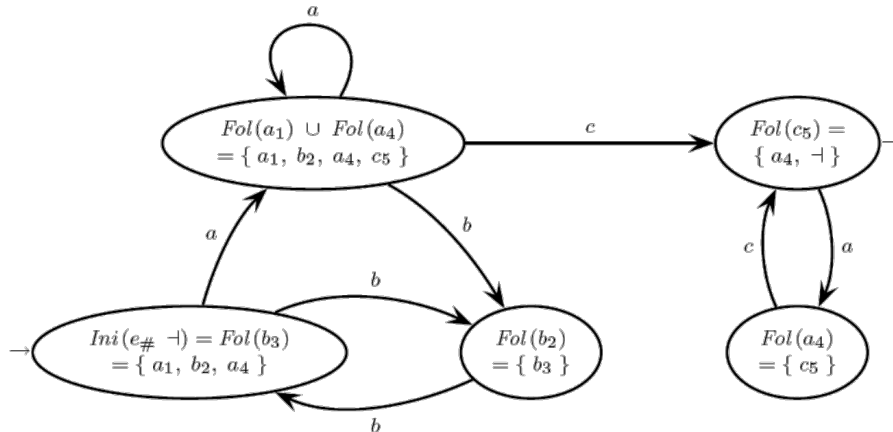
$$e_\# = (a_1|b_2 b_3)^*(a_4 c_5)^+ \dashv$$

The characteristic sets are:

- $Ini(e_\#) = \{a_1, b_2, a_4\}$.
- $Fin(e_\#) = \{\dashv\}$.
- $Dig(e_\#) = \{a_1 a_1, a_1 b_2, a_1 a_4, b_2 b_3, b_3 a_1, b_3 b_2, b_3 a_4, a_4 c_5, c_5 a_4, c_5 \dashv\}$.

The table of the followers is as follows:

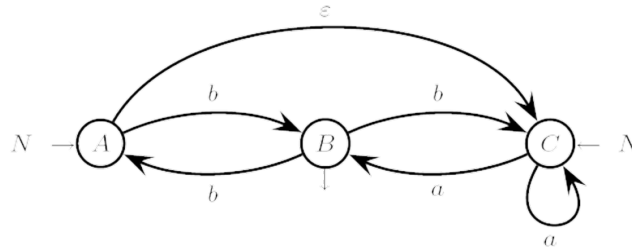| $c_\#$	$Fol(c_\#)$
a_1	$a_1 b_2 a_4$
b_2	b_3
b_3	$a_1 b_2 a_4$
a_4	c_5
c_5	$a_4 \dashv$

The resulting automaton is as follows:

The Berry-Sethi algorithm can be also used to transform a nondeterministic automaton into a deterministic one. The steps are:

1. Enumerate the elements on the arcs.

2. Create the followers table.

3. Recreate the automaton by using the followers table.

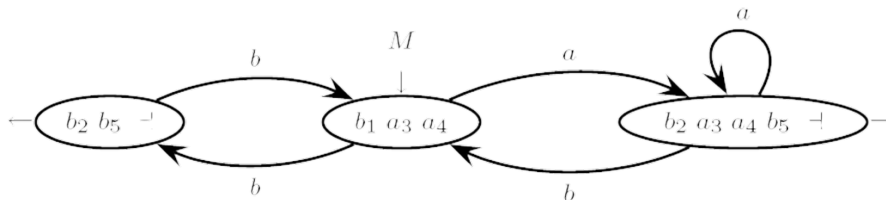**Example :** Consider the following automaton:

Its numbered version is the following:
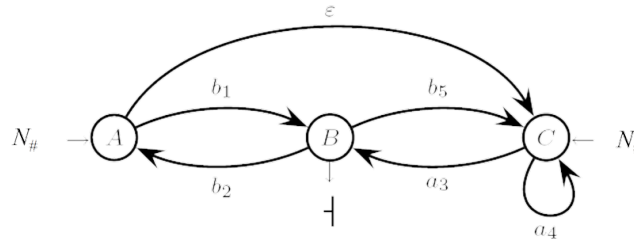
The follower table is:

$c_\#$	$\mathbf{Fol}(c_\#)$
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 final deterministic automaton is:

3.8   Complement and intersection of regular languages

Regular expressions may also contain the operators of complement, intersection and set difference, which are very useful to make the regular expression more concise. The family REG is closed under complement, intersection, and set difference.

Complement

The complement of a language L is defined as:

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

Assume the recognizer M of L is deterministic, with initial state q_0, state set Q, set of finals states F and transition function δ. To construct a deterministic automaton $\overline{M}$ of the complement language $\neg L$ we have to follow these steps:

1. Create the error state $p \notin Q$, so the states of $\overline{M}$ are $Q \cup \{p\}$.

2. The transition function $\overline{\delta}$ is:

   - $\overline{\delta}(q, a) = \delta(q, a)$, 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:

