Formal Languages and Compilers

Lorenzo Rossi and everyone who kindly helped!

2022/2023

Last update: 2023-01-12

no alpaca has been harmed while writing these notes

Contents

About the notes

This document contains the notes for the *Formal Languages and Compilers* course, relative to the *2022/2023* class of *Computer Science and Engineering* held at *Politecnico di Milano.*

- Teacher: *Giovanni Agosta*
- Support teacher: *Gabriele Magnani*
- Textbook: *S. Crespi Reghizzi, L. Breveglieri, A. Morzenti, Formal Languages and Compilation, Springer Verlag, 3rd (2019) edition*

By comparing these notes with the material *(slides, video lessons)* provided during the lectures, you will find a lot of discrepancies in the order in which they are represented.
This might look like a bizarre *(if not completely stupid)* choice, but it has been deliberate as I have found the lessons quite confusing in their ordering and how each topic was introduced. You will still find each of the topics explained during the lessons, including something more *(sometimes).*

If you find any errors and you are willing to contribute and fix them, feel free to send me a pull request on the GitHub repository found at https://github.com/lorossi/formal-languages-and-compilers-notes.

A big thank you to everyone who helped me!

1 Introduction to Formal Languages

Definitions

- **Alphabet:** a **finite** set of symbols $\Sigma = \{a_1, a_2, \ldots a_k\}$
  - **cardinality** of an alphabet: the number of **distinct** symbols in it
  - **cardinality** of alphabet Σ: $k = |\Sigma|$
- **String:** a **finite**, ordered sequence of symbols *(possibly repeated)* from an Alphabet $\Sigma = a_1 a_2 \ldots a_n$
  - the strings of a language are also called its **sentences** or **phrases**
  - **length** of a string x: the number of symbols in it, written as $|x|$
  - **number of occurrences** of a symbol a in a string w: $|a|_w = n$ where $w = a_1 a_2 \ldots a_n$
  - two strings are **equal** if and only if they have the same length and the same symbols in the same order
  - **empty string:** the string with no symbols in it, denoted by ε
- **Substring:** a string y is a substring of a string x if $x = uyv$ for some u, v in Σ^*
  - y is a **proper** substring of x if $u \neq \varepsilon \lor v \neq \varepsilon$
  - u is a **prefix** of x
  - v is **suffix** of x
- **Language:** any set of strings defined over a given alphabet Σ
  - **cardinality** of a language: the number of different strings in it
  - **cardinality** of language L: $n = |L|$, $L = \{w_1, \ldots, w_n\}$
  - sometimes the Σ is both used to denote the set of all strings over the alphabet Σ and the language of all the strings of length 1

1.1 Operations

1.1.1 Operations on strings

- **Concatenation** or product of two strings x and y: $x \cdot y$ or xy for short
  - **if** $x = a_1 a_2 \ldots a_n$ **and** $y = b_1 b_2 \ldots b_m$, **then** $x \cdot y = a_1 a_2 \ldots a_n b_1 b_2 \ldots b_m$
  - **associative** property: $x(yz) = (xy)z$
  - **length** of the product: $|xy| = |x| + |y|$
  - **product** of the empty string and any string is the empty string: $\varepsilon \cdot x = \varepsilon$
- **Reflection** of a string x: x^R
  - if $x = a_1 a_2 \ldots a_n$ then $x^R = a_n a_{n-1} \ldots a_1$
  - **double reflection** of a string x is the same string: $\left(x^R\right)^R = x$
  - **distributive** property: $(xy)^R = x^R y^R$
- **Repetition** of a string x to the power of n, $(n > 0)$: concatenation of n copies of x
  - if $n = 0$ then $x^n = x^0 = \varepsilon$
  - if $n = 1$ then $x^n = x^1 = x$
  - elevating ε to any power gives ε: $\varepsilon^n = \varepsilon$
  - inductive **definition:**
  $$\begin{cases} x^n = x \cdot x^{n-1} & \text{if } n > 0 \\ x^0 = \varepsilon & \text{otherwise} \end{cases}$$
- **Operator precedence:** repetition and reflection have higher precedence than concatenation

1.1.2 Operations on languages

Operations are typically defined in languages by applying them to each string in the language.

- **Reflection** of a language L: L^R
  - formal **definition:** $L^R = \left\{ x \mid \exists y \left(y \in L \land x = y^R \right) \right\}$

- the same properties of the string reflection apply to the language reflection
- **Prefixes** of a language L: $P(L)$
  - formal **definition:** $P(L) = \{y \mid y \neq \varepsilon \wedge \exists x \, \exists z \, (x \in L \wedge x = yz \wedge z \neq \varepsilon)\}$
  - a language L is **prefix free** if $P(L) \cap L = \emptyset$ *(no words of L are prefixes of other words of L)*
- **Concatenation** of two languages L and M: $L \cdot M$ or LM for short
  - formal **definition:** $L \cdot M = \{x \cdot y \mid x \in L \wedge y \in M\}$
  - consequences: $\emptyset^0 = \{\varepsilon\} \quad L\emptyset = \emptyset L = \emptyset \quad L\{\varepsilon\} = \{\varepsilon\}L = L$
- **Repetition** of a language L to the power of n, $(n > 0)$: concatenation of n copies of L
  - **inductive** definition:
  $$\begin{cases} L^n = L \cdot L^{n-1} & \text{if } n > 0 \\ L^0 = \emptyset & \text{otherwise} \end{cases}$$
  - **finite languages:** if $L = \{\varepsilon, a_1, a_2, \ldots, a_k\}$, then L^n is finite as all its strings have length n
- **Quotient** of a language L by a language M: L/M
  - formal **definition:** $L/M = \{y \mid \exists x \in L \, \exists z \in M \, (x = yz)\}$
  - if no string in a language M ha a string in L as a suffix, then $L/M = L, M/L = \emptyset$

Set operations   The customary operations on sets can be applied to languages as well:

- **Union:** $L \cup M = \{x \mid x \in L \vee x \in M\}$
- **Intersection:** $L \cap M = \{x \mid x \in L \wedge x \in M\}$
- **Difference:** $L \setminus M = \{x \mid x \in L \wedge x \notin M\}$
- **Inclusion:** $L \subseteq M \Leftrightarrow L \setminus M = \emptyset$
- **Strict inclusion:** $L \subset M \Leftrightarrow L \subseteq M \wedge L \neq M$
- **Equality:** $L = M \Leftrightarrow L \subseteq M \wedge M \subseteq L$

Consequences

- **Universal** language: the set of all strings over the Alphabet Σ, including ε
  - formal **definition:** $L_{\text{universal}} = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$
- **Complement** of a language L over an alphabet Σ: the set difference of the universal language and L
  - formal **definition:** $\overline{L} = L^C = L_{\text{universal}} \setminus L$
  - the universal language is **not empty:** $L_{\text{universal}} \neq \emptyset$
  - the **complement** of a **finite language** is always **infinite**
  - the **complement** of an **infinite language** is **not necessarily finite**

1.1.3   Algebraic operations on languages

> **Definition 1.1** (Reflexive and transitive closure R^* of Relation R)**.** Given a set A and a relation $R \subseteq A \times A$, $(a_1, a_2) \in \mathbb{R}$, the application of R to a_1, $a_2 \in R$ is denoted as $a_1 R a_2$ Then R^* is a relation defined by:
>
> - $x R^* x \quad \forall x \in A$, **reflexivity** property
> - $x_1 R x_2 \wedge x_2 R x_3 \wedge \ldots \wedge x_{n-1} R x_n \implies x_1 R^* x_n \quad \forall x_1, x_2, \ldots, x_n \in A$, **transitivity** property
>
> If aRb is a step in relation R^*, then aR^*b is a **chain** of $n \geq 0$ steps.

> **Definition 1.2** (Transitive closure R^+ of a relation R)**.** Given a set A and a relation $R \subseteq A \times A$, $(a_1, a_2) \in \mathbb{R}$ is also denoted as $a_1 R a_2$. Then R^+ is a relation defined by:
>
> - $x_1 R x_2 \wedge x_2 R x_3 \wedge \ldots \wedge x_{n-1} R x_n \implies x_1 R^+ x_n \quad \forall x_1, x_2, \ldots, x_n \in A$, **transitivity** property
>
> If aRb is a step in relation R^+, then aR^+b is a **chain** of $n \geq 1$ steps.

1.1.4 Star operator - Kleene star

The **star operator** is the reflexive transitive closure under the concatenation operation; It's also called the **Kleene star**.

> **Definition 1.3.** Let L be a language over an alphabet Σ. Then the **star operator** is defined as:
> $$L^* = \bigcup_{h=0}^{\infty} L^h = L^0 \bigcup L^1 \bigcup L^2 \bigcup \ldots = \varepsilon \bigcup L^1 \bigcup L^2 \bigcup \ldots$$

Properties

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

Consequences

- It represents the union of all the powers of the language L
- Every string of the star language can be chopped into substrings that are in the original language L
- The star language L^* can be equal to the base language L
- If Σ is the base language, then Σ^* is the universal language
- The language L is defined on alphabet Σ
- If L^* is finite then: $\emptyset^* = \{\varepsilon\}^* = \{\varepsilon\}$

1.1.5 Cross operator

The **cross operator** is the transitive closure under the concatenation operation. The union does not include the first power L^0.

> **Definition 1.4** (Cross operator). Let L be a language. Then the cross operator is defined as:
> $$L^+ = \bigcup_{h=1}^{\infty} L^h = L^1 \bigcup L^2 \bigcup \ldots$$

Consequences

- It can be derived from the star operator: $L^+ = L \cdot L^*$
- If $\varepsilon \in L$ then $L^+ = L^*$

2   Regular Expressions and Regular Languages

The family of **regular languages** *(also called REG or type 3)* is the simplest language family.
It can be defined in many ways, but this course will focus on the following 3:

1. **Algebraically** *(Section 2.1)*
2. Via generative **grammars** *(Section 3, Page 8)*
3. Via recognizer **automata** *(Section 4.1, Page 24)*

2.1   Algebraic definition

A language over an alphabet $\Sigma = \{a_1, a_2, \ldots, a_n\}$ is **regular** if it can be expressed by applying for a finite number of times the operations of *concatenation* ($\cdot$), *union* ($\cup$), and *star* ($*$) starting by unitary languages $\{a_1\}, \{a_2\} \ldots \{a_n\}$ or the empty string ε.
More precisely, a regular expression is a string r containing the terminal characters of Σ and the aforementioned operators, under the following **rules**:

1. $r = \varepsilon$, **empty** or **null** string
2. $r = a$, **unitary** language
3. $r = s \cup t$, **union** of two regular expressions
4. $r = s \cdot t$, **concatenation** of two regular expressions
5. $r = (s)^*$, **Kleene star** of a regular expression

where the symbols s and t are regular expressions themselves.

The correspondence between a regular expression and its denoted language is so direct that it's possible to define the language L_e via the *r.e. e* itself.

> **Definition 2.1** (Regular language)**.** A language is **regular** if it is denoted by a **regular expression**; the empty language $\{\varepsilon\}$ *(or $\emptyset$)* is considered a regular language as well, despite not being denoted by any regular expression.
> The collection of all regular languages is called the family *REG* of regular languages.

> **Definition 2.2** (Finite languages family)**.** The collection of all finite languages (all the languages with a finite cardinality) is called *FIN*.

Since every finite language is regular, it can be proven that

$$FIN \subseteq REG$$

as every finite language is the union of a finite number of strings $x_1, x_2, \ldots, x_n$, where each x_i is a regular expression *(a concatenation of a finite number of alphabet symbols)*:

$$(x_1 \cup x_2 \cup \ldots \cup x_k) = (a_{11}a_{12} \ldots a_{1n} \cup \ldots \cup a_{k1}a_{k2} \ldots a_{km})$$

Since family *REG* includes non-finite languages too, the inclusion is **proper:**

$$FIN \subset REG$$

2.1.1   Derivation of a Language from a Regular Expression

In order to **derivate** a language from a regular expression, it's necessary to follow the rules of the regular expression itself. This may lead to multiple choices, as star and crosses operators offer multiple possibilities; by making a choice, a new *r.e.*, defining a less general language *(albeit contained in the original one)* is obtained.
A regular expression is a **choice** of another by if:

1. The *r.e. e_k (with $1 \leq k \leq m$, $m > 2$)* is a **choice of the union** $e_1 \cup e_2 \cup \ldots \cup e_m$

2. The *r.e* e^m *(with $m > 1$)* is a **choice of the star** e^* or cross e^+
3. The empty string ε is a **choice of the star** e^*

Given a *r.e. e*, it's possible to derive another *r.e. e′* by making a choice: **replacing** any *"outermost" (or "top level")* sub expression with another that is a choice of it.

2.1.1.1 Derivation relation

An *r.e. e* derives another *r.e. e′* (*denoted as $e \Rightarrow e'$*) if the two *r.e.* can be factorized as:

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

where δ is a **choice** of β. Such a derivation $\Rightarrow$ is called **immediate** as it makes only a choice *(or one step)*. The derivation relation can be applied repeatedly, yielding:

- $e \overset{n}{\Rightarrow} e_n$ **if** $e \Rightarrow e_1 \Rightarrow \ldots \Rightarrow e_n$ in n steps
- $e \overset{*}{\Rightarrow} e_n$ **if** e derives e_n in $n \geq 0$ steps
- $e \overset{+}{\Rightarrow} e_n$ **if** e derives e_n in $n \geq 1$ steps

Immediate derivations

- $a^* \cup b^+ \Rightarrow a^*$
- $a^* \cup b^+ \Rightarrow b^+$
- $(a^* \cup bb)^* \Rightarrow (a^* \cup bb)\,(a^* \cup bb) = (a^* \cup bb)^2$

Some expressions produced by derivation from an expression r contain the meta symbols of union, star, and cross, while other just contain terminal characters, empty strings, or redundant parentheses; the latter expressions compose the **language denoted by the *r.e.***.

Definition 2.3 (Langauge defined by a Regular Expression). The **language defined by a regular expression** is composed by terminal characters, empty strings, and redundant parentheses. They are defined by the relation:
$$L_r = \left\{ x \in \Sigma^* \,\middle|\, r \overset{*}{\Rightarrow} x \right\}$$

Definition 2.4 (Equivalent Regular Expressions). Two *r.e.* are **equivalent** if they define the same language. A phrase of a regular language can be obtained through different choices used in the derivation.

2.1.1.2 Ambiguity of Regular Expressions

A sentence and the *r.e.* that derives it is said to be **ambiguous** if and only if it can be obtained by structurally different derivations. Derivations are structurally different if they differ not only in the order of the choices but also in the choices themselves.

Definition 2.5 (Numbered Regular Expression). A **Numbered Regular Expression** *(numbered r.e.)* is a *r.e.* where each choice is numbered. Numbered *r.e.* are used to distinguish between structurally different derivations, thus determining the ambiguity of a *r.e.* itself.

Sufficient conditions for ambiguity A *r.e. e* is ambiguous if the language of the numbered version e' includes two distinct strings x and y that coincide when the numbers are removed.

2.1.2 Extended regular expressions

In order to use regular expressions in practice, it is convenient to add the basic operators of union, concatenation, and star and the derived operators of power and cross to the already defined set of operators.
Moreover, the following operators are added for convenience:

- **Repetition** from $k \geq 0$ to $n > k$ times $[a]_k^n = a^k \cup a^{k+1} \cup \ldots \cup a^n$
- **Option** $[a]_0^1 = a^0 \cup a^1 = \varepsilon \cup a$
- **Interval** of an ordered set, for instance, the interval of the set of integers from 0 to 9 is $(0 \ldots 9)$

Sometimes, set operations of intersection, set difference and complement are included in the definition.

It can be proven *(via finite automata)* that the use of these operators does not change the expressive power of a regular expression, but they provide some convenience.

2.2 Closure property of *REG*

Let `op` be an operator to be applied to one or two languages in order to obtain another language. A language family is **closed under operator op** if the product of `op` applied to two languages in the family is also in the family.
In other words, let *FAM* be a language family and A, B two languages such that $A, B \in FAM$. Then both languages are closed under the operator `op` if and only if $C = A \operatorname{op} B \in FAM$.

> **Definition 2.6** (Closure property of the *REG* family)**.** The family *REG* is **closed** under the operators of **concatenation** ·, **union** $\cup$, **complement** $\neg$, and **star** $*$; therefore it is closed under any derived operator, such as **power** ^{n} and **cross** $^+$.
> As a direct consequence, it's **not closed** under the operators of set **difference** $\setminus$ and **intersection** $\cap$.

2.3 Limits of Regular Expressions

Simple languages such as $L = \{\texttt{begin}^n \ \texttt{end}^n \mid n > 0\}$ representing basic syntactic structures such as:

```
begin
   ...
   begin
     ...
      begin
       ...
      end
     ...
   end
...
end
```

are not regular *(and cannot be represented by a regular expression)* because:

- the **number** of `begin` and `end` is not **guaranteed to be the same**
- the **nesting** of `begin` and `end` is not **guaranteed to be balanced**

In order to represent this *(and other)* languages, a new formal model needs to be used: the **generative grammars**.

3 Generative grammars

A **generative grammar** *(or syntax)* is a set of simple rules that can be repeatedly applied in order to generate all and only the valid strings. A generative grammar defines languages via:

- Rule **rewriting**
- **Repeated application** of the rules

3.1 Context-Free Grammars

The definition of Context-Free Grammar is shown in *3.1*.

> **Definition 3.1** (Context-Free Grammar)**.** A **context-free** *(also called CF, type 2, BNF, or free)* grammar G is defined as a 4-tuple $\langle V, \Sigma, P, S \rangle$ where:
>
> - V is the set of **non-terminal symbols**, called **non-terminal alphabet**
> - Σ is the set of **terminal symbols**, called **terminal alphabet**
> - P is the set of **rules** or **productions**
> - $S \in V$ is a **specific non-terminal symbol**, called **axiom**
>
> All rules are in form of $X \to \alpha$, where $X \in V$ and $\alpha \in (V \cup \Sigma)^*$. The **left** and **right** parts of a rule are respectively called **LP** and **RP** for brevity.

Two or more rules with the same left part, such as

$$X \to \alpha_1 \quad X \to \alpha_2 \quad \cdots \quad X \to \alpha_n$$

can be grouped into a single rule, denoted as:

$$X \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n \quad \text{or} \quad X \to \alpha_1 \cup \alpha_2 \cup \cdots \cup \alpha_n$$

Where the strings $\alpha_1, \alpha_2, \ldots, \alpha_n$ are called **alternatives** of X.

In order to avoid confusions, metasymbols $\to$, $\mid$, $\cup$ and ε **should not be used** for terminal or non-terminal symbols; moreover, the terminal and non-terminal alphabets should be **disjoint** *($\Sigma \cap V = \emptyset$)*. The meta symbol $\to$ is used to separate the left part from the right part of a rule and it's different from the $\Rightarrow$ symbol, used to represent the derivation relation *(or rule rewriting)*.

The **axiom** S is used to start the derivation process and it's the only non-terminal symbol that can be used as the left part (LP) of a rule.

Normally, these conventions are adopted:

- **Terminal** characters are written as **latin lowercase letters** $\{a, b, c, \ldots, z\}$
- **Nonterminal** characters are written as **latin uppercase letters** $\{A, B, C, \ldots, Z\}$
- **Strings** containing only **terminal characters** are written as **latin lowercase letters** $\{a, b, c, \ldots, z\}$
- **Strings** containing only **non-terminal characters** are written as σ *(in general, greek lowercase letters towards the end of the alphabet)*
- **Strings** containing both **terminal** and **non-terminal characters** are written as **greek lowercase letters** $\{\alpha, \beta, \gamma, \ldots, \omega\}$

3.1.1 Rule types

Rules can be classified depending on their form, in order to make the study more immediate; such classification is shown in Table 1.

Additionally, a **rule** that is both *left recursive* and *right recursive* is called ***left-right-recursive*** or *two-side-recur*. The terminal in the RP of a rule in *operator form* is called *operator*.

class	description	model
terminal	either RP contains only terminals or it's the empty string	$\to u \mid \varepsilon$
empty or Null	RP is the empty string	$\to \varepsilon$
initial or *axiomatic*	LP is the grammar axiom S	$S \to \alpha$
recursive	LP occurs in RP	$A \to \alpha A \beta$
left recursive	LP is the prefix of RP	$A \to A\beta$
right recursive	LP is the suffix of RP	$A \to \beta A$
copy	RP consists of one non-terminal	$A \to B$
identity	LP and RP are the same	$A \to A$
linear	RP contains at most one non-terminal	$\to uBv \mid v$
left-linear or *type 3*	RP contains at most one non-terminal as the *prefix*	$\to Bv \mid w$
right-linear or *type 3*	RP contains at most one non-terminal as the *suffix*	$\to uB \mid w$
homogeneous normal	RP consists either of $n \geq 2$ non-terminals or 1 terminal	$\to A_1 \ldots A_n \mid a$
Chomsky normal	RP consists of 2 non-terminals or 1 terminal	$\to BC \mid a$
Greibach normal	RP consists of 1 terminal possibly followed by non-terminal	$\to a\sigma \mid b$
operator form	RP consists of 2 non-terminals separated by a terminal	$\to AaB$

Table 1: Rule types

3.1.2 Derivation and language generation

Firstly, the notion of **string derivation** has to be formalized. Let $\beta = \delta A \eta$ be a string containing a non-terminal symbol A and two strings δ and η. Let $A \to \alpha$ be a rule of grammar G and let $\gamma \alpha \eta$ the string obtained by replacing the non-terminal symbol A in β by applying the rule.

The relation between the two strings is called **derivation**. The string β derives the string γ for grammar G and it's denoted by the symbol:

$$\beta \underset{G}{\Longrightarrow} \gamma$$

Rule $A \to \alpha$ is applied in such a derivation and string α **reduces** to non-terminal A.

Now consider a chain of derivation, with $n \geq 0$ steps

$$\beta_0 \Rightarrow \beta_1 \Rightarrow \cdots \Rightarrow \beta_n$$

which can be shortened to

$$\beta_0 \overset{n}{\Rightarrow} \beta_n$$

where β_0 is the initial string and β_n is the final string.

If $n = 0$, every string derives itself *(as $\beta \Rightarrow \beta$)* and the relation is called **reflexive**.

To express derivations of any length, the symbols

$$\beta_0 \overset{*}{\Rightarrow} \beta_n,\ n \geq 0 \quad \text{or} \quad \beta_0 \overset{+}{\Rightarrow} \beta_n,\ n \geq 1$$

are used. In general, the **language generated** or defined by a grammar G **starting from non-terminal A** is the set of terminal strings that derive from non-terminal A in one or more steps:

$$L_A(G) = \left\{ x \in \Sigma^* \,\middle|\, A \overset{+}{\Rightarrow} x \right\}$$

If the non-terminal is the axiom S, then the **language generated** by G is:

$$L_S(G) = L(G) = \left\{ x \in \Sigma^* \,\middle|\, S \overset{+}{\Rightarrow} x \right\}$$

Additional forms

- If $A \overset{*}{\Rightarrow} \alpha$, $\alpha \in (V \cup \Sigma)$, then α is called **string form** generated by G

- If $S \overset{*}{\Rightarrow} \alpha$, $\alpha \in (V \cup \Sigma)$, then α is called **sentential from** or **phrase form** generated by G

- If $A \overset{*}{\Rightarrow} s$, $s \in \Sigma^*$ then s is called **phrase** or **sentence** generated by G

> **Property 3.1** (Grammars equivalence)**.** Two grammars G and G' are **equivalent** if they generate the same language *(i.e. $L(G) = L(G')$).*

3.1.3 Erroneous Grammars and useless rules

The Definition of **clean grammar** is shown in *(Definition 3.2)*.

> **Definition 3.2.** A grammar G is called **clean** if both the following conditions are satisfied:
>
> 1. every **non-terminal** A is **reachable** from the axiom S, as it exists a derivation $S \overset{+}{\Rightarrow} \alpha A \beta$
> 2. every **non-terminal** A is **well defined**, as it generates a non-empty language $L_G(A) \neq \emptyset$
>   - $\rightarrow$ this rule includes also the case when no derivation from A terminates with a terminal string

It's quite straightforward to check whether a grammar is clean; the following algorithm describes how to do it.

3.1.3.1 Grammar Cleaning Algorithm

The **Grammar Cleaning Algorithm** is based on the following two steps:

step 1. compute the *set* def $\subseteq V$ of the well defined **non-terminals**

- def is initialized with the **non-terminals** that occur in the **terminal rules** *(the rules having a terminal as their RP)*
$$\text{def} := \{A \mid (A \rightarrow u) \in P, \ u \in \Sigma^*\}$$

- this transformation is applied **repeatedly** until convergence is reached
$$\text{def} := \text{def} \cup \{B \mid (B \rightarrow D_1 \ldots D_n) \in P \land \forall i \, D_i \in (\Sigma \cup \text{def})\}$$

    - each symbol $D_i, 1 \leq i \leq n$ is either a **terminal in** Σ or a **non-terminal in def** already
    - at each iteration, two possible outcomes are possible:
        - (a) a new non-terminal is found that occurs as *LP* of a rule having as *RP* a string of terminals or well-defined non-terminals
        - (b) the termination condition is reached, as no new non-terminal is found

step 2. **compute** a *directed graph* **between non-terminals** using the **produce** relation $A \xrightarrow{produce} B$

- this relation indicates that a non-terminal A **produces** a non-terminal B if and only if there exists a rule $A \rightarrow \alpha B \beta$, with α, β strings
- a non-terminal C is **reachable** from the axiom S if and only if in the graph there exists a path directed from S to C
- non-terminal that are **not reachable** are **eliminated**

Often another requirement is added for the cleanliness of a grammar: it must not allow **circular derivations** $A \overset{+}{\Rightarrow} A$, as they are not essential and produce **ambiguity** *(discussed in Section 3.4)*.

Such derivations are not essential because if a string x is generated via a circular derivation such as

$$A \Rightarrow A \Rightarrow A \Rightarrow x$$

it can also be obtained by a non-circular derivation

$$A \Rightarrow x$$

3.2 Recursion and Language Infinity

An essential property of technical languages is to be infinite, as a language that can only produce a limited set of strings is not useful. In order to generate an infinite number of strings via a finite set of rules, the grammar has to derive strings of unbounded length: this feature needs **recursion** in the grammar rules, and it's formally introduced in Definition 3.3.

> **Definition 3.3** (Recursion)**.** An n-step derivation of the form $A \overset{n}{\Rightarrow} xAy$, $n \geq 1$ is called **recursive** or **immediately recursive** if $n = 1$, while the non-terminal A is called **recursive**.
> Similarly, if the strings x or y are empty *($x = \varepsilon$ or $y = \varepsilon$)*, the recursion is called respectively **left-recursive** and **right-recursive**.

A formal definition of language infinity is shown in 3.4.

> **Definition 3.4** (Language Infinity)**.** Let grammar G be clean and devoid of circular derivations. Then language $L(G)$ is infinite if and only if grammar G has a recursive derivation.*($x = \varepsilon$ or $y = \varepsilon$)*

The definitions introduce a sufficient and necessary condition for the language to be infinite:

- **Necessary condition:** if no recursive derivation is possible, every derivation has a limited length and the language is finite
- **Sufficient condition:** if the language has a rule $A \overset{n}{\Rightarrow} xAy$, then it holds $A \overset{+}{\Rightarrow} x^m A y^m$ for any $m \geq 1$ with $x, y \in \Sigma+$ (not empty because grammar is not circular)
  - **cleanliness** condition of G **implies** $S \overset{*}{\Rightarrow} uAv$ (as A is reachable from S)
  - a **successful** derivation of A **implies** $A \overset{+}{\Rightarrow} w$
  - therefore there exists non-terminals that generate an **infinite language**:

$$S \overset{*}{\Rightarrow} uAv \overset{+}{\Rightarrow} ux^m A y^m v \overset{+}{\Rightarrow} ux^m w y^m v \quad \forall\, m \geq 1$$

In other words:

- a grammar **does not have recursive derivations** $\iff$ the graph of the produce relation **has no circuits**
- a grammar **has recursions** $\iff$ the graph of the produce relation **has circuits**

3.3 Syntax Trees and Canonical Derivations

A **syntax tree** *(shown in Figure 1a)* is an oriented, sorted graph with no cycles, such that for each pair of nodes A and B there is at most one edge from A to B.

Properties

- it represents **graphically** the derivation process
- the **degree** of a node is the number of its children
- the **root** of the tree is the axiom S
- the **frontier** of the tree *(the leaves ordered from left to right)* contains the generated phrase
- the **subtree** with root N is the tree having N as its root, including all its descendants

Furthermore, two subtypes of syntax trees exist:

- **skeleton tree** *(Figure 1b)*, where only the frontier and the structure are shown
- **condensed skeleton tree** *(Figure 1c)*, where internal nodes on a non branching paths are merged; only the frontier and the structure are shown
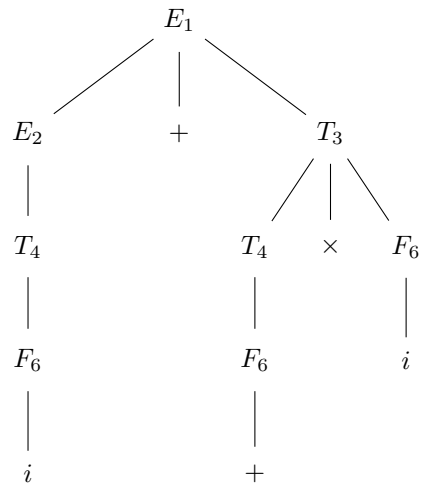
(a) Syntax tree
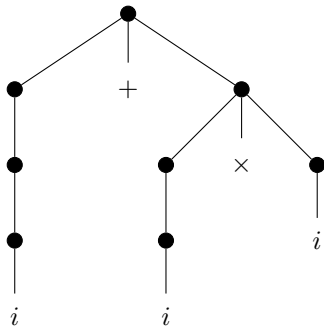
(b) Skeleton tree
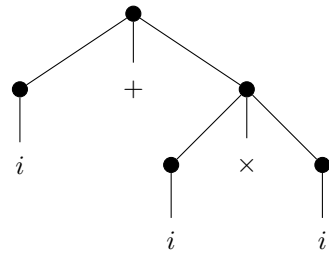
(c) Condensed skeleton tree

Figure 1: Types of syntax trees

3.3.1 Left and Right Derivation

The definition of **Left and Right Derivation** is shown in Definition 3.5.

> **Definition 3.5** (Left and Right Derivation). A derivation of $p \geq 1$ steps $\beta_0 \Rightarrow \beta_1 \Rightarrow \ldots \Rightarrow \beta_p$ where
>
> $$\beta_i = \delta_i A_i \eta_i, \ \beta_{i+1} = \delta_i \alpha_i \eta_i \quad 0 \leq i \leq p - 1$$
>
> it's called **left** *(leftmost)* **derivation or right** *(rightmost)* **derivation** if it holds $\delta_i \in \Sigma^*$ or $\eta_i \in \Sigma^*$, respectively, for every $0 \leq i \leq p - 1$ *(every left or right part of the RP is composed only by terminals)*.

In other words, at each step, a left derivation *(or a right one)* expands the rightmost *(or leftmost)* non-terminal. A letter l or r may be subscripted to the arrow sign $(\Rightarrow)$, to explicitly indicate the direction of the derivation. Other derivations that are neither left nor right exist, either because the non-terminal expanded is not always leftmost or rightmost, or because the expansion is sometimes leftmost and sometimes rightmost.

Every sentence of a context-free grammar can be generated by a left derivation and a right one; this property does not hold for other grammars *(such as context-sensitive grammars)*. Therefore, each rule of a language in the *CF* family can be transformed in either left or right derivations.

3.3.2 Regular composition of free languages

If the basic operations of regular languages *(union, concatenation, star and cross)* are applied to context-free languages, the result is still a member of the *CF* family.

Let $G_1 = (\Sigma_1, V_1, P_1, S_1)$ and $G_2 = (\Sigma_2, V_2, P_2, S_2)$ be two context-free grammars, defining respectively the languages L_1 and L_2. Let's firstly assume that their non-terminal sets are disjoint, so that $V_1 \cap V_2 = \emptyset$ and that symbol S, the axiom that is going to be used to build the new grammars, is not used by either G_1 or G_2 $(S \notin (V_1 \cup V_2))$.

- **Union:** the grammar G of language $L_1 \cup L_2$ contains all the rules of G_1 and G_2, plus the initial rules $S \Rightarrow S_1 \,|\, S_2$. In formula:

$$G = (\Sigma_1 \cup \Sigma_2, \ V_1 \cup V_2 \cup \{S\}, \ P_1 \cup P_2 \cup \{S \to S_1 \mid S_2\}, \ S)$$

- **Concatenation:** the grammar G of language $L_1 \cdot L_2$ contains all the rules of G_1 and G_2, plus the initial rules $S \Rightarrow S_1 S_2$. In formula:

$$G = (\Sigma_1 \cup \Sigma_2, \ V_1 \cup V_2 \cup \{S\}, \ P_1 \cup P_2 \cup \{S \to S_1 S_2\}, \ S)$$

- The grammar G of language L_1^* contains all the rules of G_1, plus the initial rules $S \Rightarrow S_1 \,|\, \varepsilon$
- The grammar G of language $L_1 L_2$ contains all the rules of G_1 and G_2, plus the initial rules $S \Rightarrow S_1 S_2$, thanks to to the identity $L^+ = L \cdot L^*$

Finally, the family *CG* of context-free languages is **closed** under the operations of **union**, **concatenation**, **star** and **cross**. Moreover, the mirror language of $L(G)$, $L(G)^R$, can be generated by a grammar G^R that is obtained from G by reversing the *RP* of the rules; as such, the family of *CG* languages is also **closed under the operation of mirror**.

3.4 Grammar ambiguity

In natural language, the common linguistic phenomenon of ambiguity shows up when a sentence has two or more meanings. Ambiguity can be:

- **semantic**, whenever a phrase contains a word that has two or more meanings
- **syntactic**, *(or structural)* whenever a phrase has a different meaning depending on the structure assigned

Likewise, a sentence x of a grammar G is syntactically ambiguous if it is generated by two or more syntax trees; the grammar G is called ambiguous too.

Definitions 3.6 and 3.7 introduce the degree of ambiguity of a sentence and a grammar, respectively.

> **Definition 3.6** (Degree of ambiguity of a sentence)**.** The **degree of ambiguity** of a **sentence** x of a language $L(G)$ is the number of **distinct trees** of x compatible with G.
> This value is unbounded.

> **Definition 3.7** (Degree of ambiguity of a grammar)**.** The **degree of ambiguity** of a **grammar** G is the maximum degree of ambiguity of its sentences $L(G)$.

Determining if a grammar is ambiguous is an important problem. Sadly, it's an undecidable characteristic: there is no general algorithm that, given any free grammar, terminates *(in a finite number of steps)* with the correct answer. However, the absence of ambiguity in a specific grammar can be shown on a case-by-case basis, using inductive reasoning on a finite number of cases.

The best approach to prevent the problem is to act in the design phase, by avoiding the ambiguous forms *(explained in the following Section).*

3.4.1 Catalog of Ambiguous Forms and remedies

In the following Paragraphs *(3.4.1.1 to 3.4.1.5)*, a common source of ambiguities and their respective solutions will be illustrated.

3.4.1.1 Ambiguity from Bilateral Recursion

A non-terminal symbol A is bilaterally recursive if it is both **left and right recursive**, for example, $A \xRightarrow{+} A\gamma$ and $A \xRightarrow{+} \beta A$. The cases where the two derivations are produced by the same rule or by different rules have to be treated separately:

- Bilateral recursion **from the same rule:** $E \to E + E \mid i$
  - this rule generates a regular language $L(G) = i(+i)^*$
  - non ambiguous right recursive grammar: $E \to i + E \mid i$
  - non ambiguous left recursive grammar: $E \to E + i \mid i$
- Bilateral recursion **from different rules:** $A \to aA \mid Ab \mid c$
  - this rule generates a regular language $L(G) = a^* c b^*$
  - solution 1: the two lists are generated by distinct rules $\begin{cases} S \to AcB \\ A \to aA \mid \varepsilon \\ B \to bB \mid \varepsilon \end{cases}$
  - solution 2: the rules force order in the generation $\begin{cases} S \to aS \mid X \\ X \to Xb \mid c \end{cases}$

3.4.1.2 Ambiguity from Language Union

If two languages $L_1 = L(G_1)$ and $L_2 = L(G_2)$ share some sentence, as **their intersection is not empty** $(L_1 \cap L_2 \neq \emptyset)$, the grammar $G = G_1 \cup G_2$ is ambiguous. The sentence $x \in L_1 \cap L_2$ is generated via two different trees, using respectively the rules of G_1 or G_2. On the contrary, sentences $y \in L_1 \setminus L_2$ and $z \in L_2 \setminus L_1$ are non ambiguous.

In order to fix this ambiguity, a disjoint set of rules for $L_1 \cap L_2$, $L_1 \setminus L_2$, and $L_2 \setminus L_1$ must be provided. There is no general method to achieve this goal, so it has to be done on a case-by-case basis.

3.4.1.3 Inherent ambiguity

A language is inherently ambiguous if it is not possible to define a grammar that generates it without ambiguity. In other words, a language $L(G)$ over a grammar G is inherently ambiguous if it is not possible to define a grammar G' that generates $L(G)$ without ambiguity.

This is the rarest case of ambiguity and it can be avoided in technical languages.

3.4.1.4 Ambiguity from Concatenation of Languages

Concatenating two *(or more)* non ambiguous languages $L_1 = L(G_1)$ and $L_2 = L(G_2)$ can generate ambiguity **if a suffix of L_1 is a prefix or a sentence of L_2**. The concatenation grammar of $L_1 L_2$:A

$$G = (\Sigma_1 \cup \Sigma_2, \{S\} \cup V_{N_1} \cup V_{N_2}, \{S \to S_1 S_2\} \cup P_1 \cup_2, S)$$

contains the axiomatic rule $S \to S_1 S_2$ in addition to the rules of G_1 and G_2.
Ambiguity arises if the following sentences exist in the languages:

$$u' \in L_1 \quad u'v \in L_2 \quad vz'' \in L_2 \quad z'' \in L_2 \quad v \neq \varepsilon$$

then the string $u'vz''$ is generated by two different derivations:

$$S \Rightarrow S_1 S_2 \overset{+}{\Longrightarrow} u'S_2 \overset{+}{\Longrightarrow} u'vz''$$

$$S \Rightarrow S_1 S_2 \overset{+}{\Longrightarrow} u'vS_2 \overset{+}{\Longrightarrow} u'vz''$$

To remove such ambiguity, the operation of moving a string from the suffix of L_1 to the prefix of L_2 *(and vice versa)* should be prevented. A simple solution is to introduce a new terminal as a separator between the two languages, for example, $\#$, such that the concatenation $L_1 \# L_2$ is easily defined without ambiguity by a grammar with the initial rule $S \to S_1 \# S_2$.

3.4.1.5 Other causes of ambiguity

Other causes of ambiguity are:

- Ambiguous regular expression
  - $\to$ **solution:** remove redundant productions from the rules
- Lack of order in derivations
  - $\to$ **solution:** introduce a new rule that forces the order

3.5 Strong and Weak equivalence

It's not enough for a grammar to generate correct sentences as it should also assign a suitable meaning to them; this property is called **structural adequacy**. Thanks to this definition, the equivalence between grammars can be refined in two different ways: **weak equivalence** and **strong equivalence**.
The next two Definitions *(3.8 and 3.9)* will introduce the two equivalence relations and will show how they can be used to prove the structural adequacy of a grammar.

Definition 3.8 (Weak equivalence). Two grammars G and G' are **weakly equivalent** if they generate the same language:
$$L(G) = L(G')$$

This relation is called **weak equivalence** does not guarantee that one grammar can be substituted with the other one *(for example in technical languages processors such as compilers):* the two grammars G and G' are not guaranteed to assign the same meaningful structure to every sentence.

Definition 3.9 (Strong equivalence). Two grammars G and G' are **strongly *(or structurally)* equivalent** if the following 2 conditions are satisfied:

A. $L(G) = L(G')$, weak equivalence
B. G and G' assign to each sentence two structurally similar syntax trees

---

The condition (B) has to be formulated per the intended application; a plausible formulation is: *two syntax trees are structurally similar if the corresponding condensed skeleton trees are equal.*

Strong equivalence implies weak equivalence *(due to Condition (A))*, but not vice versa; however the former is a decidable problem, while the latter is not. As a consequence, it may happen that grammars G and G' are not strongly equivalent without being able to determine if they are weakly equivalent.

The notion of **structural equivalence** can be generalized by requiring that the two corresponding trees should be easily mapped into one another by some simple transformation. This idea can be realized in different ways; for example, one possibility is to have a bijective correspondence between the subtrees of one tree and the subtrees of the other.

3.6   Grammar Normal Forms and Transformation

While normal forms are not strictly necessary for the definition of a grammar, they are used to simplify formal statements and theorem works as they constrain the rules without reducing the family of languages that can be generated by them.

In applied works, however, normal formal grammars are usually not a good choice because they are larger and less readable; in order to simplify them, several transformations can be applied, They will be presented in the following Sections *(3.6.1 to 3.6.7)*

3.6.1   Nonterminal Expansion

A general purpose transformation preserving language is **non-terminal expansion**, which consists of replacing a non-terminal with its alternatives. It replaces rule $A \to \alpha B \gamma$ with rules:

$$A \to \alpha \beta_1 \gamma \mid \alpha \beta_2 \gamma \mid \ldots \mid \alpha \beta_n \gamma \quad n \geq 1$$

where

$$B \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

are all the alternatives of B.