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

Note that a recognizing path of M ($x \in L(M)$) does not end into a final state of $\overline{M}$ and a non-recognizing path of M ($x \notin L(\overline{M})$) does not end into a final state of $\overline{M}$.
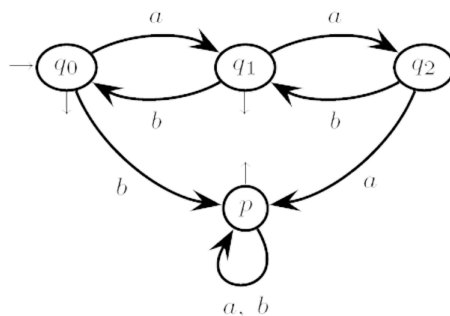
Example: Consider the following deterministic automaton:
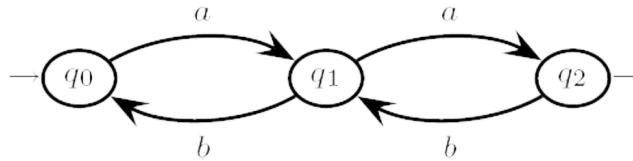
To find the complement automaton we have to add the error state p, obtaining the following automaton:
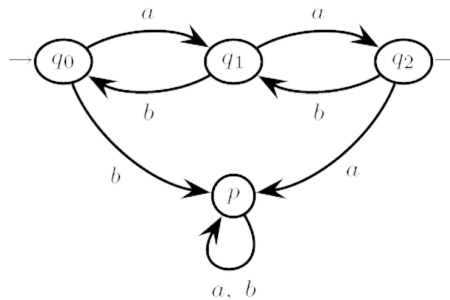
Finally, we can swap final and non-final states:

For the complement construction to work correctly, the original automaton must be deterministic, otherwise the original and complement languages may be not disjoint, which fact would be in violation of the complement definition. The complement automaton may contain useless states and may not be in the minimal form either; it should be reduced and minimized, if necessary.

Cartesian product

The product is a very common construction of formal languages, where a single automaton simulates the computation of two automata that work in parallel on the same input string. It is very useful to construct the intersection automaton. To obtain the intersection automaton we can resort to the De Morgan theorem:

1. Construct the deterministic recognizers of the two languages.

2. Construct the respective complement automata.

3. Construct their union (Thompson).

4. Make deterministic the union automaton (Berry-Sethi).

5. Complement again and thus obtain the intersection automaton.

Since the intersection of the two languages is recognized directly by the Cartesian product of their automata, we can obtain the intersection automaton directly. Suppose both automata do not contain any 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:

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

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 as follows:
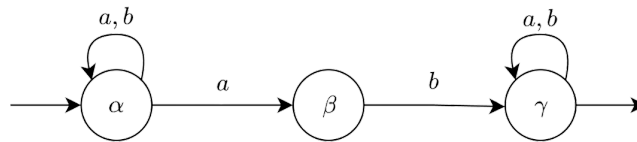
And the deterministic automaton for the language L'' is as follows:
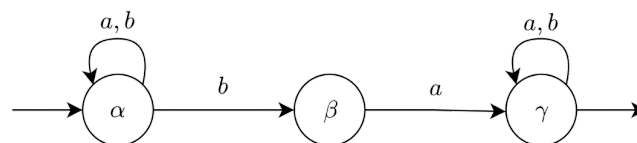
The intersection of the two is found with the following table:

Pushdown automata

4.1 Introduction

To recognize context-free languages we need an automaton with an auxiliary structured as an unbounded stack of symbols. The following operations apply to a stack:

- Push: places the symbol(s) onto the stack top.

- Pop: removes symbol from the stack top, if the stack is not empty; otherwise reads Z_0.
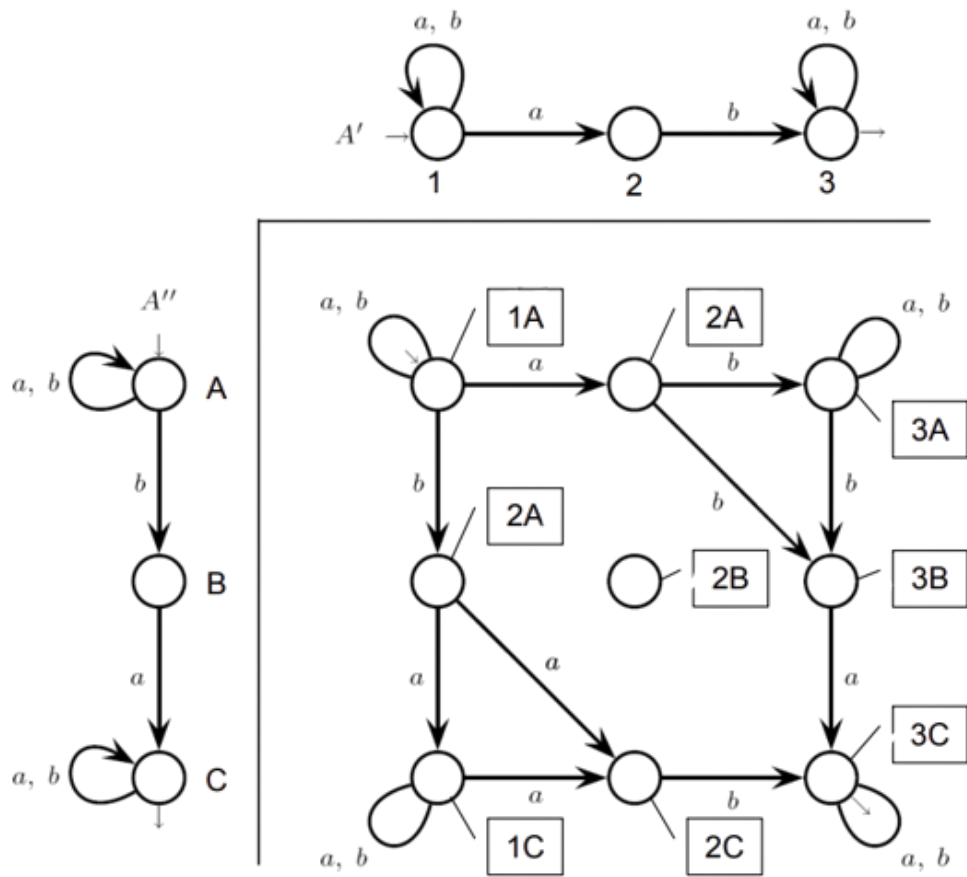
- Stack emptiness test: true if the stack is empty, false otherwise.