The language is not modified, since the two-steps derivation $A \Rightarrow \alpha B \gamma \Rightarrow \alpha \beta_i \gamma$ becomes the immediate derivation $A \Rightarrow \alpha \beta_i \gamma$.

3.6.2   Axiom elimination from Right Parts

At no loss of generality, **every RP of a rule can exclude the axiom S**.

The axiom elimination from RP consists in introducing a new axiom S_0 and the rule $S_0 \to S$: all the rules will be devoid of the axiom from the RP as they will be strings $\in (\Sigma \cup (V \setminus \{S\}))$

3.6.3   Nullable Nonterminals and elimination of Empty rules

A non-terminal A is **nullable** if it can derive the empty string; i.e. there exists a derivation $A \xRightarrow{+} \varepsilon$.

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:

$$A \in \text{Null} \Rightarrow \begin{cases} (A \to \varepsilon) \in P \\ (A \to A_1 A_2 \ldots A_2) \in P & \text{with } A_i \in V \setminus \{A\} \\ \forall 1 \leq i \leq n & \text{with } A_i \in \text{Null} \end{cases}$$

Where:

  row 1. for each rule $\in P$ add as alternatives those obtained by deleting, in the RP, the nullable non-terminals
  row 2. remove all empty rules $A \to \varepsilon$, except for $A = S$
  row 3. clean the grammar and remove any circularity

3.6.4 Copy Rules and their elimination

A **copy** *(or **subcategorization**)* **rule** has the form $A \to B$, where $B \in V$ is a non-terminal symbol. Any such rule is equivalent to the relation $L_B(G) \subseteq L_A(G)$, which means that the syntax class B is a subcategory of *(is included in)* the syntax class A.

For example, related to programming languages, the rules

$$\texttt{iterative\_phrase} \to \texttt{while\_phrase} \mid \texttt{for\_phrase} \mid \texttt{repeat\_phrase}$$

introduce three different subcategories of the iterative phrase: `while`, `for` and `repeat`.

Copy rules factorize common parts, reducing the grammar size while reducing the readability; furthermore, they don't have any practical utility. Removing these rules create shorter syntax trees: this is a common tradeoff in the design of a formal grammar.

For a grammar G and a non-terminal A, the set $Copy(A) \subseteq V$ is defined as the set of A and all the non-terminals that are immediate of transitive copies of A:

$$Copy(A) = \left\{ B \in V \,\middle|\, \exists \text{ a derivation } A \overset{*}{\Rightarrow} B \right\}$$

Assuming that G is in non-nullable normal form and that the axiom S does not occur in any RP, to compute the set *Copy* the following logical clauses have to be applied until a fixed point is reached:

- $A \in Copy(A)$, initialization
- $C \in Copy(A)$ if $B \in Copy(A) \wedge C \to B \in P$

Finally construct the rule set P' of a new grammar G', equivalent to G and copy free, as follows:

- $P' := P \setminus \{A \to B \mid A, B \in V\}$, cancellation of copy rules
- $P' := P' \cup \left\{ A \to \alpha \,\middle|\, \alpha \in \left((\Sigma \cup V)^* \setminus V\right) \wedge (B \to \alpha) \in P \wedge B \in Copy(A) \right\}$

The effect is that a non-immediate *(multi step)* derivation $A \overset{+}{\Rightarrow} B \overset{+}{\Rightarrow} \alpha$ of G shrinks to the immediate *(one step)* derivation $A \Rightarrow \alpha$ of G' while keeping all the original non copy rules.

If contrary to the hypothesis, G contains nullable terminals, the definition of set *Copy* and its computation must also consider the derivations in form $A \overset{+}{\Rightarrow} BC \overset{+}{\Rightarrow} B$ where non-terminal C is nullable.

3.6.5 Conversion of Left Recursion to Right Recursion

Another normal form, called non left-recursive, is characterized by the absence of left-recursive rules of derivations *(l-recursions)*. This form is needed for **top-down parsers**, to be studied later *(Section 5.5)*.

There are two forms of transformation:

- **Immediate**, explained in this Paragraph
- **Non-immediate**, explained in the book and not treated in the course due to its complexity

Transformation of immediate left recursion   consider all l-recursive alternatives of a non-terminal A:

$$A \to A\beta_1 \mid A\beta_2 \mid \ldots \mid A\beta_h \quad h \geq 1$$

where no string β_i is empty and let the remaining alternatives of A, needed to terminate the recursion, be:

$$A \to \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_k \quad k \geq 1$$

A new secondary non-terminal A' is introduced and the rule set is modified as follows:

$$\begin{cases} A \to \gamma_1 A' \mid \gamma_2 A' \mid \ldots \mid \gamma_k A' \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_k \\ A' \to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_h A' \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_h \end{cases}$$

Now every original derivation involving l-recursive steps such as

$$\underbrace{A \Rightarrow A\beta_2 \Rightarrow A\beta_3\beta_2}_{\text{l-recursive}} \Rightarrow \gamma_1\beta_3\beta_2$$

is replaced by the equivalent derivation:

$$A \Rightarrow \gamma_1 \underbrace{A' \Rightarrow \gamma_1\beta_3 A'}_{\text{r-recursive}} \Rightarrow \gamma_1\beta_3\beta_2$$

3.6.6 Conversion to Chomsky Normal Form

In the **Chomsky Normal Form** *(or CNF)* only two types of rules are allowed:

1. homogeneous binary, $A \to BC$ where $B, C \in V$
2. terminal with a singleton right part $A \to a$ where $a \in \Sigma$

Moreover, if the language contains the empty string, there exists the axiomatic rule $S \to \varepsilon$, but the axiom S is not allowed in any rule RP. With such constraints, any internal node of a syntax tree may have either non-terminal siblings or one terminal sibling.

In order to convert a grammar G into CNF form, the following steps are performed:

1. Each rule $A_0 \to A_1 A_2 \ldots A_n$ of length $n \geq 2$ in converted into a rule of length 2 by singling out the first symbols A_1 and the remaining suffixes $\langle A_2, \ldots, A_n \rangle$

   - a new non-terminal $\langle A_2, \ldots, A_n \rangle$ is introduced
   - a new rule $\langle A_2, \ldots, A_n \rangle \to A_2 \ldots A_n$ is created
   - the original rule is replaced by $A_0 \to A_1 \langle A_2, \ldots, A_n \rangle$

2. The rule of length 2 can still convert terminals (as it's in form $A \to aB, a \in \Sigma$) and has to be replaced

   - a new non-terminal $\langle a \rangle$ is introduced
   - a new rule $\langle a \rangle \to a$ is created
   - the original rule is replaced by $A \to \langle a \rangle B$

3. Repeat step (1) until the grammar is in CNF

3.6.7 Conversion to Real-Time and Greibach normal Form

In the **real-time normal form**, every rule starts with a terminal:

$$A \to a\alpha \text{ where } a \in \Sigma \text{ and } \alpha \in (\Sigma \cup V)^*$$

A special case of real-time form is the **Greibach normal form:**

$$A \to a\alpha \text{ where } a \in \Sigma \text{ and } \alpha \in V^*$$

Every rule starts with a terminal, followed by zero or more non-terminals; both form exclude the empty string ε from the language.

The *real time* designation will be later understood *(Section 5.1)* as a property of the pushdown automaton that can recognize the language: at each step, the automaton reads and consumes an input character. Therefore, the total number of steps equals the length of the input string to be recognized.

In order to simplify a grammar, it can be converted in *Greibach* or *real time* normal form. However, such conversion will not be discussed in this course.

3.7 Free Grammars Extended with Regular Expressions - *EBNF*

The legibility of a *r.e.* can be combined with the expressiveness of a grammar via the **extended context-free grammar** *(EBNF)* notation, which uses the best parts of each formalism. Very simply, a rule RP can be a *r.e.* over terminals and non-terminals. The right part α of an extended rule $A \to \alpha$ of a grammar G is a *r.e.* which, in general, derives an infinite set of strings; each of them can be viewed as the right part of a non-extended rule with infinitely many alternatives.

Consider a grammar G and its rule $A \to \alpha$, where α is a *r.e.* possibly containing the choice operators of star, cross, union, and option. Let α' a string that derives from α, according to the definition of *r.e.* derivation, and does not contain any choice operator. For every pair of strings δ, η, there exists a one step derivation:

$$\delta A \eta \underset{G}{\Longrightarrow} \delta \alpha' \eta$$

Then it's possible to define a multi-step derivation that starts from the axiom and produces a terminal string and consequently can define the language generated by a *EBNF* grammar, in the same manner as for basic grammars.

The tree generated by a *EBNF* grammar is generally shorter and broader than the one generated by its non-extended counterpart.

It can be shown that regular languages are a special case of context-free languages: they are generated by grammars with strong constraints on the rules form. Due to these constraints, the sentences of regular languages present *"inevitable"* repetitions.

3.7.1 From *RE* to *CF*

It's possible to create a *CF (context-free)* grammar that generates the same language of a *r.e.*: a one-to-one correspondence between the respective rules of the two is shown in Table 2.

RE	RP of CF rule
$r = r_1 \cdot r_2 \cdot \ldots \cdot r_k$	$E_1 E_2 \ldots E_k$
$r = r_1 \cup r_2 \cup \ldots \cup r_k$	$E_1 \cup E_2 \cup \ldots \cup E_k$
$r = (r_1)^*$	$EE_1 \mid \varepsilon$ or $E_1 E \mid \varepsilon$
$r = (r_1)^+$	$EE_1 \mid E_1$ or $E_1 E \mid E_1$
$r = b \in \Sigma$	b
$r = \varepsilon$	ε

Table 2: Correspondence between *RE* and *CF* rules

It can therefore be concluded that every regular language is free, while there are free languages that are not regular *(for example the language of palindromes)*. The relation between families of grammars is then:

$$REG \subseteq CF$$

3.8 Linear Grammars

The definition of **linear grammar** is shown in Definition 3.10; refer to Definition 3.1 *(Page 8)* for the definition of *CF* grammar.

> **Definition 3.10** (Linear grammar)**.** A **linear grammar** is a *CF* grammar that has at most one non-terminal in its right part. This family of grammars gives evidence to some fundamental properties and leads to a straightforward construction of the automaton that recognizes the strings of a regular language.

All of its rules have the form:
$$A \rightarrow uBv \quad \text{with } u, v \in \Sigma^*, B \in (V \cup \varepsilon)$$

that is, there's at most one non-terminal in the *RP* of a rule. The family of linear grammar is still more powerful than the family of *RE*.

3.8.1 Unilinear Grammars

The unilinear grammars represent a subset of linear grammars.
The rules of the following form are called respectively **right-linear** and **left-linear:**

- right-linear rule: $A \rightarrow uB$ where $u \in \Sigma^*$ and $B \in (V \cup \varepsilon)$
- left-linear rule: $A \rightarrow Bu$ where $u \in \Sigma^*$ and $B \in (V \cup \varepsilon)$

Both cases are linear and obtained by deleting on either side of the two terminal strings that embrace non-terminal B in a linear grammar. A grammar where the rules are either right or left-linear is termed **unilinear** *(or of type 3)*.
Regular expressions can be translated into unilinear grammars via finite state automata.

3.8.2 Linear Language Equations

It's possible to define a **linear language** as a set of linear equations with regular languages as a solution. the rules illustrated in Section 3.7.1 *(Correspondence between RE and CF rules)* will be used to create the equations. For simplicity, consider a grammar $G = \langle V, \Sigma, R, S \rangle$ in strictly right-linear with all the terminal rules empty *(e.g. $A \to \varepsilon$)*. The case of left-linear grammars is analogous.
A string $x \in \Sigma^*$ is a language $L_A(G)$ if:

- string x is empty *($x = \varepsilon$)*, and set P contains rule $A \to \varepsilon$
- string x is empty *($x = \varepsilon$)*, and set P contains rule $A \to B$ with $\varepsilon \in L_B(G)$
- string $x = ay$ starts with character a, set P contains rule $A \to aB$ and string $y \in \Sigma^*$ is in the language $L_B(G)$

Every rule can be transcribed into a linear equation that has as unknowns the languages generated from each non-terminal.
Let $n = |V|$ be the number of non-terminals of grammar G. Each non-terminal A_i is defined by a set of alternatives:

$$A_i \to aA_1 \mid bA_1 \mid) \mid aA_n \mid bA_n \mid \ldots \mid A_1 \mid \ldots \mid A_n \mid \varepsilon$$

where some of the alternatives are empty. The rule $A_i \to A_i$ is never present since the language is non-circular. Then the set of corresponding linear equations is:

$$L_{A_i} = aL_{A_1} \cup bL_{A_1} \cup \ldots \cup aL_{A_n} \cup bL_{A_n} \cup \ldots \cup L_{A_1} \cup \ldots \cup L_{A_n} \cup \varepsilon$$

where the last term is the empty string.
This system of n equations in n unknowns can be solved via substitution and the **Arden identity**, shown in Definition 3.11.

> **Definition 3.11** (Arden identity)**.** The equation in the unknown language
>
> $$X = KX \cup L$$
>
> where K is a non-empty language and L is any language, has only one unique solution, provided by the **Arden Identity:**
> $$X = K^*L$$

It's simple to see that language K^*L is a solution of the equation $X = KX \cup L$ since by substituting it for the unknown on both sides, the equation turns into the identity:

$$K^*K = (KK^*L) \cup L$$

3.9 Comparison of Regular and Context-Free Grammars

This section is dedicated to the introduction of properties that are useful to show that soma languages are not regular: regular languages *(and therefore unilinear grammars and regular expressions)* share peculiar properties.

3.9.1 Strings Pumping

First of all, recall that in order to generate an infinite language, a grammar has to be recursive, as only a derivation such as $A \overset{+}{\Longrightarrow} uAv$ can be iterated for an unbounded number of times n producing a string $u^n A v^n$.

> **Theorem 3.1** (Pumping Lemma)**.** *Let G be a unilinear grammar. For any sufficiently long sentence x, longer than some constant dependent only on the grammar, it's possible to find a factorization $x = tuv$ where $u \neq \varepsilon$ such that for every $n \geq 0$, the string tu^nv is in the language.*

It can be said that the given sentence can be *pumped* by injecting the substring u arbitrarily many times.

> *Proof.* Consider a strictly right-linear grammar and let k be the number of non-terminal symbols. The syntax tree of any sentence x of length k has two nodes with the same non-terminal label A *(illustrated in Figure 2)*
>
> Consider the factorization into $t = a_1 a_2 \ldots$, $u = b_1 b_2 \ldots$, and $b = c_1 c_2 \ldots c_m$. Therefore, there is recursive derivation:
> $$S \overset{+}{\Longrightarrow} tA \overset{+}{\Longrightarrow} tuA \overset{+}{\Longrightarrow} tuv$$
>
> that can be repeated to generate the string tu^+v. $\qquad\square$

Figure 2: Syntax tree of a sentence of length k in a strictly right-linear grammar

3.9.2 Role of Self-nesting Derivations

Since the *REG* family is strictly included within the *CF* family *(as REG $\subset$ CF)*, the focus of this Section is what makes some languages not regular. Typical non-regular languages, such as the Dyck language, the palindromes, and two power languages have a common feature: a **recursive derivation that is neither left nor right-linear**. Such a derivation has the form:

$$A \overset{+}{\Longrightarrow} \alpha A \beta \quad \alpha \neq \varepsilon \wedge \beta \neq \varepsilon$$

where the *RP* contains the non-terminal symbol A between two strings α and β of both terminals and non-terminals.

A grammar is **not self nesting** if, for all non-terminals A, every derivation $A \overset{+}{\Longrightarrow} \alpha A \beta$ has either $\alpha = \varepsilon$ or $\beta = \varepsilon$; self nesting derivations cannot be obtained with the grammars of the *REG* family.
Therefore:
$$\text{grammars without self nested derivations} \;\Rightarrow\; \text{regular languages}$$

while the opposite does not necessarily hold.

This introduces a big limitation to the family of languages generated via *r.e.*, as all sufficiently long sentences necessarily contain two substrings that can be repeated an unbounded number of times, thus generating self-nested structures.

3.9.3   Closure properties of *REG* and *CF* families

Languages operations can be used to combine existing languages into a new one; when the result of an operation does not belong to the original family it cannot be generated with the same type of grammar.
Therefore, the closure of the *REG* and *CF* families is here explained, keeping in mind that:

- a **non-membership** *(such as $\overline{L} \notin CF$)* means that the left term does not always belong to the family, while some of the complement of the language may belong to the family
- the **reversal** of a language $L(G)$ is generated by the mirror grammar, which is obtained by reversing the right rule parts of the original grammar G
- if a language is **left-linear**, its mirror is **right-linear**, and vice versa
- the symbol $\oplus$ is used as a symbol for **union** *(normally represented as $\cup$ or cdot)*

The closure of the two families is shown in Table 3.

reversal	*star*	*union*	*complement*	*intersection*
$R^R \in REG$	$R^* \in REG$	$R_1 \oplus R_2 \in REG$	$\overline{R} \in REG$	$R_1 \cap R_2 \in REG$
$L^R \in CF$	$L^* \in CF$	$L_1 \oplus L_2 \in CF$	$\overline{L} \notin CF$	$L_1 \cap L_2 \notin CF$

Table 3: Closure of the *REG* and *CF* families

The properties of the *REG* closure are easily proven via finite state automata. The reflection and star properties of *CF* have already been shown while:

- *CF* is **not closed under complement** because it is closed under union but not under intersection
- the closure of *CF* under union can be proven by defining suitable grammars
- the non-closure of *CF* under intersection can be proven by using finite state automata

Free languages can be intersected with regular languages in order to make a grammar more discriminatory, forcing some constraints on the original sentences. The intersection of a free language L with a regular language R is still a part of the *CF* family:

$$L \cap R \in CF$$

This property is shown in Section 5.1.3.

3.10   Chomsky classification of Grammars

Context-free grammars cover the main constructs occurring in technical languages, such as hierarchical lists and nested structures, but fails with other syntactic structures as simple as the replica language or the three-power language.
American linguist *Noam Chomsky* proposed a categorization of languages based on the complexity of their grammars, which is still used today; such categorization is called the **Chomsky hierarchy** and is shown in Table 4.
The relation between language families is shown in Figure 3.

grammar	rule form		family	model
type 0	$\beta \to \alpha$ with $\alpha, \beta \in (\Sigma \cup V)^+$		recursively enumerable	Turing machine
type 1	$\beta \to \alpha$ with $\alpha, \beta \in (\Sigma \cup V)^*$		context sensitive	linear bounded
type 2	$A \to \alpha$ with $A \in V$ and $\alpha \in (\Sigma \cup V)^*$		context-free	pushdown automaton
type 3	$\begin{cases} A \to uB & \textit{(right)} \\ A \to Bu & \textit{(left)} \end{cases}$ with $\begin{cases} A \in V \\ u \in \Sigma^* \\ B \in (V \cup \{\varepsilon\}) \end{cases}$		regular	finite automaton

Table 4: Chomsky hierarchy
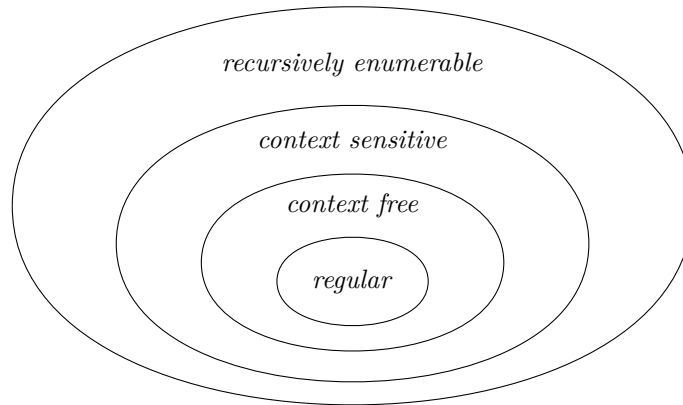
Figure 3: Chomsky hierarchy

4 Finite Automata and Regular Language Parsing

Finite automata are used by compilers to recognize and accept the syntactic structure of a sentence; hence, they are called **acceptors** or **recognizers**.

In order to know whether a string is valid in a specific language, a recognition algorithm is needed: it must produce a `yes` or `no` answer to the question *"is this string valid?"*. The input domain of the automaton is a set of strings over an alphabet Σ.

The answer the recognition algorithm α to a string x, denoted as $\alpha(x)$, in defined as:

$$\alpha(x) = \begin{cases} \textbf{accepted if} & \alpha(x) = \texttt{yes} \\ \textbf{rejected if} & \alpha(x) = \texttt{no} \end{cases}$$

The language of recognized is then denoted of $L(\alpha)$ and is the set of the accepted strings:

$$L(\alpha) = \{x \in \Sigma^* \mid \alpha(x) = \texttt{yes}\}$$

The algorithm itself is assumed to always terminate for every input string, making the recognition problem decidable; however, it if does not terminate for a specific string x, then x is not part of the language $L(\alpha)$. If this happens, the membership problem is semidecidable and the language L is recursively enumerable.

4.1 Recognizing Automaton

An **automaton** is a simple machine that features a small set of simple instructions. The most general form is shown in Figure 4.
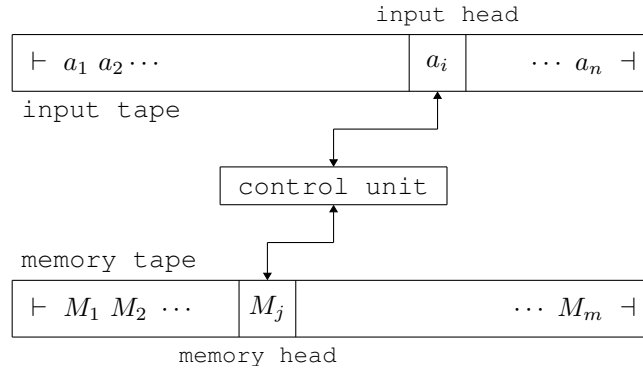
Figure 4: General model of a recognizer automaton

The control unit has a limited store size, represented by a finite set of states; the input and memory tape have unbounded size. In order to represent the start and the end of the data written in the tape, the symbols $\vdash, \dashv$ are respectively used.

The input *(or read-only)* tape contains the given input or source string, one character per cell, while the memory tape can be written to and read from. The automaton can perform the following actions:

- **read** the current character a_i from the **input tape**
- **move** the input tape head to the left or right
- **read** the current symbol M_j from the **memory tape**, optionally replacing it with another symbol
- **move** the **memory tape** and changing the current state of the next one

The automaton processes the source by making a series of moves; the choice of the next move depends on the current input symbol, the current memory symbol, and the current state. A move may have one of the following effects:

- **shifting** the **input head** to the left or right by one position

- **overwriting** the current **memory symbol** with another one and shifting the memory head to the left or right
- **changing** the **state** of the control unit

A machine is **unidirectional** if the input head only moves in one direction *(normally, from left to right)*.

At any time, the future behaviour of the machine depends on the **instantaneous configuration**, shown in Definition 4.1.

Definition 4.1. The **instantaneous configuration** 3-tuple $\langle q, a_i, M_j \rangle$ where:

- $q \in Q$ is the current **state**
- $a_i \in \Sigma$ is the current **input symbol**
- $M_j \in \Sigma$ is the current **memory symbol**

The initial configuration is $\langle q_0, \vdash, \vdash \rangle$:

- $q_0 \in Q$ is the **initial state**
- $\vdash$ is the **start** of the input and memory tapes

After being *"turned on"*, the machine performs a computation *(a sequence of moves)* that leads to new configurations. If more than one move is possible for a certain configuration, the change is called **non-deterministic**; otherwise, it's **deterministic**. Non deterministic automata represent an algorithm that may explore alternative paths in some situations.

A configuration is **final** if the control unit is in a state specified as terminal while the input head is on the terminator $\dashv$. Sometimes an additional constraint is added to the final configuration: the memory tape may contain a specific symbol or string; normally the memory needs to be empty.

The source string x is **accepted** if the automaton, starting in the initial configuration with $x \dashv$ as input, performs a computation that leads to a final configuration; a nondeterministic automaton may reach a final configuration by different computations. The language **accepted** or **recognized** by the machine is the set of accepted strings.

A computation terminates either when the machine has entered a final configuration or when no move can be applied to the current configuration. In the latter case, the string is not accepted by the computation but may be accepted by another, non deterministic, computation. Two automata accepting the same language are called **equivalent**; they can belong to different classes of automata or have different complexities.

The representation of an automaton is usually done by a **transition table** or **transition diagram** (see Figure 5).
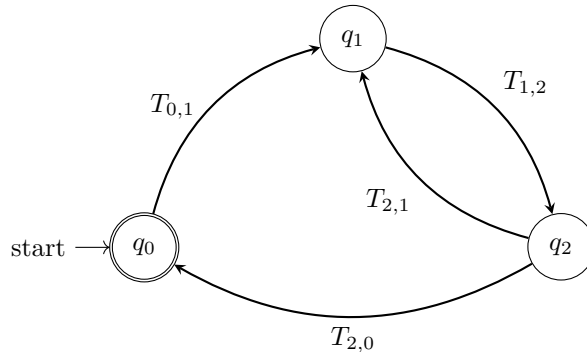
Figure 5: Transition diagram of a deterministic automaton

A formal definition is shown in *(4.2)*.

Definition 4.2 (Deterministic Automaton). A deterministic finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of **states**
- Σ is a finite set of **symbols**
- $\delta : (Q \times \Sigma) \to Q$ is a transition function that maps a **state** and a **symbol** to a **state**
- $q_0 \in Q$ is the **initial state**
- $F \subseteq Q$ is the set of **final states**

The transition function encodes the moves of the automaton M: the meaning of $\delta(q, a) = r$ is that the automaton moves from state q to state r when it reads a from the input tape. This move is also sometimes denoted the symbol:

$$q \xrightarrow{a} r$$

The automaton processes a non-empty string x by making a series of moves, one for each symbol in the string. If the value $\delta(q, a)$ is **undefined** *($\delta(q, a) = \varepsilon$)*, automaton M stops and enters an error state; the strings that were being processed are rejected.

The function δ can be applied recursively as follows:

$$\delta(q, ya) = \delta\left(\delta\left(q, y\right) a\right) \quad a \in \Sigma, y \in \Sigma^*$$

Therefore, the same transition function is defined inductively as:

$$\begin{cases} \delta^* (q, \varepsilon) = q & \text{base case} \\ \delta^* (q, xa) = \delta\left(\delta^*(q, x), a\right), x \in \Sigma^*, a \in \Sigma & \text{inductive step} \end{cases}$$

For brevity, with abuse of notation, δ is also used to denote the function δ^*.

There is a univocal correspondence between the values of δ and the paths in the state transition graph of the automaton: $\delta(q, y) = q'$ if and only if there exists a path from node q to node q', such that the concatenated labels of the path arcs make string y; y is the label of the path, while the path itself represents a computation of the automaton.

A **string** is **recognized** *(or accepted)* by automaton M if it is the label of a path from the initial state to a final. The empty string ε is accepted by M if the initial state is also a final state.

The language $L(M)$ **or accepted** *(or accepted)* by automaton M is the set of all strings accepted by M:

$$L(M) = \{x \in \Sigma^* \mid \delta(q_0, x) \in F\}$$

the class of such languages is called **regular languages**. Two automata are equivalent if they accept the same language; the recognized languages are equivalent as well.

The complexity class of this automaton is called *real time* as the number of steps to accept a string x is equal to its length $|x|$.

Automata are represented via state transition graphs; directed graphs $G = (V, E)$ where:

- V is the set of **states** of the automaton
- E is the set of **arcs**, each labelled with a symbol $s \in \Sigma$, representing the transitions of the automaton

4.1.1 Complete automaton

If a move is not defined in a state q while reading an input symbol a, the automaton enters an error state q_{err} and stops: it can never be left, and no other move can be performed. The error state is also called *sink* or *trap* state.

The state transition function δ can be made total by adding the error state and the transitions from and into it:

$$\begin{cases} \forall q \in Q, \forall a \in \Sigma, \delta(q, a) = \delta(q, a) & \text{if defined} \\ \delta(q, a) = q_{err} & \text{otherwise} \end{cases}$$

4.1.2 Clean automaton

An automaton may contain useless parts that do not contribute to the acceptance of any string: they are best eliminated, as they just bloat the machine by making it more complex. This concept holds for all classes of automata, including non-deterministic ones.

A state is called:

- **reachable** from a state p if there exists a computation going from p to q
- **accessible** if it can be reached from the initial state
- **post-accessible** if a final state can be reached from it

By using these definitions, a state can then be:

- **useful** if it is accessible and post-accessible
- **useless** otherwise

4.1.3 Minimal automaton

An automaton is **clean** *(or minimal)* if all its states are **useful**.

> **Property 4.1** (Equivalent Clean Automaton). For every finite automaton, there exists an equivalent clean automaton. The cleanliness condition can be reached by identifying useless states, deleting them and all their incident arcs.

> **Definition 4.3** (Indistiguishable states). Two states p and q are **indistinguishable** if and only if, for every input string $x \in \Sigma^*$, the next states $\delta(p, x)$ and $\delta(q, x)$ are both final or both not final. This property is a **binary relation**, as it's reflexive, symmetric and transitive; as such, it's an equivalence relation.
> The complementary condition is termed **distinguishable**.

Two **indistinguishable** states can be merged into a single state, as they are equivalent, without changing the language accepted by the automaton. The new set of states is the quotient set with respect to the equivalence class.

4.1.4 Automaton Clearing Algorithm

Computing the indistinguishability relation directly from the definition is an undecidable problem, as it would require computing the whole accepted language which may be infinite.

The distinguishability relation can be computed through its inductive Definition 4.4.

> **Definition 4.4** (Inductive indishinguishability). A state p is distinguishable from a state q if and only if one of the following conditions holds:
>
> - p is **final** and q is not *(or viceversa)*
> - $\delta(p, a)$ is **distinguishable** from $\delta(q, a) \; \forall \, a \in \Sigma$

Consequences

⇒ The **error state** q_{err} **is distinguishable** from every postaccessible state p, because

- $\exists$ string x such that $\delta(p, x) = \in F$ *(postaccessible state)*
- $\forall$ string $x : \delta(q_{err}, x) = q_{err}$ *(error state)*

⇒ The state p is **distinguishable** from q *(both assumed to be postaccessible)* if the set of labels on arcs outgoing from p to q are **different**, while not necessarily disjointed

In fact, if $\exists a$ such that $\delta(p, a) = p'$, with p' postaccessible, and $\delta(q, a) = q_{err}$, then p is distinguishable from q because q_{err} is **distinguishable from every postaccessible state**.

4.1.4.1 Minimal Automaton Construction

The minimal automaton M', equivalent to the given automaton M, has for states the equivalence of the indistinguishability relation. Machine M' contains the arc:

$$\overbrace{[\ldots, p_r, \ldots]}^{C_1} \xrightarrow{b} \overbrace{[\ldots, q_s, \ldots]}^{C_2}$$

between the equivalence classes C_1 and C_2 if and only if machine M contains the arc $p_r \xrightarrow{b} q_s$, between two states, respectively, belonging to the two classes. The same arc of M' may derive from several arcs of M.

The minimization procedure provides proof of the existence and unicity of a minimum automaton equivalent to any given one; this procedure does not hold for non deterministic automata.

State minimization provides a method for checking deterministic automata equivalence:

1. **clean** the automata
2. **minimize** the automata
3. check if they are **identical** *(same number of states and same arcs)*

4.2 Nondeterministic Finite Automata

There are 3 different forms of nondeterminism in finite automata:

1. alternate moves for a unique input *(Figure 6a)*
2. distinct initial states *(Figure 6b)*
3. state change without input consuming, called spontaneous or epsilon move *(Figure 6c)*

(a) Alternative moves for a unique input   (b) Distinct initial states

(c) Spontaneous moves
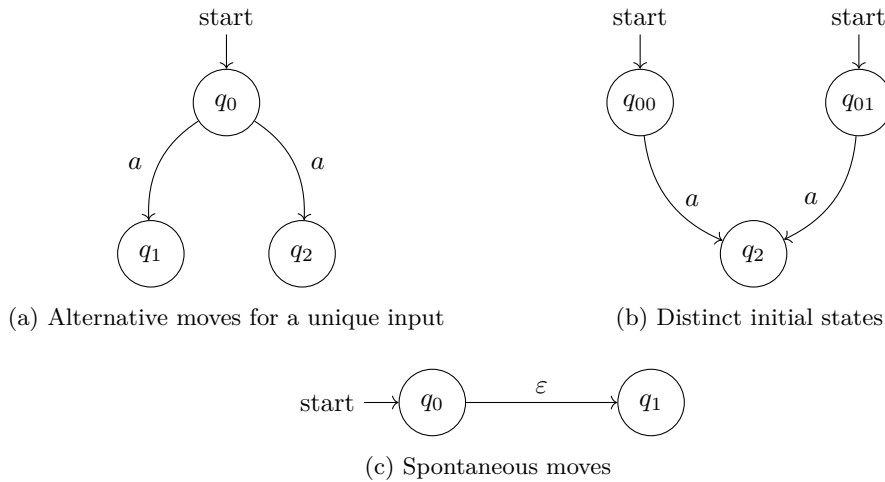
Figure 6: Causes of nondeterminism

For two of them there is an analogy with the corresponding grammar:

- **alternate** moves - grammar with two alternatives $A \to aB \mid aC, a \in \Sigma$
- **spontaneous** moves - grammar with copy rule $A \to \varepsilon$

4.2.1 Motivations for non determinism in finite state automata

While seemingly a nuisance, nondeterminism introduces many useful side effects in the grammar generations. A few motivations are:

- **Concision** - defining a language with nondeterministic machines often results in a more readable and more compact definition

- **Language reflection** - in order to recognize the reversal L^R of language L, the initial and final states must be exchanged while the arcs must be reversed
    - → multiple final states end up being multiple initial states
    - → multiple arcs incident to a state lead to alternate moves

4.2.2 Nondeterministic Finite Automaton

The formal definition of Nondeterministic Finite Automaton is shown in 4.5.

Definition 4.5 (Nondeterministic Finite Automaton). A **Nondeterministic Finite Automaton** N, without spontaneous moves, is a 5-tuple $\langle Q, \Sigma, I, F, \delta \rangle$:

- Q is a finite set of **states**
- Σ is an alphabet of **terminal characters**
- I and F are two subsets of Q, containing respectively the **initials** and **final** states
- δ is a **transition function**, included in the Cartesian product $Q \times \Sigma \times Q$

The machine may have multiple initial states; its representation is analogous to its deterministic counterpart.

As before, a computation of length n is a series of n transitions such that the origin of each corresponds to the destination of the previous. Its representation is:

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \ldots \xrightarrow{a_n} q_n \quad \text{or} \quad q_0 \xrightarrow{a_1 a_2 \ldots a_n} q_n$$

where the computation label is the string $a_1 a_2 \ldots a_n$.

A computation is successful if the first state q_0 is **initial** and the last state q_n is **final**. A string x is **accepted** by the automaton if it's the label of a successful computation.

The empty string ε is accepted if and only if it holds $q_i \in I$ and $q_i \in F$ *(an initial state must be also final)*. The language L_N is recognized by automaton N is the set of accepted strings:

$$L(N) = \left\{ x \in \Sigma^* \,\middle|\, q \xrightarrow{x} r \text{ with } q \in I \wedge r \in F \right\}$$

4.2.2.1 Transition Function

The **Transition Function** δ of a nondeterministic function computes sets of values, as opposed to a deterministic one that computes single values. A formal definition is shown in Definition 4.6.

Definition 4.6. For a machine $N = \langle Q, \Sigma, \delta, I, F \rangle$ with no spontaneous moves, the **transition function** is defined as:

$$\delta : Q \times \Sigma \to \wp(Q)$$

where symbol $\wp(Q)$ indicates the set of all subsets of Q.

The meaning of the function $\delta(q, a) = [p_1, p_2, \ldots, p_n]$ is that the machine, after reading input character a while on state q, can **arbitrarily** move into one of the states $p_1, \ldots, p_n$. The function can be extended to any string y, including the empty one, as follows:

$$\forall q \in Q \quad \delta(q, \varepsilon) = [q]$$

$$\forall q \in Q, \forall y \in \Sigma^* \quad \delta(q, y) = \left[p \mid q \xrightarrow{y} p \right]$$

Or it holds $p \in \delta(p, q)$ if there exists a computation labelled y from q to p. Therefore, the language accepted by automaton N is:

$$L(N) = \{ x \in \Sigma^* \mid \exists q \in I \text{ such that } \delta(q, x) \cap F \neq \emptyset \}$$

i.e. the set computed by function delta must contain a final state for a string to be recognized.

4.2.2.2 Automata with Spontaneous Moves

Another kind of nondeterministic behaviour occurs when an automaton changes state **without reading a character**, performing a **spontaneous move**, represented by an ε-arc.

The number of steps *(and the time complexity)* of the computation can exceed the length of the input string, because of the presence of ε-arcs. As a consequence the algorithm no longer works in real-time, despite having still a linear complexity; the assumption that no cycle of spontaneous moves happens in the computation holds with every machine and string.

The family of languages recognized by such nondeterministic automata is called **finite-state**.

4.2.2.3 Uniqueness of the initial state

The definition of nondeterministic machine *(Section 4.2.2)* allows two or more initial initial states; however, it is possible to construct an equivalent machine with only one.

In order to do so, it suffices to:

1. add a **new state** q_0, which will be the new unique initial state
2. add ε **arcs** going from q_0 to the formerly initial states

Any computation of this new automaton accepts *(or rejects)* a string if and only if the old one accepts *(or rejects)* it.

4.2.2.4 Ambiguity of Automata

An automaton is ambiguous if it accepts a string with two different computations; as a direct consequence, every deterministic automaton is not ambiguous *(or unambiguous)*.

Since there's a one-to-one correspondence between automata and unilinear grammars, the ambiguity of an automaton is equivalent to the ambiguity of the grammar that generates it.

4.2.3 Correspondence between Automata and Grammars

It's possible to build a univocal mapping between a **right-linear grammar** and its corresponding **automaton**, as shown in Table 5. In order to build an automaton from a left-linear grammar, it's necessary to first reverse it and then apply the same mapping; the language has then to be reversed.
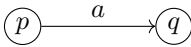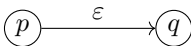
rule #	grammar	automaton
1	non-terminal alphabet $V = Q$	state set $Q = V$
2	axiom $S = q_0$	initial state $q_0 = S$
3	$p \rightarrow aq$ where $a \in \Sigma, p, q \in V$	$\overset{a}{(p) \longrightarrow (q)}$
4	$p \rightarrow q$ where $p, q \in V$	$\overset{\varepsilon}{(p) \longrightarrow (q)}$
5	$p \rightarrow \varepsilon$	final state $(p) \longrightarrow$

Table 5: Correspondence between a right-linear grammar and its corresponding automaton

Consider a **right-linear grammar** $G = (V, \Sigma, P, S)$ and a **nondeterministic automaton** $N = (Q, \Sigma, \delta, q_0, F)$ with a single initial state. Initially, assume that all the grammar rules are strictly **unilinear**: the states Q match the non-terminals V, the initial state q_0 matches the axiom S *(rules 1 and 2 of the Table)*. The pair of alternatives $p \rightarrow aq \mid ar$ corresponds to a pair of nondeterministic moves *(rule 3)*. A copy rule matches a spontaneous move *(rule 4)*; finally, a final rule matches a final state *(rule 5)*.

Every grammar derivation matches a machine computation and vice versa: as such, a language is recognized *(or accepted)* by a finite automaton if and only if it's generated by a unilinear grammar.

4.2.4 Correspondence between Grammars and Automata

In real-world applications, it's sometimes needed to compute the *r.e.* for the language defined by a machine. Since an automaton is easily converted into a right-linear grammar, the *r.e.* of the language can be computed by solving linear simultaneous equations.

The next direct elimination method, named *BMC* after *Brzozowski* and *McCluskey*, is often more convenient.

4.2.4.1 Brzozowski-McCluskey Algorithm

> **Definition 4.7** (Brzozowski-McCluskey (BMC) Algorithm)**.** Let $N = (Q, \Sigma, \delta, q_0, F)$ be a **nondeterministic finite automaton**. For simplicity, suppose that the initial state i is **unique** and no arcs enter it; if not, a new state i' can be added, with ε arcs going from i' to i. The final state t is unique or can be made unique using the same technique.
> Every state other than i *(or i')* and t *(or t')* is internal.
> An equivalent automaton, called **generalized**, is built by allowing the arc tables to be not just terminal characters but also regular languages (*a label can be a r.e.*).
> The idea is to eliminate the internal states one by one while compensating them by introducing new arcs labelled with an *r.e.* until only the initial and final states remain; then the label of arc $i \to t$ is the *r.e.* of the language.

4.2.5 Elimination of Nondeterminism

While, as already discussed, nondeterminism might be useful while designing a machine, it's often necessary to eliminate it in order to obtain a more efficient design; thanks to the following property, an algorithm that eliminates non determinism can be easily implemented.

> **Property 4.2** (Equivalence of Deterministic and Nondeterministic automata)**.** Every nondeterministic automaton can be transformed into a deterministic one; every unilinear grammar admits an equivalent nonambiguous grammar.