The symbol Z_0 is the stack bottom and can be read but not removed. The symbol $\dashv$ is the terminator character of the input string.

The configuration is specified by: current state, current character, and stack contents. With a move the pushdown automaton:

- Reads the current character and shifts the input head, or performs a spontaneous move without shifting the input head.

- Reads the stack top symbol and removes it from the top if the stack is not empty, or reads the stack symbol Z_0 if the stack is empty.

- Depending on the current character, state and stack top symbol, it goes into the next state and places none, one or more symbols onto the stack top.

Definition

A pushdown automaton M is defined by:

- Q a finite set of states of 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 has:

- Domain: $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$.

- Image: the set of the subsets of $Q \times \Gamma^*$.

The possible moves are:

- Reading move: in the state q with symbol Z_0 on the stack top, the automaton reads char a and enters one of the states p_i with $1 \leq i \leq n$, after orderly executing the operations pop and push (γ_i):
$$\delta(q, a, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \ldots, (p_n, \gamma_n)\}$$

- Spontaneous move: in the state q with symbol Z_0 on the stack top, the automaton does not read any input character and enters one of the states p_i with $1 \leq i \leq n$, after orderly executing the operations pop and push (γ_i):

$$\delta(q, \varepsilon, Z) = \{(p1, \gamma_1), (p2, \gamma_2), \ldots, (p_n, \gamma_n)\}$$

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

There is non-determinism: for a triple (state, input, stack top) there are two or more possible moves that consume none or one input character.

Definition

The *instantaneous configuration* of a machine M is a 3-tuple:

$$(q, y, \eta) \in Q \times \Sigma^* \times \Gamma^+$$

which specifies:

- q: the current state of the control unit.

- y: the part of the input string x that still has to be scanned.

- η: the current contents of the pushdown stack.

The *initial* configuration of machine M is:

$$(q_0, x, Z_0)$$

The *final* configuration of machine M is:

$$(q, \varepsilon, \lambda)$$

Applying a move, a transition from a configuration to another occurs, to be denoted as:

$$(q, y, \eta) \rightarrow (p, z, \lambda)$$

Note that a chain of one or more transitions is denoted by $\xrightarrow{+}$.

An input string x is accepted by final state if there is the following computation:

$$(q_0, x, Z_0) \xmapsto{*} (q, \varepsilon, \lambda)$$

where $q \in F$ and $\lambda \in \Gamma^*$, whereas there is not any specific condition for λ; sometimes λ happens to be the empty string, but this is not necessary.

State-transition graph

The transition function of a finite automaton can be graphically represented. The numerator indicates the characters read in the input tape and in the memory. The denominator indicates the replacement for the element in the stack memory.

Example: The language $L = \{uu^R | u \in \{a, b\}^*\}$ of the palindromes of even length is accepted with final state by the pushdown recognizer.
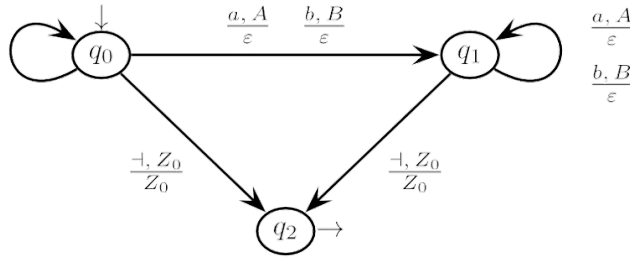
From grammar to pushdown automata

Grammar rules can be viewed as the instructions of a non-deterministic pushdown automaton. Intuitively such an automaton works in a goal-oriented way and uses the stack as a notebook of the sequence of actions to undertake in the next future.

The stack symbols can be both terminals and non-terminals of the grammar. If the stack contains the symbol sequence $A_1 \ldots A_k$, then the automaton executes first the action associated with A_k, which should recognize if in the input string from the position of the current character a_i there is a string w that can be derived from A_k; if it is so, then the action shifts the input head of $|w|$ positions.

An action can be recursively divided into a series of sub-actions, if to recognize the non-terminal symbol A_k it is necessary to recognize other non-terminals.