The determinization of a finite automaton is conceptually separated into two phases:

1. elimination of spontaneous moves *(ε-moves)*, thus obtaining generally deterministic machine
    - → if a machine has multiple initial states, a new initial state is added, with ε arcs going from it to the old initial states
2. replacement of multiple nondeterminism transitions with one transition that enters a new state
    - → this phase is called **powerset construction**, as the new states constitute a subset of the state set
    - → this phase is not covered in the course

4.2.5.1 Elimination of Spontaneous Moves

The elimination of spontaneous moves is divided into 4 steps:

1. transitive closure of ε-moves
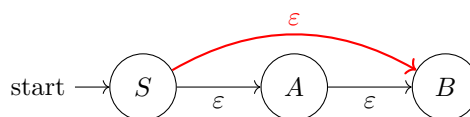
2. backward propagation of scanning moves over ε-moves
3. backward propagation of the finality condition for final states reached by ε-moves
4. elimination of ε-moves and useless states
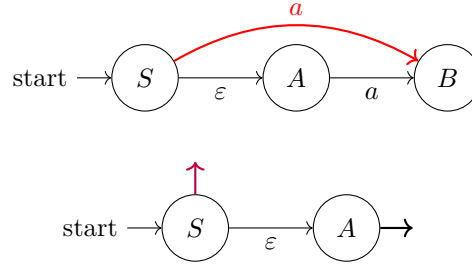
An algorithm that eliminates spontaneous moves following the previous steps is described as follows: let δ be the original state transition graph, and let $F \subseteq Q$ be the set of final states. Furthermore, let a ε-path be a path made only by ε-arcs.

Input   A finite state automaton with ε-moves

Output   An equivalent finite state automaton without ε-moves

The pseudocode is shown in Code 1.

```
// transitive closure of the ε-paths
do
   if graph δ contains a path p⟹q⟹r with p ≠ q then:
      add the arc p⟹r to graph δ
   end if
until no more arcs have been added in the last iteration
// backward propagation of the scanning moves over the ε-moves
do
   if graph δ contains a path p⟹q⟹r with p ≠ q then:
      add the arc p⟹r to graph δ
   end if
until no more arcs have been added in the last iteration
// new final states
F ≔ F ∪ { q | the ε-arc q→f is in δ and f ∈ F }
// clean up
delete all the ε-arcs from graph δ
delete all the states that are not accessible from the initial state
```

Code 1: Direct elimination of spontaneous moves

4.3   From Regular Expressions to Recognizers

When a language is specified via a *r.e.*, it's often necessary to build a machine that recognizes it; two main construction methods are possible.

- **Thompson** *(or structural)* method
    - builds the recognizer of subexpressions
    - combines them via ε-moves
    - resulting automata are normally nondeterministic
    - explained in Section 4.3.1

- **Berry-Sethi** *(or BS)* method
    - builds a deterministic automaton
    - the result is not necessarily minimal
    - explained in Section 4.3.3

4.3.1 Thompson Structural Method

Given an *r.e.*, the **Thompson method** builds a recognizer of the language by building the recognizer of the subexpressions and combining them via ε-moves.

In this construction, each component machine is assumed to have only one initial state without incoming arcs and one final state without outgoing arcs. If not, a new initial state is added, with ε arcs going from it to the old initial states, and a new final state is added, with ε arcs going from the old final states to it.

The Thompson method incorporates the mapping rules between *r.e.* and automata schematized in Table 5 *(described in Section 4.2.3)* and the rules shown in Table 6. Such machines have many non deterministic bifurcations, with outgoing ε-arcs.
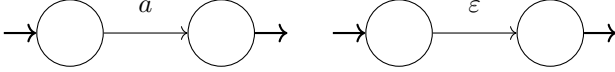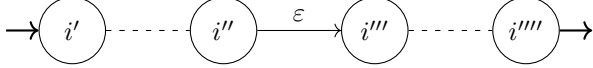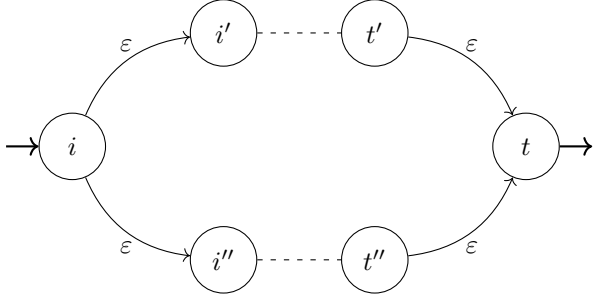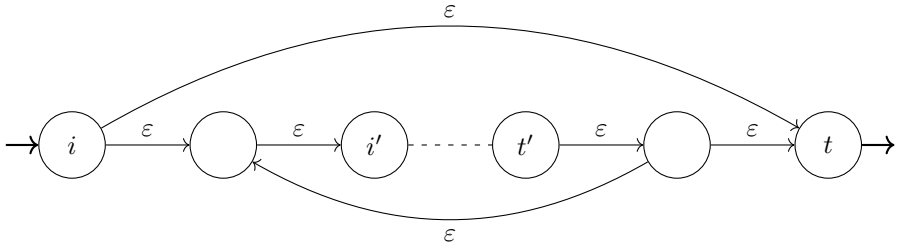
Table 6: Thompson rules

The validity of this method comes from it being a reformulation of the closure properties of regular languages under concatenation, union and Kleene star *(as described in Section 2.2)*.

4.3.2 Local languages

In order to justify the use of the next method, it's necessary to define the family of **local languages** *(also called locally testable or LOC)*, a subset of regular languages.

The *LOC* family is a **proper subfamily** of the *REG* family

$$LOC \subset REG, \ LOC \neq REG$$

For a language L over an alphabet Σ, the local sets are called Ini, Fin, and Dig, and are defined in 4.8, 4.9, and 4.10.

Definition 4.8 (Set of initials). The set of **initials** *(the starting characters of the sentences)* is:

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

Definition 4.9 (Set of finals)**.** The set of **finals** *(the ending characters of the sentences)* is:

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

Definition 4.10 (Set of digrams)**.** The set of **digrams** *(the substrings of length 2 present in the sentences)* is:

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

Furthermore, an additional set *(the **complementary digrams**)* is defined in 4.11.

Definition 4.11 (Set of complementary digrams)**.** The set of **complementary digrams** is:

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

Thanks to the previous definitions, it's possible to define the **Local Language** *(Definition 4.12)*.

Definition 4.12 (Local language)**.** A language L **is local** *(and as such $L \in LOC$)* if and only if it satisfies the following identity:

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

In other words, the non-empty phrases of language L are defined precisely by sets Ini, Fin, Dig.

Not every language is local, but it should be clear that every language L satisfies the previous condition if the equality ($=$) is replaced by an inclusion ($\subset$); by definition, every sentence starts and ends with a character respectively from $\text{Ini}(L)$ and $\text{Fin}(L)$ and its digrams are contained in $\text{Dig}(L)$, but such conditions may be also satisfied by other strings that do not belong to the language.

The definition provides a necessary condition for a language to be local and therefore a method for proving that a language is not local.

4.3.2.1 Automata Recognizing Local Languages

The interest in languages in the LOC family spawns from the simplicity of their recognizers: they need to scan the string from left to right, checking that:

1. the **initial** character is in $\text{Ini}(L)$
2. any pairs of **adjacent** characters are in $\text{Dig}(L)$
3. the **final** character is in $\text{Fin}(L)$

The recognizer of the local language specified by sets Ini, Fin, Dig is a deterministic automaton constructed as follows:

- The **initial state** q_0 is unique
- The **non-initial state set** is Σ - each non-initial state is identified by a terminal character
- The **final state set** is Fin, while no other state is final
  - if $\varepsilon \in L$, then q_0 is also final
- The **transitions** are $q_0 \xrightarrow{a} a$ if $a \in \text{Ini}$ and $a \xrightarrow{b} b$ if $ab \in \text{Dig}$
  - δ can also be defined as:
  $$\begin{cases} \forall\, a \in \text{Ini} & \delta(q_0, a) = a \\ \forall\, xy \in \text{Dig} & \delta(x, y) = y \end{cases}$$

Such an automaton is in the state identified by the letter $\oplus$ *(where $\oplus$ is a character $\in \Sigma$)* if and only if the string scanned so far ends with $\oplus$ *(the last read character is $\oplus$)*. The automaton acts like it has a sliding window with a width of two characters, moving from left to right, triggering the transition from the previous state to the current one if the current digram is in $\text{Dig}(L)$.

This automaton might not be minimal; a stricter accepting condition for a language L is given by a minimal automaton obtained from the normalized local automaton by merging its indistinguishable states.

4.3.3   Berry-Sethi Method

The **Berry-Sethi** *(or BS)* method derives a **deterministic automaton** that recognizes the language specified by the *r.e.*. It works by combining the steps to build a normalised local automaton and the following determinization of such an automaton.

Let e be a *r.e.* of alphabet Σ and let $e' \dashv$ be its **numbered version** over Σ_N terminated by the end marker. For each symbol $a \in e'$, the set of **Followers** of a (or $\text{Fol}(a)$) is defined as the set of symbols in every string $s \in L(e' \dashv)$ that immediately follow a:

$$\text{Fol}(a) = \{ b \mid ab \in \text{Dig}(e' \dashv) \}$$

Hence:

$$\dashv \in \text{Fol}(a) \quad \forall a \in \text{Fin}(e')$$

The algorithm *(shown in Code 2)* tags each state with a subset of $\Sigma_N \cup \{\dashv\}$. A state is created and marked as unvisited, and upon examination, it is marked as visited to prevent multiple examinations. The final states are those containing the end marker $\dashv$.

Input   the sets Ini and Fol of a numbered *r.e. e'*.

Output   recognizer $A = \langle \Sigma, Q, q_0, \delta, F \rangle$ of the unnumbered *r.e. e*.

```
I(q_0) := Ini(e′ ⊣) // create initial state
unmark q_0
Q := { q_0 } // create queue of unmarked states
δ := {} // create transition function
while ∃ unmarked q ∈ Q do // process each unmarked state
  for each a ∈ Σ do // scan each input symbol a
    I(q_1) := {} // create new empty state
    unmark q_1
    for each a_i ∈ I_a(q_1) do // scan each symbol a_i of class a in q_1
      I(q_1) := I(q_1) ∪ Fol(a_i) // add followers of a_i to I(q_1)
    end for
    if q_1 != {} then // if the new state is not empty
      if q_1 ∉ Q then // if the new state is not in Q
        Q := Q ∪ { q_1 } // add q_1 to queue
      end if
      δ := δ ∪ { (q →ᵃ q_1) } // add transition
    end if
  end for
  mark q
end while
F := { q ∈ Q | ⊣ ∈ I(q) } // create final states set F
```

Code 2: Berry-Sethi Algorithm

4.3.3.1 Berry-Sethi Method to Determinize an Automaton

The *BS* algorithm is a valid alternative to the powerset construction *(seen in Section 4.2.5)* for converting a nondeterministic machine N into a deterministic one M.
The algorithm is defined in the following steps.

Input   a **non deterministic automaton** $N = \langle Q, \Sigma, \delta, q_0, F \rangle$; note that every form of non determinism is allowed.

Output   a **deterministic automaton** $M = \langle Q, \Sigma, \delta, q_0, F \rangle$.

1. Number the **labels of the non-ε arcs** of automaton N, obtaining the numbered automaton N' with alphabet σ_N
2. Compute the **local sets** Ini, Fin, and Fol for language $L(N')$ by inspecting the graph of N' and exploiting the identity $\varepsilon a = a \varepsilon = a$
3. Construct the deterministic automaton M by applying *BS Algorithm (see Code 2)* to the sets Ini, Fin, and Fol
4. *(optional)* **minimize** M by merging indistinguishable states

4.3.4 Recognizer for complement and intersection

As already seen *(Section 2.2)*, the *REG* family is closed under complementation and intersection. Let L and L' be two regular languages. Their complement $\overline{L}$ and intersection $L \cap L'$ are regular languages as well.

4.3.4.1 Complement

It's possible to build a recognizer for the **complement** of a regular language L with the following steps.

Input   a **deterministic** finite state automaton M.

Output   a **deterministic** finite state automaton $\overline{M}$ that recognizes the complement of $L(M)$.

1. create a **new sink state** $p \notin Q$ and the state set $\overline{M}$ is $Q \cup \{p\}$
2. the **transition function** $\overline{\delta}$ of $\overline{M}$ is:

$$\overline{\delta}(q, a) = \begin{cases} \delta(q, a) & \text{if } \delta(q, a) \in Q \\ p & \text{if } \delta(q, a) = \epsilon \quad \text{\textit{(if } δ \textit{ is not defined)}} \\ p & \text{for every } a \in \Sigma \end{cases}$$

3. the **final state set** of $\overline{M}$ is $\overline{F} = (Q \setminus F) \cup \{p\}$

For this construction to work, the input automaton M must be deterministic; otherwise, the language accepted by the constructed machine may be not disjoint from the original one, violating the complement property.

4.3.5 Intersection

The construction of the recognizer for the **intersection** of two regular languages L and L' is similar to the one for the complement: it's created exploiting the *De Morgan* identity $L_1 \cap L_2 = \neg(\overline{L_1} \cup \overline{L_2})$. Therefore the algorithm is:

1. Build the deterministic recognizers of L_1 and L_2
2. Derive the recognizers of $\overline{L_1}$ and $\overline{L_2}$
3. Build the recognizers of $\overline{L_1} \cup \overline{L_2}$ by using the Thompson method *(see Section 4.3.1)*
4. Determinize the automaton
5. Derive the complement automaton

An alternative, more direct, technique consists in building the cartesian product of given machines M' and M''; such automaton accepts the intersection of languages, as shown in the following Paragraph.

4.3.5.1 Cartesian Product of two automata

The product of two automata M' and M'' is an automaton M that accepts the language $L(M') \cap L(M'')$, assuming that M' and M'' are devoid of ε-transitions.
The product machine M is defined in 4.13.

> **Definition 4.13** (Product of two automata)**.** The **product machine** M of M' and M'' has state set $Q' \times Q''$ *(the cartesian product of the two state sets)*; as a consequence, each state is a pair $\langle q', q'' \rangle$, where the $q' \in Q'$ is a state of M' and $q'' \in Q''$ is a state of M''. For such a pair or product state $\langle q', q'' \rangle$, the outgoing arc is defined as
> $$\langle q'q'' \rangle \xrightarrow{a} \langle r'r'' \rangle$$
> if and only if there exist the arcs $q' \xrightarrow{a} r'$ in M' and $q'' \xrightarrow{a} r''$ in M''.
> The initial state set I of M is the product $I = I' \times I''$ of the initial state sets of each machine. The final state set F of M is the product $F = F' \times F''$ of the final state sets of each machine.

To justify the correctness of the product construction, consider any string $x \in L(M') \cap L(M'')$. Since string x is accepted by a computation of M' and by a computation of M'', it is also accepted by one of the machine M that traverses the state pairs respectively traversed by the two computations.
Conversely, if x is not in the intersection, at least one of the computations by M' or M'' does not accept x *(as it does not reach a final state)*; therefore, M does not reach a final state either.

5 Pushdown Automata and Context-Free languages parsing

The algorithms for recognizing whether a string is a legal sentence of context-free languages require more memory than the ones for regular languages. The topic of free language parsing via **pushdown automata** *(in Section 5.1)* and **parsing** *(in Section 5.2)* will be introduced.

Normally, compilers and interpreters are built using a parser generator, which is a program that takes as input a grammar and produces a parser for that grammar.

5.1 Pushdown automaton

A **pushdown automaton** *(or PDA)* is a *FSA (see Section 4)* that uses an *unbounded stack* to store information about the current state of the computation. The stack is organized as a `LIFO` structure, where the last element inserted is the first one to be removed; it stores the symbols $A_1, \ldots, A_k$ and often the a special symbol Z_0 is used to denote its end:

$$Z_0 \mid \overset{\text{bottom}}{A_1} \quad A_2 \ldots \overset{\text{top}}{A_k} \quad k \geq 0$$

The input tape is read left to right and the currently read character is called **current** or *cc*; the tape is often delimited on the right by a special end marker $\dashv$:

$$a_1, \ldots, \overset{\text{current}}{a_i} \ldots, a_n \dashv \quad n \geq 0$$

The following three operations are performed on the stack:

- **Pushing**: the symbol A is added to the top of the stack
    - $\rightarrow$ several push operations $push(B_1), \ldots, push(B_m)$ can be combined in one command:

$$push(B_1, \ldots, B_m)$$

- **Popping**: the top symbol of the stack is removed *(if present)*
- **Emptiness** test: the predicate empty is true if and only if $k = 0$ *(the stack is empty)*

At each instant, the machine configuration is specified by the remaining portion of the input string left to read, the current state and the stack contents. Within a move, an automaton can:

- **read** the current character and **shift** the reading head or perform a move without reading *(spontaneous move)*
- **read** and **pop** the top symbol, or read the bottom Z_0 if the stack is empty
- **compute** the next state from the current values of the state, character and top-of-stack symbol
- **push** zero, one or more symbols into the stack

Definition 5.1 (Pushdown Automaton). A Pushdown Automaton M is a 7-tuple $\langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$ where:

- Q is a finite set of **states** of the control units
- Σ is the **input alphabet**
- Γ is the **stack alphabet**
- δ is the **transition function**
    - $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow \wp(Q \times \Gamma^*)$
    - ε is caused by the spontaneous moves
    - $\wp$ is the power set operator
- $q_0 \in Q$ is the **initial state**
- $Z_0 \in \Gamma$ is the bottom of the **stack symbol**
- $F \subseteq Q$ is the set of **final states**

Definition 5.2 (Instantaneous configuration)**.** The **instantaneous configuration** of the machine M is a 3-tuple $\langle q, y, \eta \rangle$ where:

- $q \in Q$ is the **current state**
- $y \in \Sigma$ is the **unread portion** of the input string
    - the input string is composed of characters or tokens
- η is the **stack content**

The domain of the configuration is $Q \times \Sigma^* \times \Gamma^+$.

Particular configurations are:

- **Initial** configuration (q_0, x, Z_0)
- **Final** configuration (q, ε, Z_0) *(if $q \in F$)*

Definition 5.3 (Pushdown Automaton moves)**.** The moves are defined as follows:

- A **reading** move: $\delta(q, a, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \dots, (p_n, \gamma_n)\}$ with $n \geq 1$, $Z \in \Gamma$, $p_i \in Q$, $\gamma_i \in \Gamma^*$
- A **spontaneous** move: $\delta(q, \varepsilon, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \dots, (p_n, \gamma_n)\}$ with $n \geq 1$, $Z \in \Gamma$, $p_i \in Q$, $\gamma_i \in \Gamma^*$
- A **nondeterministic** move: for any given configuration there exist more than one possible move among reading and spontaneous

Notes

- The choice of the i-th action of n possibilities is not **deterministic**
- The reading head automatically shifts forward on input
- The top symbol is always popped, while the string pushed in the stack may be empty
- While any move performs a pop that erases the top symbol, the same symbol can be pushed again by the same move

The transition of a configuration to the next one is possible if there exists the transition between the current configuration and the next one:

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

The transition sequence is represented via the symbols:

$$\xrightarrow{+} \quad \text{and} \quad \xrightarrow{*}$$

respectively for a sequence of *one or more* and *zero or more* moves.

A string x is **recognized** *(or accepted)* **by final state** if there exists a computation that entirely read the string and terminates in a final state:

$$(q_0, z, Z_0) \xrightarrow{*} (q, \varepsilon, \lambda) \quad q \in F, \lambda \in \Gamma^+$$

When the machine recognizes and halts, the stack contains some string λ not further specified, since the recognition modality is by final state; as such, the string λ is not necessarily empty. The applied moves defined by their current and next configurations are shown in Table 7.

Transition functions of pushdown automata are usually represented via a transition diagram *(see Figure 7)*: the stack alphabet features 3 symbols (A, B, Z_0). The arc

$$q_0 \xrightarrow[\varepsilon]{a, A} q_1$$

denotes any reading move from the initial state q_0 to the state q_1 with the input symbol a and the stack top symbol A; no symbol is pushed in the stack, hence the empty string ε.

current configuration	next configuration	applied move
$(q, az, \eta Z)$	$(p, z, \eta \gamma)$	reading move $\delta(q, a, Z) = \{(p, \gamma), \rightarrow\}$
$(q, az, \eta Z)$	$(p, az, \eta \gamma)$	spontaneous move $\delta(q, \varepsilon, Z) = \{(p, \gamma), \rightarrow\}$

<div align="center">Table 7: Pushdown automaton moves</div>
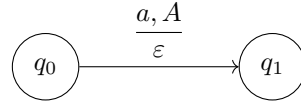
<div align="center">Figure 7: Pushdown automaton transition diagram</div>

5.1.1 Varieties of Pushdown Automata

5.1.1.1 Accepting Modes

The acceptance condition of a pushdown automaton can be:

1. **Empty stack**: the stack must be empty at the end of the computation
2. **Final state**: the computation must end in a final state

The two of them can be combined into recognition of **final state and empty stack**.
For the family of nondeterministic pushdown automata, all three acceptance modes are equivalent: acceptance by empty stack, by final state and combined, have the same capacity with respect to language recognition.

5.1.1.2 Absence of Spontaneous Loops

A **spontaneous loop** is a sequence of moves that starts from a configuration and returns to the same configuration without reading any input symbol. Any pushdown automaton can be converted into an equivalent one:

- **without cycles** of spontaneous moves
- which can decide **acceptance** right after reading the last input symbol

5.1.1.3 Real Time Pushdown Automata

An automaton works in real time if at each step it reads an input character *(i.e. if it does not perform any spontaneous moves)*; this definition applies both to deterministic and non deterministic machines.
In particular, a Pushdown Automaton has the **real time property** if the transition function δ is such that for all the states $q \in Q$ and for all the stack symbols $A \in \Gamma$, the value of $\delta(q, \varepsilon, A)$ is undefined; in other words, the domain of the transition function is $Q \times \Sigma \times \Gamma$ and not $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$.
For every context-free language, there exists a nondeterministic pushdown machine that has the real-time property and recognizes the language.

5.1.2 From Grammar to Nondeterministic Pushdown Automaton

Given a grammar $G = (V, \Sigma, P, S)$, Table 8 shows the construction of a nondeterministic pushdown automaton M that recognizes the language $L(G)$. As before, letter b denotes a terminal symbol, letters A and B denote non-terminals, letter A_i represents any symbol, and cc represents the current symbol.

rule #	grammar rule	automaton rule
1	$A \to B\,A_1\,\ldots\,A_m,\ m \geq 0$	if top $= A$, then *pop*; *push*$(A_m\,\ldots\,A_1\,B)$
2	$A \to b\,A_1\,\ldots\,A_m,\ m \geq 0$	if $cc = b$ and top $= A$, then *pop*; *push*$(A_m\,\ldots\,A_1)$; *shift* reading head
3	$A \to \varepsilon$	if top $= A$, then *pop*
4	for any character $b \in \Sigma$	if $cc = b$ and top $= b$, then *pop*; *shift* reading head
5	*acceptance condition*	if $cc =\dashv$ and the stack is *empty*, then *accept*

Table 8: Grammar to Pushdown Automaton

Rules comment

#1. To recognize A, orderly recognize $B\,A_1\,\ldots\,A_m$
#2. Character b is expected as next and is read, so it remains to orderly recognize $A_1\,\ldots\,A_m$
#3. The empty string deriving from A is recognized
#4. Character b is expected as next and is read
#5. The input string has been entirely scanned and the stack is empty

Rules details

- For rules of Form 2, the *RP* of the rule starts with a terminal and the move is triggered on reading it
- Rules of Form 1 and 3, create spontaneous moves that do not check the current character
- Rules of Form 4 checks that a terminal surfacing the stack matches the current character
- Rules of Form 5 accepts the string if the stack is empty upon reading the end marker $\dashv$

Initially, the stack contains only the bottom symbol Z_0 and the reading head is positioned on the first character of the string. At each step, the automaton chooses a move among the possible ones and applies it; the choice is not deterministic. The machine recognizes the string if there exists a computation that ends with move 5: the sentence is recognized by empty stack.

The conversion rules of grammars into pushdown automata are not unique, and the resulting automata are not equivalent; however, the correspondence between the two is bidirectional. Therefore, the family of context-free languages *CF* coincides with the family of the languages recognized by a pushdown automaton.

5.1.3 Intersection of Regular and Context-free Languages

The statement introduced in Section 3.9.3 *(Closure properties of REG and CF families)*

$$CF \cap REG \in CF$$

can now be proven via pushdown automaton.
Given grammar G and automaton A, a *pushdown automata M* accepting $L(G) \cap L(A)$ is constructed as follows:

1. build an automaton N accepting $L(G)$ via **empty stack**
2. build machine M, product of N and A, applying the known construction for finite automata adapted so that the product machine M manipulates the stack in the same way as N

The resulting pushdown automaton incorporates the states of A; it can check that input $x \in L(A)$.
The machine thus built:

- has internal states that are the product of the state sets of the component machines
- accept with final state and empty stack
- final states are those including a final state of the finite automaton A
- is deterministic fi so are both machines N and A
- accepts exactly the strings of $L(G) \cap L(A)$

5.1.4 Deterministic Pushdown Automata and Languages

Deterministic recognizers *(and their corresponding languages)* are widely adopted in compilers thanks to their computational efficiency. While observing a pushdown automaton, three different non deterministic situations can be found, namely the uncertainty between:

1. **reading moves** if, for a state q, a character a and a stack symbol A, the transition function δ has two or more values
2. **a spontaneous move and a reading move**, if both $\delta(q, \varepsilon, A)$ and $\delta(q, a, A)$ are defined
3. **spontaneous moves**, if for some state q and symbol A, the function $\delta(q, \varepsilon, a)$ has two or more values

If none of the three forms occurs in the transition function δ, then the pushdown machine is **deterministic**; sometimes deterministic *PDA* are called *DPDA*. The language recognized by a deterministic pushdown machine is called **deterministic** as well and the family of such languages is called *DET*.
The family of deterministic languages is a proper subfamily of context-free languages:

$$DET \subset CF$$

Due to a direct consequence of this definition, a deterministic pushdown machine allows spontaneous moves.

5.1.4.1 Deterministic Pushdown Automata and Deterministic Language

If a language is accepted by a deterministic automaton, each sentence is recognized with exactly one computation and it's provable that the language is generated by a non-ambiguous grammar.

> **Property 5.1** (Deterministic Pushdown Automata and Deterministic Language)**.** Let M be a *DPDA* and $L(M)$ the language recognized by M. The following statements are equivalent:
>
> 1. $L(M)$ is **deterministic**
> 2. $L(M)$ is generated by a **non ambiguous** grammar
> 3. $L(M)$ is generated by a **deterministic** grammar

5.1.4.2 Simple Deterministic Languages

A grammar is called **simple deterministic** if it satisfies the next conditions:

1. every rule RP starts with a **terminal character**
    - empty rules and rules starting with **non-terminal characters** are not allowed
2. for any non-terminal A, there do no exist alternatives that start with the same character

A formal definition of simple deterministic grammar is shown in 5.4.

> **Definition 5.4** (Simple Deterministic Grammar)**.** A grammar G is called **simple deterministic** if it satisfies the following condition:
>
> $$\nexists (A \to a\alpha \,|\, a\beta) \quad a \in \Sigma, \ \alpha, \beta \in (\Sigma \cup V)^*, \ \alpha \neq \beta$$

5.1.5 Closure propertires of deterministic *CF* languages

Deterministic languages are a proper subclass of context-free languages.
Table 9 shows the closure properties of the *DET* family of languages:

- L denotes a language belonging to the *CF* family
- D denotes a language belonging to the *DET* family
- R denotes a language belonging to the *REG* family

A procedure to determine if a given *CF* grammar is also in the *DET* family will be shown later.

operation	property	property already known
reflection	$D^R \notin DET$	$D^R \in CF$
union	$D_1 \cup D_2 \notin DET, \ D \cup R \in DET$	$D_1 \cup D_2 \in CF$
complement	$\overline{D} \in DET$	$\overline{L} \notin CF$
intersection	$D \cap R \in DET$	$D_1 \cap D_2 \notin CF$
concatenation	$D_1 \cdot D_2 \notin DET, \ D \cdot R \in DET$	$D_1 \cdot D_2 \in CF$
star	$D^* \notin DET$	$D^* \in CF$

Table 9: Closure properties of *DET* languages

5.2   Parsing

Parsing, also called *Syntax* , is the process of analysing a string of symbols according to a grammar, intending to determine a derivation *(or syntax)* tree; an analyser is simply a recognizer capable of recording a derivation tree, reading the input and eventually stopping on an error or accepting the string. If the source string is ambiguous, the result of the analysis is a set of derivation trees, also called tree forest.

As studied before *(Section 3.3.1)* the same syntax tree corresponds to many derivations, notably the leftmost and rightmost ones, as well as to less relevant others. Depending on the derivation being the leftmost or rightmost, and on the construction order, two important parser classes are obtained:

1. **top-down parsers**: construct the **leftmost derivation** starting from the axiom, growing the root toward the leaves
   - each algorithm step corresponds to a **derivation** step
   - the syntax tree is built through **expansions**

2. **bottom-up parsers**: construct the **rightmost derivation** starting from the leaves, growing them toward the root
   - each algorithm step corresponds to a **reduction**
   - the syntax tree is built through **reductions**

In order to store the previous states in a stack of unbounded length, **pointers** are used; the **pointer** is a reference to a node in the syntax tree, and it is used to store the current state of the parser.

5.2.1   Bottom-up and Top-down

Bottom-up Analysis   The right bottom-up parser builds a derivation in reverse order because the input string is read from right to left. The reduction operations transform the prefix of a phrase form into a string $\alpha \in (V \cup \Sigma)^*$, called **viable prefix**, that may include the result of previous reductions *(stored in the stack)*.
At each step of the , the parser must decide whether:

- continue and **read the next symbol** via a shift operation
- **build a subtree** for a portion of the viable prefix

The choice is made based on the symbols coming after the current one *(lookahead)*.

In principle, syntax analysis would work as well from right to left by scanning the reversed source string; however reversing the scanning order may cause a loss of determinism, as the *DET* family is not closed under reflection *(as shown in Table 9)*.
Moreover, all existing languages are designed for left-to-right processing, as their reading direction matches the natural language direction *(left to right)*.

Top-down Analysis The top-down parser builds a derivation in the natural order because the input string is read from left to right. The parsing procedure starts from the axiom *(the root of the syntax tree)* and it grows the tree by expanding the non-terminals *(the leaves of the syntax tree)*.

5.3   Grammars as networks of Finite Automata

A grammar can be represented as a **network of finite automata**. Despite looking like a useless mind experiment, it has several advantages:

- Offers a pictorial representation of the grammar
- Gives evidence of similarities between parsers
- Allows handling grammars with regular expressions
- Maps recursive descent parser implementations

In grammar, each non-terminal is the left part of one or more alternatives; if a grammar G is in the extended context-free form *(see Section 3.7)*, a rule RP may contain the union operator, which makes it possible to define each non-terminal by just one rule, for example:

$$A \to \alpha \quad \alpha \ r.e. \in (\Sigma \cup V)^*$$

The *r.e.* α defines a regular language, which can be recognized by an automaton M_A. In the trivial case, where α contain just terminal symbols, M_A recognizes $L_A(G)$ starting from non-terminal A; generally, M_A recognizes $L_A(G)$ starting from the initial state and invoking the automaton M_B for each non-terminal B found in α *(for rules in form $A \to B$, $B \to \beta$)*.
The case where $B = A$ is accepted; the resulting invocation is called *recursive.*

A formal definition is given in 5.5.

> **Definition 5.5** (Grammar as network of automata)**.** Let Σ, V and S be respectively the set of **terminals**, **non-terminals** and **axiom** of an *EBNF* grammar G. For each non-terminal A there is exactly one grammar rule $A \to \alpha$ and the rule right part α is a *r.e.* over alphabet $\Sigma \cup V$.

Denotations

- $S \to \sigma$, $A \to \alpha$, $B \to \beta$, ... represent the **grammar rules**
- R_S, R_A, R_B, ... represent the **regular languages** over alphabet $\Sigma \cup V$ defined by the grammar rules σ, α, β, ... respectively
- M_S, M_A, M_B, ... represent the **automata** recognizing R_S, R_A, R_B, ... respectively
- $\mathcal{M} = \{M_S,\ M_A,\ M_B,\ \dots\}$ is the **collection** of all automata *(the net)*
- $Q_A = \{0_A,\ \dots,q_A,\ \dots\}$ is the **state set** of automaton M_A, where 0_A is the only initial state and its final state set is $F_A \subseteq Q_A$
- the state set Q of a net $\mathcal{M}$ is the **union of all the states** of the component machines

$$Q = \bigcup_{M_A \in \mathcal{M}} Q_A$$

- the **names of the states are unique**, so that $q_A \in Q_A$ and $q_B \in Q_B$ are different for $A \neq B$
- the function δ represent the **transition** for each individual automaton
- for a state q_A, the symbol $R(M_A, q_A)$ *(or $R(q_A)$ for brevity)* denotes the **regular language** over alphabet $\Sigma \cup V$ recognized by M_A starting from state q_A. For the initial state, $R(0_A) \equiv R_A$
- the set of **terminal strings** generated along the path of a machine, starting from state q_0 and reaching a final state is called $L(q)$

Furthermore, every machine M_I must not have any arc entering the initial state 0_I; this requirement ensures that the initial state is not visited twice. If this situation occurs in a machine, then a new initial state $0'_I$ is added, and an arc is added from $0'_I$ to 0_I.
Automata satisfying this condition are called **normalized** or with initial state non recirculating *(or non reentrant)*; this condition does not forbid the existence of initial states that are also final.

5.3.1 Notable sets

5.3.1.1 Initials

> **Definition 5.6** (Set of initials)**.** The **set of initials** $\mathrm{Ini} \subseteq \Sigma$ of a state q_A is defined as:
>
> $$\mathrm{Ini}(q_A) = \mathrm{Ini}\left(L\left(q_A\right)\right) = \{a \in \Sigma \mid a\Sigma^* \cap L\left(q_A\right) \neq \emptyset\}$$

Notes

- Ini may **not** contain the null string ε
- A set $\mathrm{Ini}(q_A)$ is **empty** if and only if the empty string is the only generated string from q_A: $L\left(q_A\right) = \{\varepsilon\}$

Let symbol a be **terminal**, symbols A and B be **non-terminals**, and q_A and r_A be **states of automaton** M_A. Set Ini is computed by applying the following logical clauses until a fixed point is reached; the relation $a \in \mathrm{Ini}(q_A)$ holds if one or more of the following clauses is true:

1. $\exists$ arc $q_A \xrightarrow{a} r_A$
2. $\exists$ arc $q_A \xrightarrow{B} r_A \wedge a \in \mathrm{Ini}(0_B)$, excluding the case $0_A \xrightarrow{A} r_A$
3. $\exists$ arc $q_A \xrightarrow{B} r_A \wedge L(0_B)$ is nullable $\wedge\ a \in \mathrm{Ini}(r_A)$

5.3.1.2 Candidates

> **Definition 5.7** (Candidate)**.** A pair $\langle state, token \rangle$ is a **candidate** *(or item)*: more precisely, a candidate is a pair
>
> $$\langle q_A, a \rangle \in Q \times (\Sigma \cup \{\dashv\})$$

The intended meaning is that token a is a **legal lookahead** for the current state q_A of machine M_A. When the parsing operation begins, the initial state of the axiomatic machine M_S is encoded by the candidate $\langle 0_S, \dashv \rangle$: it means that the end of text character $\dashv$ is expected when the entire input is reduced to the axiom S.

In order to calculate the candidates for a given grammar or machine net, a function named **closure** is defined.

5.3.1.3 Closure

> **Definition 5.8** (Closure)**.** Let C be a set of candidates. The **closure** of C *(written as Closure(C))* is the function defined by applying the following clause:
>
> $$\mathrm{Closure}(C) = \{C\}$$
>
> $$\langle 0_B, b \rangle \in \mathrm{Closure}(C) \ \text{if} \ \begin{cases} \exists \ \text{candidate} \ \langle q, a \rangle \in C \ \wedge \\ \exists \ \text{arc} \ q \xrightarrow{B} r \in \mathcal{M} \ \wedge \\ b \in \mathrm{Ini}\left(L\left(r\right) \cdot a\right) \end{cases}$$
>
> until a fixed point is reached.

The function closure computes the **set of the machine states that are reached from a given state** q through one or more *invocations*, without any intervening state transition. For each reachable machine M_B, represented by the initial state 0_B, the function returns any input character b that can legally occur when the machine terminates; such a character is part of the lookahead set.

When the clause terminates, the closure of C contains a set of candidates, with some of them possibly associated with the same state q as follows:

$$\{\langle q, a_1 \rangle, \ \ldots, \langle q, a_k \rangle\} = \langle q, \{a_1, \ \ldots, a_k\} \rangle$$

The collection $\{a_1, a_2, \ldots, a_k\}$ is called the **lookahead set** of the state q in the closure of C; by construction, it's never empty *(it can always contain the end of text character $\dashv$)*.
For brevity, the singleton lookahead set $\langle q, \{b\} \rangle$ is written $\langle q, b \rangle$.

5.4 Bottom-up Parsing

The formal conditions that allow a Bottom-up to be **deterministic** are named $LR(k)$, where the parameter $k \geq 0$ represents the number of consecutive characters inspected by the parser to decide the next action deterministically. Since the *EBNF* grammars are normally considered and $k = 1$ is a common value, the condition is referred to as $ELR(1)$. The language family accepted by the parsers of type $LR(1)$ *(or $ELR(1)$)* is exactly the family *DET* of deterministic context-free languages.
The $ELR(1)$ parsers implement a deterministic automaton equipped with a pushdown stack and with a set of internal states *(called macro states or m-states)* that consist of a set of candidates. The automaton performs a series of moves of two types:

- **shift move**, which reads an incoming character *(token)* and applies a state transition function to compute the next m-state; the two of them are then pushed in the stack
- **reduction move**, applied as soon as the sequence of topmost stack symbols matches the recognizing path in a machine M_A, under the condition that the current character is in the current lookahead set
    - $\rightarrow$ the fragment of the **syntax tree** computed so far **grows**
    - $\rightarrow$ in order to update the stack, the topmost part *(handle)* is **popped**
    - $\rightarrow$ if more than one reduction can be chosen, a **reduction-reduction conflict** occurs

The *PDA* accepts the input string if the last move reduces the stack to the initial configuration and the input has been entirely scanned. The latter condition can be expressed by saying that the special end marker character $\dashv$ is the current input token.

The conditions for **determinism** are:

1. in every parser configuration, if **a shift move is permitted** then **a reduction move is impossible**
2. in every configuration, **at most one reduction move is possible**

5.4.1 Multiple Transition Property and Convergence

> **Definition 5.9** (Multiple Transition Property)**.** A pilot m-state I has the multiple transition property *(MTP)* if it includes two candidates $\langle q, \pi \rangle$ and $\langle r, \rho \rangle$ with $q \neq r$, such that for some grammar symbol $\oplus \in \Sigma$ both transitions $\delta(q, \oplus)$ and $\delta(r, \oplus)$ are defined.

> **Definition 5.10** (Convergent m-state)**.** The m-state I and the transition $\theta(I, \oplus)$ *(with $\oplus \in \Sigma$)* are called **convergent** if it holds $\delta(q, \oplus) = \delta(r, \oplus)$. This condition may affect determinism.

In other words, the *MTP* occurs when two states within an m-state have outgoing arcs labelled with the same grammar symbol.
A **convergent transition** has a **convergence conflict** if $\pi \cap \rho \neq \emptyset$, meaning the two lookahead sets of the candidates are not disjoint.

5.4.1.1 Single Transition Property

> **Definition 5.11** (Single Transition Property)**.** The same pilot m-state I has the **single transition property** *(STP)* if it does not have the *MTP*.

For a given grammar, the *STP* condition has 2 important consequences:

1. the **range of parsing choices** at any time is restricted to 1
2. the presence of **convergent arcs** in the pilot is automatically **excluded**

5.4.2 *ELR*(1) condition

Since two or more paths can lead to the same final state, two or more reductions may be applied when the parsers enter m-state I; to choose the correct reduction, the parser has to store additional information on the stack. The next conditions *(5.12 and 5.13)* ensure that all steps are deterministic.

Definition 5.12 (*ELR*(1) condition). An *EBNF* grammar or its machine net meets the condition **ELR**(1) if the corresponding pilot satisfies the following conditions:

- Every m-state I satisfies the next two clauses:
    1. no **shift-reduce conflict** occurs: for all the candidates $\langle q, \pi \rangle \in I$ such state q is final and for all the arcs $I \xrightarrow{a} I'$ that go out from I with terminal label a, it must hold $a \notin \pi$
    2. no **reduce-reduce conflict** occurs: for all the candidates $\langle q, \pi \rangle, \langle r, \rho \rangle \in I$ such that states q and r are final, it must hold $\pi \cap \rho = \emptyset$
- No transition in the pilot graph has a **convergence conflict**

Definition 5.13 (*LR*(1) contition). If the grammar is purely *BNF*, then the previous condition in referred to as *LR*(1) instead of *ELR*(1); conflicts never occur in the *BNF* grammars.