The initial action is the grammar axiom: the pushdown recognizer must check if 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 on the initial character of the input string. At every step the automaton chooses (non-deterministically) one applicable grammar rule and executes the corresponding move. The input string is recognized accepted when, and only when, it is completely 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 are summarized as follows:

Grammar rule	Automaton move
$A \rightarrow BA_1 \dots A_n$ with $n \geq 0$	If top $= A$ then pop Push $(A_n \dots A_1 B)$
$A \rightarrow bA_1 \dots A_n$ with $n \geq 0$	If $cc = b$ and top $= A$ then pop Push $(A_n \dots A_1)$ Shift reading head
$A \rightarrow \varepsilon$	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

This construction may lead to a nondeterministic automaton.

Example: Consider the language $L = \{a^n b^m | n \geq m \geq 1\}$. The nondeterministic grammar that generates the string of this language is as follows:

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

Using the rules of the previous table we can construct the pushdown automaton in the following way:

#	Grammar rule	Automaton move
1	$S \rightarrow aS$	If $cc = a$ and top $= S$ then pop Push S Shift reading head
2	$S \rightarrow A$	If top $= S$ then pop Push(A)
3	$A \rightarrow aAb$	If $cc = a$ and top $= A$ then pop Push bA Shift reading head
4	$A \rightarrow ab$	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: for every successful automaton computation there exists a grammar derivation and vice versa; thus the automaton simulates the leftmost derivations of the grammar. However, the automaton cannot guess the correct derivation; it has to examine all the possibilities. A string is accepted by two or more different computations if and only if the grammar is ambiguous.

The construction from grammar to automaton can be inverted and used to obtain the grammar by starting from the pushdown automaton, if this is one-state.

Property 4.1. The family of free languages generated by free grammars coincides with the family of the languages recognized by one-state pushdown automata.

Unfortunately in general the resulting pushdown automaton is non-deterministic, as it explores all the moves applicable at any point and has an exponential time complexity with respect to the length of the source string.

Varieties of pushdown automata

The acceptance modes can be:

1. By final state: accepts when enters a final state independently of the stack contents.

2. By empty stack: accepts when the stack gets empty independently of the current state.

3. Combined: by final state and empty stack.

Property 4.2. For the family of (non-deterministic) pushdown automata with states, the three acceptance modes listed above are equivalent.

A generic pushdown automaton may execute an unlimited number of moves without reading any input character. This happens if and only if it enters a loop made only of spontaneous moves. Such a behavior prevents it of completely reading the input string, or causes it to execute an unlimited number of moves before deciding whether to accept or reject the string. Both behaviors are undesirable in the practice. It is always possible to build an equivalent automaton with no spontaneous loops.

A pushdown automaton operates in on-line mode if it decides whether to accept or reject the string as soon as it reads the last character of the input string, and then it does not execute any other move. Clearly from a practical perspective the on-line mode is a desirable behavior. It is always possible to build an equivalent automaton that works in on-line mode.

Context-free languages and pushdown automata

The language accepted by a generic pushdown automaton is free. Every free language can be recognized by a non-deterministic one-state pushdown automaton.

Property 4.3. The family of context-free languages coincides with that of the languages recognized by unrestricted pushdown automata.

Property 4.4. The family CF of context-free languages coincides with that of the languages recognized by the one-state non-deterministic pushdown automata.

Intersection of regular and context-free languages

It is easy to justify that the intersection of a free and a regular language is free as well. Given a grammar G and a finite state automaton A, the pushdown automaton M that recognizes the intersection $L(G) \cap L(A)$ can be obtained as follows:

1. Construct the one-state pushdown automaton N that recognizes $L(G)$ by empty stack.

2. Construct the pushdown automaton M (with states), the state-transition graph of which is. The Cartesian product of those of N and A, by the Cartesian product construction so that the actions of M on the stack are the same as those of N.

The obtained pushdown automaton M:

1. As its states, has pairs of states of the component machines N and A.

2. Accepts by final state and empty stack (combined acceptance mode).

3. The states that contain a final state of A are themselves final.

4. Is deterministic, if both component machines N and A are so.

5. Accepts by final state all and only the strings that belong to the intersection language.
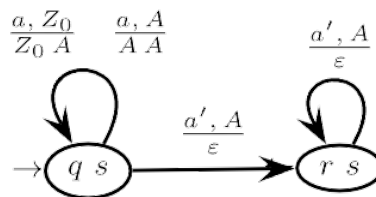
Example: Consider the Dyck language $L = \{a^n a'^n | n \geq 1\}$. It is possible to construct a finite state automaton that recognizes the language by final state, that is:
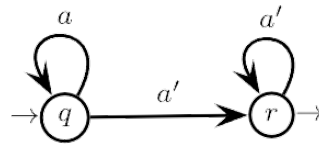
And also a pushdown automaton that recognizes the language by empty stack, that is:
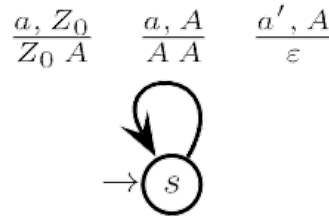
By applying the intersection we obtain a product pushdown automaton that recognizes by empty stack in the final state (r, s), that is:

4.2 Deterministic pushdown automata

Nondeterminism is absent if the transition function δ is one-valued and

- 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 every $a \in \Sigma$.

If the transition function does not exhibit any form of non-determinism, then the automaton is deterministic and the recognized language is deterministic, too. Note that a deterministic pushdown automaton may have spontaneous moves.

Property 4.5. The family DET of the deterministic free languages is strictly contained in the family CF of all the free languages:

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

Given L, D and R a language that belongs to the family CF, DET, and REG we have the following closure properties:

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	$D \cap R \in \text{DET}$	$D_1 \cap D_2 \notin \text{CF}$

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.

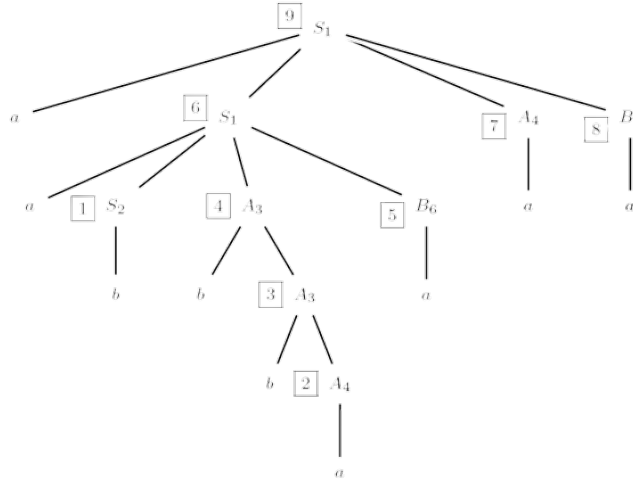**Example :** Consider the string *aabbbaaaa* generated by the grammar:

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

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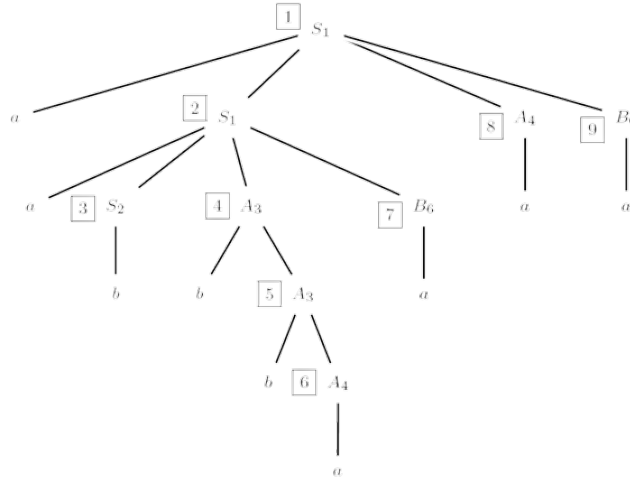
Note that the numbering indicates the application order of the rules.

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: $V = \{S, A, B, \ldots\}$.

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:

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

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:
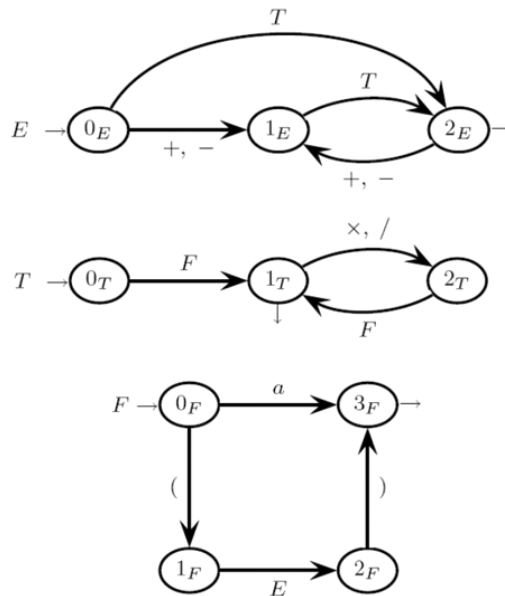
Note that the machine T has been normalized.

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:

```
 1: call T
 2: while cc = + do
 3:     cc := next
 4:     call T
 5: end while
 6: if cc ∈ {(⊣} then
 7:     return
 8: else
 9:     error
10: end if
```

The procedure for the grammar T is:

```
 1: call F
 2: while cc = × do
 3:     cc := next
 4:     call F
 5: end while
 6: if cc ∈ {(+ ⊣} then
 7:     return
 8: else
 9:     error
10: end if
```