5.4.3 Construction of *ELR*(1) parsers

Given an *EBNF* grammar, an *ELR*(1) parser can be built by the following 3 steps:

1. From the automaton net, a deterministic *FSA (called pilot automaton)* is built
    - a pilot state *(named macro-state or m-state)* includes a non empty set of *candidates*
    - *see Paragraph 5.3.1.2*
2. The pilot is examined to check the conditions for deterministic parsing by inspecting the components of each m-state; the three types of conflicts are:
    - A. **shift-reduce**, when both a shift and a reduce move are possible with the same configuration
    - B. **reduce-reduce**, when two or more reductions are possible
    - C. **convergence**, when two different parser computations that share a candidate lead to the same m-state
3. If the previous test is passed, then the deterministic *PDA* is built; the pilot *FSA* must be enriched with features for pointer management, needed to handle stacks of unbounded length
4. The *PDA* can be encoded in a programming language

Definition 5.14 (Pilot automaton). The **pilot automaton** $\mathcal{P}$ is the 5-tuple defined by:

1. The **set** R of m-states
2. The **pilot alphabet** as the union $\Sigma \cup V$ of the terminal and non-terminal symbols, named grammar symbols
3. The **initial m-state** $I_0 = \text{Closure}\,(\langle 0_S, \dashv \rangle)$ *(thanks to Definition 5.8)*
4. The **m-state set** $R = \{I_0, I_1, \dots\}$ and the state-transition function $\theta : R \times \Sigma \cup V \to R$ are computed starting from I_0 as next specified

---

Let $\langle p_A, \rho \rangle$, $p_A \in Q$, $\rho \subseteq \Sigma \cup \{\dashv\}$, be a candidate and be $\oplus$ be a grammar symbol. Then the **terminal shift** (or **non-terminal shift**, *according to the nature of $\oplus$*) under $\oplus$, is:

$$
\begin{cases}
\theta\,(\langle p_A, \rho \rangle, \oplus) = \langle q_A, \rho \rangle & \text{if the arc } p_A \xRightarrow{\oplus} q_A \text{ exists} \\
\emptyset & \text{otherwise}
\end{cases}
$$

For a set C of candidates, the shift under a symbol $\oplus$ is the union of the shifts of the candidates in C:

$$
\theta(C, \oplus) = \bigcup_{\forall \text{ candidate } \gamma \in C} \theta(\gamma, \oplus)
$$

Finally, the algorithm to build the state set R and the state transition function θ is shown in Code 3.

```
R_1 := { I_0 } // prepare the initial m-state I_0
do
  R := R_1 // update the m-state set R
  for each m-state I ∈ R and symbol ⊕ ∈ Σ ∪ V do
    I_1 := closure(θ(I, ⊕)) // compute the closure of the shift under ⊕
    if I_1 != ∅ then // if the shift is not empty
        add arc I →⊕ I_1 to the graph of θ
      if I_1 ∉ R_1 then // if the new m-state is not already in R_1
        add I_1 to R_1
      end
    end
  end
while R ≠ R_1 // repeat until R_1 does not change
```

Code 3: Algorithm to build $ELR(1)$ pilot graph

5.4.3.1 Base, Closure and Kernel of m-state

For an m-state I, the set of candidates is categorized in 2 classes, named **base** and **closure**. An additional set, **kernel** also exists.

Definition 5.15 (Base set). The **base** includes the non-initial candidates:

$$I_{\text{base}} = \{\langle q, \pi \rangle \in I \mid q \text{ is not an initial state }\}$$

Clearly, for the m-state I' computed in the algorithm, the base I'_{base} coincides with the pairs computed by $\theta(I, \oplus)$.

Definition 5.16 (Closure set). The **closure** contains the candidates of I not included in the base:

$$I_{\text{closure}} = I \setminus I_{\text{base}} = \{\langle q, \pi \rangle \in I \mid q \text{ is an initial state }\}$$

By definition, the base of the initial m-state I_0 is empty and all other m-states have a non-empty base, while their closure may be empty.

Definition 5.17 (Kernel set). The **kernel** of an m-state I is the projection on the first component of every candidate:

$$I_{\text{kernel}} = \{q \in Q \mid \langle q, \pi \rangle \in I\}$$

Two m-states with the same kernel *(differing just for some lookahead set)* are called **kernel-equivalent**.

5.4.3.2 Pilot Graph

The **Pilot Graph** represents graphically the transitions of the pilot automaton. Each m-state is split by a double line into the **base** *(upper part)* and the **closure** *(lower part)*; either part can be missing. The lookahead items are grouped by the state they belong to; the final states are marked with a circle.
An example of pilot graph is shown in Figure 8.

(a) State of a Pilot graph

(b) State of a Pilot graph without closure

(c) State of a Pilot graph without base
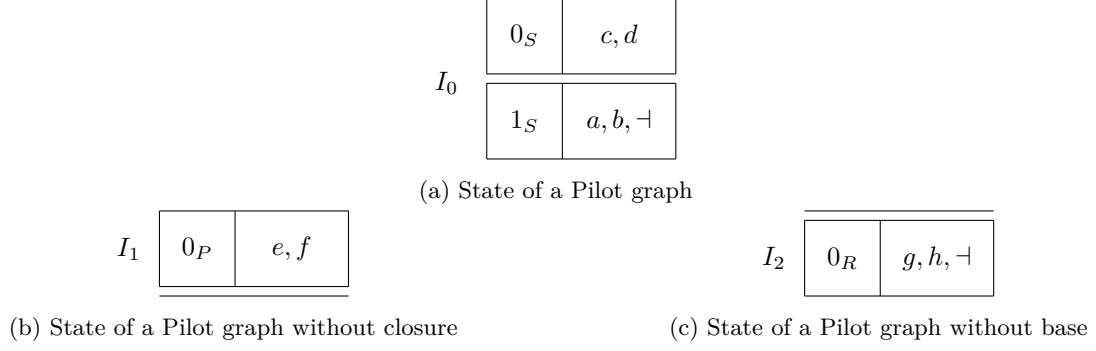
Figure 8: Example states of a Pilot graph

5.4.3.3 Stack Content

Since for any given net $\mathcal{M}$, there are finitely many different candidates, the number of m-states is bounded and the number of candidates in any m-state is always lower than $|Q|$, by assuming that all the candidates with the same state are *merged*. Moreover, it's safe to assume that the candidates in an m-state are ordered, so that each one occurs at a position *(or offset)* which will be referred to by other m-states using a pointer called candidate identifier *(or cid)*.

The stack elements are of two types, interleaving each other:

1. **grammar symbols**, elements of $\Sigma \cup V$
2. **stack m-states** *(sms)*, denoted by letter J and containing ordered set of 3-tuples of the form:

$$\langle q_A, \pi, cid \rangle \text{ where } \begin{cases} q_A \in Q & \text{state} \\ \pi \subseteq \Sigma \cup \{\dashv\} & \text{lookahead set} \\ 1 \leq cid \leq |Q| \text{ or } cid = \bot & \text{candidate identifier} \end{cases}$$

where $\bot$ represent the NIL value for the cid pointer. For readability, a cid value will be prefixed by the marker character $\sharp$.

A function μ maps the set of *sms* to the set of m-states defined by $\mu(J) = I$ if and only if deleting the third field *(the cid)* from the stack candidates of J, the result is exactly the candidates of I.

5.4.3.4 Bottom-up Parser Construction

Let the current stack be $J[0] \, a_1 \, J[1] \, a_2 \, \ldots \, a_k \, J[k]$, where a_i is a grammar symbol and $J[k]$ a *sms*.
The steps of the algorithm are shown in the following Subparagraphs.

Initialization The initial stack just contains the *sms*:

$$J_0 = \{s \mid s = \langle q, \pi, \bot \rangle \text{ for every candidate } \langle q, \pi \rangle \in I_0\}$$

thus $\mu(J_0) = I_0$

Shift Let the top *sms* be J and $I = \mu(J)$, let the current token be $a \in \Sigma$ and assume that the m-state I has transition $\theta(I, a) = I'$. The shift move performs the following two operations:

1. push token a on stack and get the next token
2. push on stack the *sms* J' computed as follows:

$$J' = \left\{ \langle q'_A, \rho, \sharp j \rangle \mid \langle q_a, \rho, \sharp j \rangle \text{ is at position } i \text{ in } J \wedge q_A \xrightarrow{a} q'_A \in \delta \right\} \cup \{ \langle 0_B, \sigma, \bot \rangle \mid \langle 0_B, \sigma \rangle \in I'_{\text{Closure}} \}$$

thus $\mu(J') = I'$

Reduction of non-initial candidates The m-states corresponding to the current stack are:

$$I[i] = \mu\left(J[i]\right), \, 1 \leq i \leq k$$

while the current token is a; assume that by inspecting $I[k]$ the pilot chooses the reduction candidate $c = \langle q_a, \pi \rangle \in I[k]$, where q_A is a final and non initial state. Let $t_k = \langle q_A, \rho, \sharp i_k \rangle \in J[k]$ be the only candidate such that the current token a satisfies $a \in \rho$. From i_k a *cid* search starts, which links stack candidate t_k to a stack candidate $t_{k-1} = \langle p_A, \rho, \sharp i_{k-1} \rangle \in J[k-1]$ until a stack candidate $t_h \in J[h]$ is reached such that has a NIL *cid* and therefore its state is initial.

The reduction move does as follows:

1. **grows the syntax forest** by applying the following reduction

$$a_{h+1}\, a_{h+2}\, \ldots\, a_k \rightsquigarrow A$$

2. **pops** the following stack symbol

$$J[k]\, a_k\, J[k-1]\, a_{k-1}\, \ldots\, J[h+1]\, a_{h+1}$$

3. executes the non-terminal **shift** *(see below)*

Reduction of initial candidate The reduction of initial candidates differs from the preceding case in the state of the chosen candidate $c = \langle 0_A, \pi, \bot \rangle$ which is both final and initial; the parser move grows the syntax forest by the reduction $\varepsilon \rightsquigarrow A$ and performs the non-terminal shift move corresponding to $\theta(I[k], A) = I'$.

Nonterminal shift The non-terminal shift move is the same as the shift move, but the shifted token is a non-terminal $A \in V$. The only difference is that the parser does not read the next input token at line (2) of the shift move.

Acceptance The parser accepts and halts when the stack is J_0, the move is the non-terminal shift defined by $\theta(I_0, S)$ and the current token is $\dashv$.

5.4.4   Time complexity

While analysing a string x of length $n = |x|$, the number of elements in the stack is $\mathcal{O}(n + 1) = \mathcal{O}(n)$. The number of total moves performed by the *PDA* is the sum of:

1. number of **terminal shift** moves: n_T
2. number of **non-terminal shift** moves: n_N
3. number of **reductions**: n_R

Since $n_T = n, n_N = n_R$ *(for each reduction there's one non-terminal shift)*, the total number of moves is $n_T + n_N + n_R = n + \cdot n_R$.
Furthermore:

1. the number of reductions with one or more terminals is lower than n
2. the number of reductions of type $A \rightarrow \varepsilon$ is lower than n
3. the number of reductions without any terminals is lower than n

Thus the time complexity is $\mathcal{O}(n)$.

5.4.5   Implementation via Vector Stack

The *PDA* can be implemented using a vector stack: this solution is potentially faster and offers the ability to access the items in the stack via their integer index. As seen before *(Paragraph 5.4.3.3)*, the stack contains elements of two alternating types: grammar symbols and vector stack m-states *(or vsms)*.
A *vsms*, denoted by J, is a set of pairs, named vector-stack candidates in form $\langle q_A, \texttt{elemid} \rangle$ differing from the previous definition of stack candidates because the second component is a positive integer named element identifier *(elemid)* instead of the candidate identifier *(cid)*; furthermore, the set is now ordered. Each `elemid` points back to the vector-stack element containing the initial state of the current machine so that when a reduction move is performed, the length of the string to be reduced (and the reduction handle) can be obtained directly without inspecting the stack elements below the top one.

Notable values

- In a closure item, the value of `elemid` is the current stack element index
- In a base item, the value of `elemid` is the same as the item before

The surjective mapping from vector-stack m-states to pilot m-states is denoted μ as before.

5.5 Top-down Parsing

Top-down parsing is a strategy in which the parser tries to match the input string with the grammar rules starting from the root of the grammar *(the axiom)*, expanding each *RP* via recursive descent. This strategy requires making a hypothesis about the parse tree structure and then verifying if the basic rules of the grammar are satisfied.

The analysis procedure is driven by the machine network, without the need for a pilot graph, as the machine itself can work as a pilot for the *PDA*; however, the final net needs annotations to choose the correct transition when in doubt. The syntax analyser can recognize early which rule it should apply to a phrase, as soon as it finds the leftmost character of the rule.

5.5.1 *ELL*(1) Condition

A simple yet flexible top-down parsing method, called *ELL*(1), can be applied to *ELR*(1) grammars with additional conditions. A machine net $\mathcal{M}$ meets the *ELL*(1) condition if the following conditions are satisfied:

1. there are **no left recursive** derivations *(Section 3.3.1)*
2. the net meets the *ELR*(1) condition *(Section 5.4.2)*; it must not have either **shift-reduce**, **reduce-reduce** or **convergence conflicts**
3. the net has **single transition property** *(STP) (Definition 5.11)* for each state

Notes

- condition (1) excludes grammars with non immediate left recursion
- the requirements are not completely independent from each other: certain types of left-recursive grammars violate clause (2) and (3)
- condition (1) could be further restricted since a grammar with left recursion can be converted to a grammar without it.

5.5.2 Step by Step derivation of *ELL*(1) Parsers

Starting from bottom-up parsers, a *ELL*(1) parser can be derived step by step, via a series of transformations. The *STP* condition permits to reduce greatly the number of m-states, down to the number of net states; furthermore, convergent arcs disappear and consequently, the chain of stack pointers can be eliminated from the parser.

By requiring that the grammar is not left-recursive, a parser via a **parser control flow graph** *(or PCFG)* is built. The latter is isomorphic to the machine net and can be used to generate the parser.

The final result is a top-down predictive parser able to build the syntax tree as it proceeds.

5.5.2.1 Pilot Compaction

Kernel identical m-states can be merged into a single m-state, obtaining a smaller pilot behaving exactly as the original one. This operation does not work under *ELR*(1) condition but it's correct under the *STP* hypothesis. The following algorithm defines the merging operation, which coalesces two kernel-identical m-states I_1, I_2 into a single m-state.

Operation `merge`(I_1, I_2)

1. replace m-states I_1, I_2 with a new kernel identical m-state $I_{1,2}$, having as the lookahead set the union of the corresponding ones in the merged m-states

$$\langle p, \pi \rangle \in I_{1,2} \iff \langle p, \pi_1 \rangle \in I_1 \quad \text{and} \quad \langle p, \pi_2 \rangle \in I_2 \quad \text{and} \quad \pi = \pi_1 \cup \pi_2$$

2. m-state $I_{1,2}$ is the new target of all arcs in I_1 and I_2

$$I \xrightarrow{\oplus} I_{1,2} \iff I \xrightarrow{\oplus} I_1 \quad \text{or} \quad I \xrightarrow{\oplus} I_2$$

3. for each pair of arcs leaving I_1 and I_2 with the same label $\oplus$, the target m-states are merged

$$\text{if} \quad \theta(I_1, \oplus) \neq \theta(I_2, \oplus) \quad \text{then} \quad \text{call } \texttt{merge}(\theta(I_1, \oplus), \theta(I_2, \oplus))$$

The operation then terminates, producing a graph with fewer nodes. By applying `merge` to the members of every equivalence class, then new graph called **compact pilot** is built.
The compact pilot is denoted by $\mathcal{C}$.

Candidate Identifier   Thanks to the *STP* property, the algorithm shown in Paragraph 5.4.3.4 *(Bottom-up Parser Construction)* can be simplified to remove the need for *cid* and stack pointers: they were used to find the reach of a non-empty reduction move into the stack, popping elements until the chain reached the end *(the cid had value NIL)*.

Parser Control Flow Graph - *PCFG*   The pilot graph $\mathcal{C}$ can be made isomorphic to the original machine net $\mathcal{M}$, thus obtaining a graph called **parser control flow graph** *(PCFG)* representing the blueprint of the parser code.

Steps of the build

1. every m-node of $\mathcal{C}$ containing n candidates is split in n different nodes
2. the kernel-equivalent nodes are merged into one and the original lookahead sets are combined
3. new arcs *(named call arcs)* are added to the graph, representing the transfers of control from a machine to another
4. each call arc is labelled with a set of terminals *(named guide sets)*, which will determine the parser decision to transfer control to the target machine

Every node of the *PCFG*, denoted by $\mathcal{F}$, is identified and denoted by a state q of machine net $\mathcal{M}$. Every final node f_A additionally contains a set of π of terminals, named prospect set; such nodes are therefore associated to the pair $\langle A, \pi \rangle$.
The prospect set is the union of the lookahead sets π_i of evert candidate $\langle f_A, \pi_1 \rangle$ existing in the compact pilot graph $\mathcal{C}$ as follows:

$$\pi = \bigcup_{\forall \langle f_A, \pi_i \rangle \in \mathcal{C}} \pi_i$$

The arcs of pilot $\mathcal{F}$ are of two types, shift and call, represented respectively by the symbols $\rightarrow$ and $\dashrightarrow$:

1. there exists in $\mathcal{F}$ a shift arc $q_A \xrightarrow{\oplus} r_A$ with $\oplus$ either terminal or nonterminal if the same arc is in the machine M_A
2. there exists in $\mathcal{F}$ a call arc $q_A \dashrightarrow^{\gamma_1} 0_{A_1}$ where A_1 is a nonterminal possibly different from A, if arc $q_A \xrightarrow{A_1} r_A$ is in the machine M_A

  $\rightarrow$ necessary condition 1. m-state K of pilot $\mathcal{C}$ contains candidates $\langle q_A, \pi \rangle$ and $\langle 0_{A_1}, \rho \rangle$
  $\rightarrow$ necessary condition 2. the next m-state $\theta(K, A_1)$ of pilot $\mathcal{C}$ contains candidates $\langle r_A, \pi_{r_A} \rangle$

The call arc label is called guide set:

1. for every call arc $q_A \dashrightarrow^{\gamma_1} 0_{A_1}$ associated to a nonterminal shift arc $q_A \xrightarrow{A_1} r_A$, a terminal b is in the guide set γ_1, also written as $b \in \mathrm{Gui}(q_A \dashrightarrow 0_{A_1})$, if and only if one of the following conditions hold:

$$b \in \mathrm{Ini}\left(L(0_{A_1})\right) \tag{A}$$

$$A_1 \text{ is nullable and } b \in \mathrm{Ini}\left(L(r_A)\right) \tag{B}$$

$$A_1, L(r_A) \text{ are both nullable and } b \in \pi_{r_A} \tag{C}$$

$$\exists \text{ in } \mathcal{F} \text{ a call arc } 0_{A_1} \dashrightarrow^{\gamma_2} 0_{A_2} \text{ and } b \in \gamma_2 \tag{D}$$

2. for every terminal shift arc $p \xrightarrow{a} q$ with $a \in \Sigma$, the guide set γ is the singleton set $\{a\}$:

$$\mathrm{Gui}(p \xrightarrow{a} q) := \{a\} \tag{E}$$

3. for every arrow that tags a final node containing candidate $\langle f_A, \pi \rangle$ with f_A final:

$$\mathrm{Gui}(f_A \rightarrow) := \pi \tag{F}$$

Guide sets are normally represented as arc labels enclosed within braces.
Relations A, B, C are not recursive, and consider that:

1. In equation A, b is generated by M_{A_1} and called by M_A
2. In equation B, b is generated by M_A starting from state r_A
3. In equation C, b is a terminal following M_A

Relation D is recursive and traverses the net as far as the chain of call sites activated; it induces the set inclusion relation $\gamma_1 \supseteq \gamma_2$ between two any concatenated calls $q_A \dashrightarrow^{\gamma_1} 0_{A_1} \dashrightarrow^{\gamma_2} 0_{A_2}$.

Disjointness of guide sets

1. For every node q of the *PCFG* of a grammar G that satisfies the *ELL*(1) condition, the guide sets of any two arcs originating from q are disjoint
2. If the guide sets of a *PCFG* are disjoin, the machine net satisfies the *ELL*(1) condition

Statement (1) says that, for the same state, membership in different guide sets is exclusive; statement (2) says that, if the guide sets are disjoint, the machine net satisfies the *ELL*(1) condition.
This condition can also be checked directly on the *PCFG*, without needing to build the *ELR*(1) pilot *(Section 5.5.4)*.

5.5.2.2 Top-down Predictive Parser construction

There are two kinds of top-down predictive parsers: a **deterministic pushdown automaton** *(called DPDA)* or a **recursive syntactic procedures**; both are easily obtainable from the *PCFG*.
In this Paragraph, the focus will be on the former.