The procedure for the grammar F is:

```
 1: if cc = a then
 2:     cc := next
 3: else if cc = ( then
 4:     cc := next
 5:     call E
 6:     if cc =) then
 7:         cc := next
 8:         if cc ∈ {)× ⊣} then
 9:             return
10:         else
11:             error
12:         end if
13:     else
14:         error
15:     end if
16: else
17:     error
18: 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:
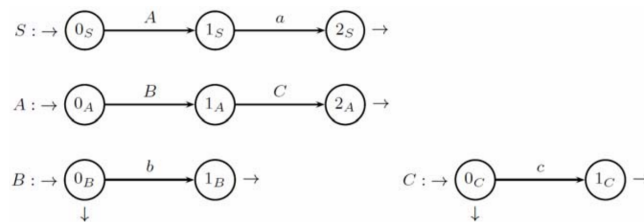
To find the set of initials for S_0 we have to check the following states:

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

The final result is:
$$\mathrm{Ini}(0_S) = \{b\} \cup \{c\} \cup \{a\}$$

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 closure$(C) = C$):

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

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 $R = I_0, I_1, \ldots$ and the state-transition function $\theta : R \times (\Sigma \cup V) \to R$ 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

1: $R' \leftarrow \{I_0\}$
2: **while** $R \neq R'$ **do**
3:      $R \leftarrow R'$
4:      **for** each m-state $I \in R$ and symbol $X \in \Sigma \cup V$ **do**
5:          $I' \leftarrow \text{closure}(\vartheta(I, X))$
6:          **if** $I' \neq \varnothing$ **then**
7:              add arc $I \xrightarrow{X} I'$ to the graph of ϑ
8:              **if** $I' \notin R$ **then**
9:                  add m-state I' to the set R'
10:              **end if**
11:          **end if**
12:      **end for**
13: **end while**

Example : Consider the grammar:

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

The corresponding machine net is:
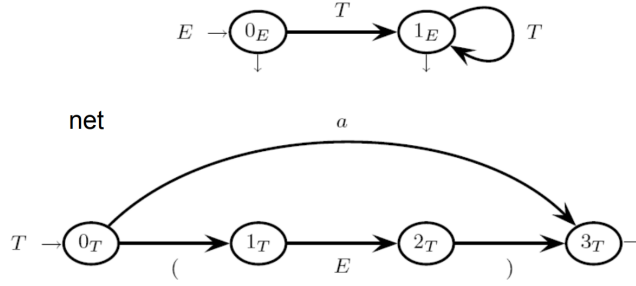
By applying the previous algorithm we obtain the following pilot graph:
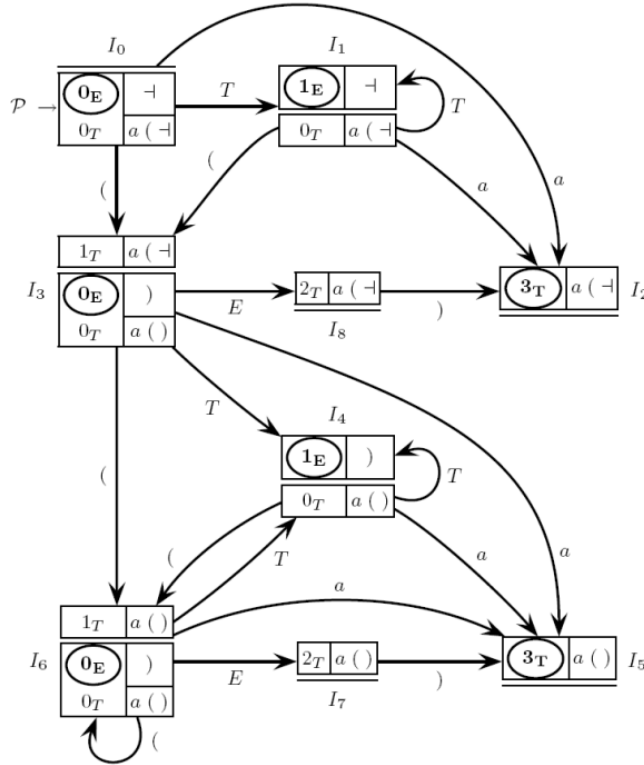
It is possible to note that the kernel-equivalent m-states are:

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

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 $(A \rightarrow a)$ is less or equal to n.

- The number of reductions of type null $(A \rightarrow \varepsilon)$ and copy $(A \rightarrow B)$ is linearly bounded by n.

- The number of reductions without any terminals $(A \rightarrow BC)$ 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:

$$ELL(1) \subset ELR(1)$$

Example: Consider the grammar:

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

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:

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

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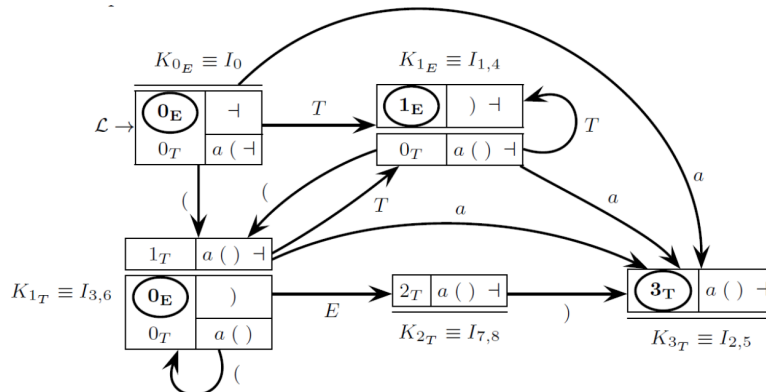
Parser control-flow graph

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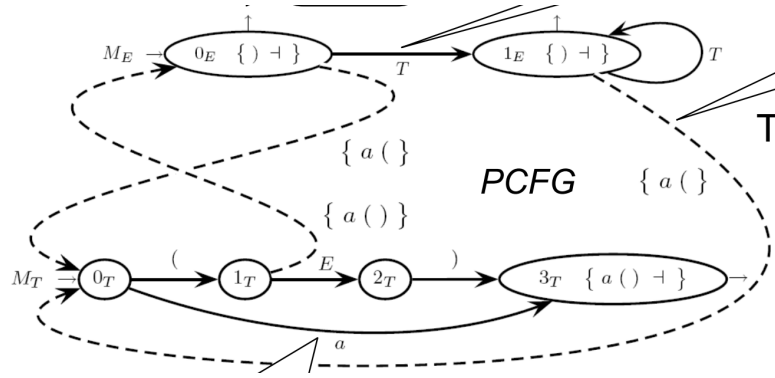
The character b is in the guide set, denoted as $b \in \mathrm{Gui}(q_A \dashrightarrow 0_{A_1})$, if one of the following properties holds:

- $b \in \mathrm{Ini}(L(0_{A_1}))$.

- A_1 is nullable and $b \in \mathrm{Ini}(L(r_A))$.

- 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 $p \overset{a}{\to} q$ 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 $cc \in \mathrm{Gui}(q_A \to 0_B)$.

- 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 $p \overset{a}{\to} q$ are interpreted as (unconditioned) return instructions from a machine.

Property 5.1. 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 $q_A \overset{cc}{\to} r_A$ 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 $q_A \overset{B}{\to} r_A$ 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:

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

For every non-initial state we have:

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

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):

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

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

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.

<div align="right">

CHAPTER **6**

</div>

Laboratory

6.1   Regular expressions

The basic character set of regular expression is:

Syntax	Matches
x	The x character
.	Any character except newline
$[xyz]$	x or y or z
$[a-z]$	Any character between a and z
$[^a-z]$	Any character except those between a and z

The composition rules are the following:

Syntax	Matches
R	The R regular expression
RS	Concatenation of R and S
R|S	Either R or S
R*	Zero or more occurrences of R
R+	One or more occurrences of R
R?	Zero or one occurrence of R
R$\{m,n\}$	A number of R occurrences ranging from n to m
R$\{n,\}$	n or more occurrences of R
R$\{n\}$	Exactly n occurrences of R

Other utilities for regular expressions are:

Syntax	Matches
(R)	Override precedence / capturing group
^R	R at beginning of a line
R$	R at the end of a line
\t	Tab character (just like in C)
\n	Newline (just like in C)
\w	A word (same as [a-zA-Z0-9_])
\d	A digit (same as [0-9])
\s	Whitespace (same as [\t\r\n])
\W, \D, \S	Complement of \w, \d, \s respectively

6.2 Flex

A lexical analysis must recognize tokens in a stream of characters and possibly decorate tokens with additional info. Flex is a scanner generators based on regular expression description. A scanner is just a big finite state automaton. In a compiler, instead, the scanner prepares the input for the parser:

- Detects the tokens of the language.

- Cleans the input.

- Adds information to the tokens.

The input of the lexical analyzer generator called flex is a specification file of the scanner, while the output is a C source code file that implements the scanner. A flex file is structured in three sections separated by %%:

- Definitions: declare useful regular expressions. The definition associates a name to a set of characters using regular expressions, and are usually employed to define simple concepts. They are recalled by putting their name in curly braces

- Rules: bind regular expressions combinations to actions. A rule represents a full token to be recognized, and it is defined with a regular expression. They define a semantic action to be made at each match. The semantic actions are executed every time the rule is matched, and can access matched textual data. Simple applications put the business logic directly inside semantic actions. More complex applications that also use a separate parser instead assign a value to the recognized token, and return the token type.

- User code: C code (generally helper functions). This code is copied to the generated scanner as is. It usually contains the main function and any other routine called by actions.

The scanner generated by flex is called "lex.yy.c". The yylex() function parses the file yyin until a semantic action returns or the file ends (return value 0).

Flex requires you to implement a single function "int yywrap(void)" that is called when the file ends. It gives the opportunity to open another file and continue scanning from there. It must return 0 if the parsing should continue or 1 if the parsing should stop. If you don't want this, you must put the following line in the scanner source: "%option noyywrap"

Some last important rules to remember:

- Longest matching rule: if more than one matching string is found, the rule that generates the longest one is selected.

- First rule: if more than one string with the same length is matched, the rule listed first will be triggered.

- Default action: if no rules are found, the next character in input is considered matched implicitly and printed to the output stream as is.

The generated parser implements a non-deterministic finite state automaton that tries to match all possible tokens at the same time, and as soon as one is recognized:

1. The semantic action is executed.

2. The stream skips past the end of the token.

3. The automaton reboots

Actually, the NFA is translated into a deterministic automaton using a modified version of the Berry-Sethi algorithm.

Multiple scanners

Sometimes is useful to have more than one scanner together. In order to support multiple scanners: rules should be marked with the name of the associated scanner (start condition), and we need to have special actions to switch between scanners. A start condition S: is used to mark rules with as a prefix $\langle S \rangle$ RULE, and it marks rules as active when the scanner is running the S scanner. Moreover:

- The * start condition matches every start condition.

- The initial start condition is INITIAL.

- Start conditions are stored as integers.

- The current start condition is stored in the YY_START variable.

Start conditions can be:

- Exclusive: declared with $\%x$ S; disables unmarked rules when the scanner is in the S start condition.

- Inclusive: declared with $\%s$ S; unmarked rules active when scanner is in the S start condition.

The INITIAL condition is inclusive. Other special actions are:

- BEGIN(S): place scanner in start condition S.

- ECHO: copies yytext to output.

6.3 Bison

What syntax is valid or not is defined by the grammar. A syntactic analysis must:

- Identify grammar structures.

- Verify syntactic correctness.

- Build a (possibly unique) derivation tree for the input.

Syntactic analysis does not determine the meaning of the input. That is the task of the semantic analysis. The syntactic analysis is performed over a stream of terminal symbols. Non-terminal symbols are only generated through reduction of grammar rules.

Bison is the standard tool to generate LR parsers, and it is designed to work seamlessly together with flex. It is a generated parser that uses LALR(1) methodology. The generated parser implements a table driven push-down automaton:

- The pilot automaton is described as finite state automaton.

- The parsing stack is used to keep the parser state at runtime.

- Acts as a typical shift-reduce parser.

File format

A bison file is structured in four sections:

- Prologue: useful place where to put header file inclusions, variable declarations.

- Definitions: definition of tokens, operator precedence, non-terminal types.

- Rules: grammar rules.

- User code: C code (generally helper functions), specified in BNF notation.

Different syntactic elements can be defined using %token. In the generated parser each token is assigned a number; in this way you can use them in the lexer.

Just like Flex, Bison allows to specify semantic actions in grammar rules. A semantic action is a conventional C code block and can be specified at the end of each rule alternative. Semantic actions are executed when the rule they are associated with has been completely recognized. The consequence is that the order of execution of the actions is bottom-up. You can also place semantic actions in the middle of a rule. Internally bison normalizes the grammar in order to have only end-of-rule actions, and this can introduce ambiguities

%union declaration specifies the entire collection of possible data types. Type specification for terminals (tokens) in the token declaration. Type specification for non-terminals in special %type declarations. The semantic value of each grammar symbol in a production is a variable called $i, where i is the position of the symbol. $$ corresponds to the semantic value of the rule itself.
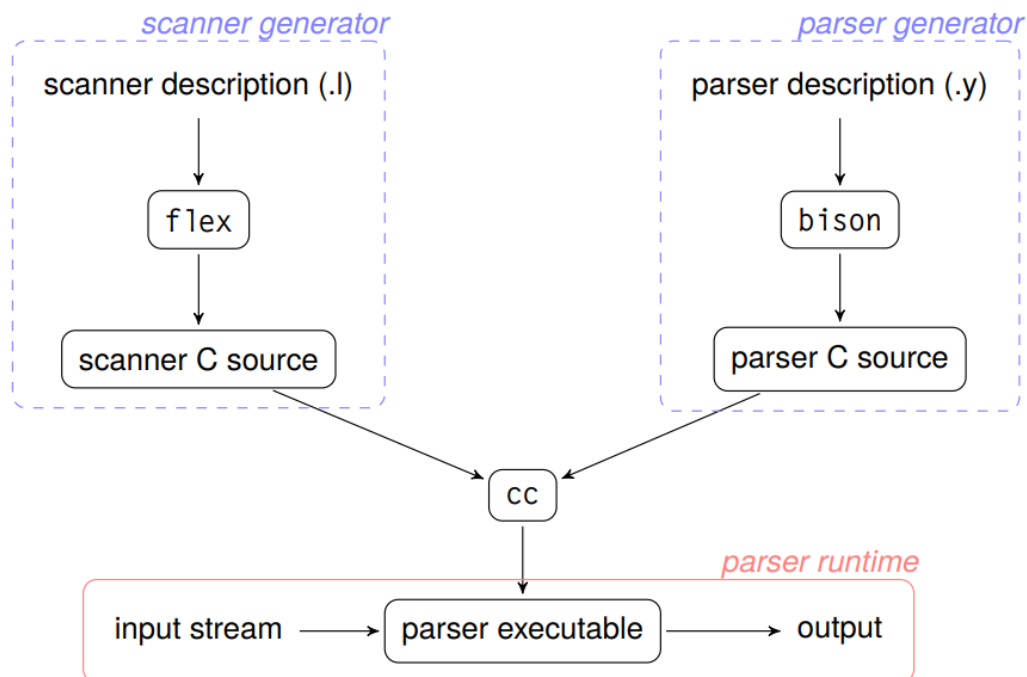
Interface and integration

Bison generates a parser that is a C file with suffix .tab.c and an header with declarations with suffix .tab.h. The main parsing function is "int yyparse(void)". For reading tokens the parser uses the same "yylex()" function that flex-generated scanners provide.

To integrate Flex and Bison we have to:

1. Include the *.tab.h header generated by Bison.

2. In the semantic actions: assign the semantic value of the token (if any) to the correct member of the yylval variable, and return the token identifiers declared in Bison.

3. Declare and implement the main() function.

4. Generate the flex scanner by invoking Flex.

5. Generate the Bison parser by invoking Bison.

6. Compile the C files produced by Bison and Flex together.

The first two points are for Flex and the third one is for Bison source code. Graphically we have the following schema.

6.4 ACSE