Characteristics of the *DPDA*

- The stack top identifies the state of the active machine
- The inner stack elements refer to the return states of suspended machines, in order of suspension
- It allows 4 move types:

  1. a scan move, associated with a terminal shift arc, reads the current token cc as the corresponding machine would do
  2. a call move, associated with a call arc, checks the enabling predicate, saves on the stack the return state and switches to the invoked machine without consuming token cc
  3. a return move triggered when the active machine enters a final state, the prospect set of which includes token cc
  4. a recognizing move, terminating the parsing process

5.5.3 Construction of the *ELL*(1) Parser

- The stack elements are the states of *PCFG* $\mathcal{F}$
- The stack is initialized with element $\langle 0_S \rangle$
- Let $\langle q_A \rangle$ be the top of stack element, meaning that the active machine M_A is in state q_A

The different move types are defined in the following Subparagraphs.

scan move if the shift arc $q_A \xrightarrow{cc} r_A$ exists, then scan the next token and replace the stack top by $\langle r_A \rangle$. The currently active machine does not change.

call move if there exists a call arc $q_A \dashrightarrow^{\gamma} 0_B$ such that $cc \in \gamma$, let $q_A \xrightarrow{B} r_A$ be the corresponding nonterminal shift arc; then pop, push element $\langle r_A \rangle$ and push element $\langle 0_B \rangle$.

return move if q_A is a final state and token cc is in the prospect set associated with q_A, then pop.

recognition move If M_A is the axiom machine, q_A is the final state and $cc = \dashv$, then accept and `halt`. In any other case, reject the string and `halt`.

5.5.3.1 Procedural interpretation

The top-down parsing algorithm can be seen as a sequence of procedures: each machine M in the net is a sub-procedure, corresponding to the set of nonterminals. A machine state is the same as a program point inside the procedure code, a nonterminal label on an arc is the same as a procedure call, the return state corresponds to the program point after the procedure call, and bifurcation states represent conditional statements. The analogy can go further: a machine invocation is a procedure call, and the guide set is the set of arguments passed to the procedure call.

All the arcs in the *PCFG*, excluding the nonterminal shifts, can be viewed as conditionals instructions: they are enabled if the current character cc is in the guide set $\mathrm{Gui}(a)$ of the arc a.
As such:

- a terminal shift arc labelled as $\{a\}$ is enabled by predicate $cc \in \{a\}$, simplified as $cc = a$
- a call arc labelled with set γ represents the procedure invocation conditioned by predicate $cc \in \gamma$
- a final node arrow labelled with set π represents a conditional return `from procedure` instruction to be executed if $cc = \pi$
- the remaining *PCFG* arcs are nonterminal shifts, representing unconditional `return from procedure` instructions

5.5.4 Direct construction of the *PCFG*

The *PCFG* can be directly built from the machine net, without having to build the *ELR*(1) pilot.
A set of recursive equations is used to compute the prospect and guide set of all the states and arcs of the *PCFG*; the equations are instructions to be executed iteratively to compute the sets. In order to compute the prospect sets of the final states, it's necessary to compute the prospect sets of all the other states, eventually discarding them.

Equations for the prospect sets

1. If the graph includes any nonterminal shift arc $q_i \xrightarrow{A} r_i$ *(therefore also the call arc $q_i \dashrightarrow 0_A$)*, then the prospect set π_{0_A} for the initial state 0_A of machine M_A is computed as:

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

2. if the graph includes any terminal or nonterminal shift arc $p_i \xrightarrow{X_i} q_i$, then the prospect set π_q of state q is computed as:

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

The two sets of rules apply in an exclusive way to disjoints sets of nodes because in a normalized machine no arc enters the initial state. To initialize the computation, the prospect state of the initial machine 0_S is initialized as:

$$\pi_{0_S} := \{ \dashrightarrow \}$$

All other sets are initialized as empty.

Equations for the guide sets

1. For each **call arc** $q_A \dashrightarrow 0_{A_1}$ associated with a **nonterminal shift arc** $q_A \xrightarrow{A_1} r_A$, such that possibly other call arcs $0_{A_1} \dashrightarrow 0_{B_i}$ depart from state 0_{A_1}, the guide set $\mathrm{Gui}(q_A \dashrightarrow 0_{A_1})$ is computed as:

$$\mathrm{Gui}(q_A \dashrightarrow 0_{A_1}) := \bigcup \begin{cases} \mathrm{Ini}\,(L(A_1)) \\ \textbf{if } \mathrm{Nullable}(A_1) \textbf{ then } \mathrm{Ini}\,(L(r_a)) \textbf{ else } \emptyset \\ \textbf{if } \mathrm{Nullable}(A_1) \wedge \mathrm{Nullable}\,(L(r_A)) \textbf{ then } \pi_{r_A} \textbf{ else } \emptyset \\ \cup_{0_{A_1} \dashrightarrow 0_{B_i}} \mathrm{Gui}(0_{A_1} \dashrightarrow 0_{B_i}) \end{cases}$$

2. For a **final state** $f_A \in F_A$, the guide set of the tagging dart equals the prospect set of the state:

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

3. For a **terminal shift arc** $q_A \xrightarrow{a} r_A$, $a \in \Sigma$, the guide set is the singleton set composed by the shifted terminal:

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

5.5.5 Increasing the Lookahead

In order to pragmatically obtain a deterministic top-down parser when the grammar is not $ELL(1)$ is to increase the lookahead of the current character and examine the following ones; this is often a sufficient condition to obtain a deterministic parser.

A grammar has the $ELL(k)$ property if there exists an integer $k \geq 1$ such that, for every net machine and for every state, at most one choice among the outgoing arrows is compatible with the characters that may occur ahead of the current character, at distance less than or equal to k.

Formally, the elements of $ELL(k)$ guide sets are the strings of length up to k terminals.

5.6 Syntax Analysis of Nondeterministic Grammars

The **Earley** method *(also called **tabular** method)* deals with any grammar type, including ambiguous and non deterministic ones. It works by building in parallel all the possible derivations of the string prefix scanned so far, without having to implement a lookahead; furthermore, it does not need a stack since a vector of sets is used to store the current state of the parsing process.

While analysing a string $x = x_1 x_2 \ldots x_n$ or $x = \varepsilon$ of length $|x| = n \geq 0$, the algorithm uses a vector $E\,[0 \ldots n]$ of $n + 1$ elements. Every element $E[i]$ is a set of **pairs** $\langle q_\oplus, j \rangle$, where:

- $q_\oplus$ is a state of machine $M_\oplus$
- j is an **integer pointer** that indicates element $E[i]$, with $0 \leq i \leq j \leq n$, preceding or corresponding to $E[j]$ and containing a **pair** $\langle 0_\oplus, j \rangle$
  - $0_\oplus$ belongs to the same machine as $q_\oplus$
  - j marks the position in the input string x from where the current derivation of $\oplus$ started, and it's represented by $\uparrow$
  - if $j = i$, the string is empty ε

- an example of this representation is

$$x_1 \ldots x_j \uparrow \underbrace{x_{j+1} \ldots x_i}_{\oplus} \ldots x_n$$

A pair has the form $\langle q_\oplus, j \rangle$ and it's called:

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

5.6.1 Earley's Method

Initially, the Earley vector is initialized by setting all element $E[1], \ldots, E[n]$ to the empty set, *($E[i] = \emptyset \, \forall 1 \leq i < n$)* and the first element $E[0]$ is set to the set containing the **initial pair** $\langle 0_S, 0 \rangle$; in this pair, the state 0_S is the initial state of the machine M_S of the start symbol S, and the integer pointer 0 indicates the position in the input string from where the current derivation of S started. As parsing proceeds, when the current character is x_i, the current element $E[i]$ will be filled with one or more pairs; the final Earley vector will contain all the possible derivations of the input string.

Three different operations can be applied to the current element of the vector $E[i]$: **closure**, **terminal shift** and **nonterminal shift**. They resemble the operations of the similar *ELR*(1) parser *(Paragraph 5.4.3.4)* and are presented in the next subparagraphs.

Closure   Applies to a pair with a state from where an **arc** with a **nonterminal label** $\oplus$ **originates**. Suppose that the pair $\langle p, j \rangle$ is in the element $E[i]$ and that the net has an arc $p \xrightarrow{\oplus} q$ with a nonterminal label $\oplus$ and a *(non relevant)* destination state q. The operation **adds a new pair** $\langle 0_\oplus, i \rangle$ to the same element $E[i]$: the **state** of this pair is the initial one $0_\oplus$ of machine $M_\oplus$ of that nonterminal $\oplus$, and the **pointer** has value i, which means that the pair is created at step i starting from a pair already in the element $E[i]$.

The effect of this operation is to add the current element $E[i]$ to all the pairs with the initial states of the machines that can recognize a substring starting from the next character x_{i+1} and ending at the current character x_i.

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

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

The operation **will certainly find at least one such pair** and the nonterminal shift applies to it. Then the operation writes the pair $\langle q, l \rangle$ into $E[i]$; if more than one pair is found, the operation is applied to all of them.

5.6.1.1 Earley's Algorithm

The algorithm for *EBNF* grammars makes use of two procedures, called `completion` and `terminalshift`. The input string is denoted by $x = x_1 x_2 \ldots x_n$, where $|x| = n \geq 0$ *(if $n = 0$, then $x = \varepsilon$)*.
The codes for the two procedures are presented in Code 4 and Code 5, respectively.

```
completion(E, i) // 0 ≤ i ≤ n
  do
    for each pair ⟨p, j⟩ ∈ E[i] and ⊕, q ∈ V, Q such that p →⊕ q do
      add pair ⟨0⊕, j⟩ to E[i]
    end
```

```
      for each pair ⟨f,j⟩ ∈ E[i] and ⊕,q ∈ V,Q such that f ∈ F_⊕ do
        for each pair ⟨p,l⟩ ∈ E[j] and q ∈ Q such that p⊕→q do
          add pair ⟨q,l⟩ to E[i]
        end
      end
  while any pair is added
```

Code 4: `completion` procedure

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

```
terminalshift(E, i, x_i) // 1 ≤ i ≤ n
  for each pair ⟨p,j⟩ ∈ E[i-1] and q ∈ Q such that p^{x_i}→q do
    add pair ⟨q,j⟩ to E[i]
  end
```

Code 5: `terminalshift` procedure

The `terminalshift` procedure adds to the current vector element $E[i]$ all the pairs that can be reached from the pairs in $E[i-1]$ by a terminal shift, scanning token x_i, $1 \leq 1 \leq n$. It may fail to add any pair to the element, that will remain empty; a nonterminal that exclusively generates the empty string ε never undergoes a terminal shift. Finally, notice that the procedure works correctly even when the element $E[i]$, or its predecessor $E[i-1]$, is empty.
The full algorithm for the Earley parser is presented in Code 6.

```
// analyse terminal string x for acceptance
// define the Earley vector E[0..n] of sets of pairs

E[0] := { ⟨0_S,0⟩ } // initial pair
for i := 1 to n do
  E[i] := ∅ // initialize all elements of E
end
completion(E, 0) // apply closure to initial pair
i := 1
while (i <= n and E[i-1] != ∅) do
  // while the vector is not finished and the previous element is not empty
  terminalshift(E, i, x_i) // put into the current element E[i]
  completion(E, i) // complete the current element E[i]
  i := i + 1
end
```

Code 6: Earley's Algorithm

The algorithm can be summarized into the following steps:

1. the initial pair $\langle 0_S, 0 \rangle$ is added to the first element $E[0]$ of the vector.
2. the elements $E[1]$ to $E[n]$ *(if present)* are initialized to the empty set
3. $E[0]$ is completed
4. if $n \geq 1$ (if the string x is not empty), the algorithm puts pairs in the current element $E[i]$ through `terminalshift` and finishes element $E[i]$ through `completion`.

$\rightarrow$ if `terminalshift` fails to add any pair to $E[i]$, the element remains empty

5. the loop iterates as far as the last element $E[n]$, terminating when the vector is finished or the previous element $E[i-1]$ is empty

> **Property 5.2** (Acceptance condition). When the Earley algorithm terminates, the string x is accepted if and only if the last element $E[n]$ of vector E contains a final axiomatic pair $\langle f_S, 0 \rangle$, with $f_S \in F_S$

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

1. A vector element $E[i]$ contains several pairs $\langle q, j \rangle$ that are linearly limited by i, as the number of states in the machine net is constant and $j \leq i$. As such, the number of pairs in $E[i]$ is bounded by n:

$$|E(i)| = \mathcal{O}(n)$$

2. For a pair $\langle p, j \rangle$ checked in the element $E[i-1]$, the terminal shift operation adds one pair to $E[i]$. As such, for the whole $E[i-1]$, the `terminalshift` operation needs no more than n steps:

$$\texttt{terminalshift} = \mathcal{O}(n)$$

3. The `completion` procedure iterates the operations of closure and nonterminal shift as long as they can add some new pair. Two operations can be performed on the whole set $E[i]$:

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

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

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

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

As such, the following property holds:

> **Property 5.3** (Complexity of Earley's Algorithm). The asymptotic time complexity of the Earley algorithm in the worst case is $\mathcal{O}(n^3)$, where n is the length of the string analysed.

5.6.2 Syntax Tree Construction

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

- nonterminal $\oplus \in V$, root of the tree to be built
- state f, final for the machine $M_\oplus \in \mathcal{M}$

$\rightarrow$ f is the end of the computation path in $M_\oplus$ that corresponds to analysing the substring generated by $\oplus$

- two non negative integers i and j

  $\rightarrow$ i and j always respect the condition $0 \le i \le j \le n$

  $\rightarrow$ they respectively represent the start and end of the substring generated by $\oplus$

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

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

To build the tree of string x with root node S, procedure `buildtree` is called with parameters $(S, f_S, 0, n)$; then it builds recursively all the subtrees and will assemble them in the final tree. The code is shown in Code 7.

```
// ⊕ is a nonterminal, f is a final state f of M⊕, i and j are integers
// return as a parenthesized string the syntax tree rooted at node S
// node ⊕ will contain a list C of terminal and nonterminals child nodes
// either list C will remain empty, or it will be filled from right to left
buildtree(⊕, f, i, j)
  C := ε // set to empty the list C of child nodes of ⊕
  q := f // set to f the state q in machine M⊕
  k := i // set to i the index k of vector E
  // walk back the sequence of terminals and nonterminals in M⊕
  while q != 0⊕ do // q is not initial
    // check if node ⊕ has terminal x_k as its current child leaf
    if ∃ h = k - 1 and ∃ p ∈ Q⊕ such that ⟨ p, j ⟩ ∈ E[h]
      and net has p --x_k--> q then:
        C := C ∪ x_k // add x_k to the list C of child nodes of ⊕
    end
    // check if node ⊕ has nonterminal Y as its current child leaf
    if ∃ Y ∈ V and ∃ e ∈ F_Y and ∃ h, j (j <= h <= k <= i)
      and ∃ p ∈ Q⊕ such that
        (⟨ p, j ⟩ ∈ E[k] and ⟨ e, h ⟩ ∈ E[h] and net has p --Y--> q)
      then:
        // recursively built the subtree of node Y
        // concatenate the result to the list C of child nodes of ⊕
        C := C ∪ buildtree(Y, q, k, j)
    end
  q := p // shift the current state q to the previous state p
  k := h // shift the current index k to the previous index h
  end

  return (C)x // return the parenthesized string of the tree rooted at ⊕
```

Code 7: `buildtree` procedure

Essentially, `buildtree` walks back on a computation path in machine $M_\oplus$ and jointly scans back the Earley vector E from $E[n]$ to $E[0]$; during the walk, it recovers the terminal and nonterminal shift operations to identify the **children of the same node** $\oplus$. In this way, the procedure reconstructs in reverse order the shift operations performed by the Earley parser.

The `while` loop runs zero or more times, recovering .**one shift per iteration** The **first** condition in the loop recovers a **terminal shift** appending the related leaf to the tree, while the **second** one recovers a **nonterminal shift** and recursively calls itself to build the subtree of the related nonterminal node. State e is final for machine M_Y, and inequality $0 \le h \le k \le i$ is guaranteed by the definition of the Earley vector E. If the parent

nonterminal $\oplus$ immediately generates the **empty string** *(as there exists a rule $\oplus \to \varepsilon$)*, the **leaf ε is the only child** and the loop does not run again.

Function `buildtree` uses two local variables in the `while` loop the current state q of the machine $M_\oplus$ and the current index k of the Earley vector element $E[k]$, both updated at each iteration: initially, q is **final**; at the end of the algorithm, q is the **initial** state $0_\oplus$ of the machine $M_\oplus$. At each iteration, the current state q is shifted to the previous state p and the current index k is shifted from i to j, through jumps of different lengths. Sometimes k may stay in the same position: this happens if and only if the function processes a series of nonterminals that end up generating the **empty string** ε.

The two `if` conditions are **mutually exclusive** if the grammar is **not ambiguous**: the first one is true if the child is a leaf; the other is true if the child has its own subtree.

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

The total complexity can be estimated as follows:

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

$$\mathcal{O}(n) \times \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(n)$$

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

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

Finally, since the total number of terminal plus nonterminal shifts to be recovered *((3) and (4))* is linearly bounded by the numbers of nodes to be built, the total complexity of the algorithm is:

$$((3) \ + \ (4)) \times \ \# \ of \ nodes \ = \left(\mathcal{O}(n) + \mathcal{O}\left(n^2\right)\right) \times \mathcal{O}(n) = \mathcal{O}\left(n^3\right)$$

Computational complexity reduction via Earley vector ordering   Since the `buildtree` procedure does not write the Early vector, it's possible to reduce its complexity by reordering the vector E in a way that the `while` loop runs fewer times.

Suppose that each element $E[k]$ is ranked according to the value of its pointer: this operation is done in $(n+1)\mathcal{O}\left(n\log(n)\right) = \mathcal{O}\left(n^2\log(n)\right)$.

Now the second `if` statement requires a time $\mathcal{O}(n)$ to find a pair with final state e and pointer h in the stack, while searching the related pair with a fixed pointer j takes $\mathcal{O}(n)$ time. Similarly the first `if` statement will only require a time $\mathcal{O}\left(\log(n)\right)$.

The total time complexity of the algorithm is:

$$\begin{aligned}
E \ sorting \ &+ \ ((3) \ + \ (4)) \times \ \# \ of \ nodes \\
&= \mathcal{O}\left(n^2\log(n)\right) + \left(\mathcal{O}(\log(n)) + \mathcal{O}\left(n\log(n)\right)\right) \times \mathcal{O}(n) \\
&= \mathcal{O}\left(n^2\log(n)\right)
\end{aligned}$$

Optimization via lookahead   The items in the sets $E[k]$, $\forall\,k$ of the Earley vector can be extended by including a lookahead, computed in the same way as $ELR(1)$ parsers: by siding a lookahead each time, the Earley algorithm avoids putting into each set the items that correspond to choices that cannot succeed.

This technique may cause an increase in the number of items in the vector itself for some grammars, and as such its use is not always beneficial.

Application of the algorithm to ambiguous grammars Ambiguous grammars deserve interest in the processing of natural languages: the difficult part is representing all the possible syntax trees related to a string, the number of which can grow exponentially with respect to its length.

This can be done by using a **Shared Packed Parse Forest** *(SPPF)*, a graph type more general than the tree built that still takes a worst-case cubic time for building.

6 Syntax and Semantic translations

Generally speaking, a **translation** is a function *(or a mapping)* from a *source language* to a *target language*. Two approaches are normally used: via **coupled grammars** or via a **transducer** *(an automaton similar to an acceptor with the ability to produce output)*.

Such methods can be called **purely syntactic translations**: they extend and complete the language definition and the parsing method studied so far; however, they are not able to provide a semantic interpretation *(the meaning)* of the input string. An opposing method is called **attribute grammar model**, and it takes advantage of the syntactic modularity of grammar rules.

The difference between **syntax** and **semantics** is that the former is concerned with the **form** of a sentence, while the latter is concerned with its **meaning**. In computer science, however, the difference comes from the domain of the entities and operations that are permitted by each of them:

- **syntax** uses the concepts and operations of **formal language theory** and represents **algorithms as automata**
    - $\rightarrow$ the **entities** are *alphabets*, *strings* and *syntax trees*
    - $\rightarrow$ the **operations** are *concatenation*, *union*, *intersection*, *complementation*, and so on
- **semantics** uses the concepts and operations of **logic** and represents **algorithms as programs**
    - $\rightarrow$ the **entities** are not as limited as in syntax: *numbers*, *string*, and any *data* of structures are available
    - $\rightarrow$ the **complexity** level is higher than in syntax: the operations are not limited to the ones of formal language theory

6.1 Syntactic translation

Definitions

- **Transducer**: an automaton that can produce output via an output function
- **Translation**: correspondence between two texts that have the same meaning in two different languages
    - the given text language is called **source** and it's denoted by Σ
    - the other language is called **target** and it's denoted by Δ
- **Purely syntactic method**: a method that applies local transformations to the source text, without considering its meaning
    - **translation grammar**: a purely syntactic translation method that uses pushdown transducers on top of a parsing method
    - **translation regular expression**: a purely syntactic translation method that uses a finite transducer enriched with an output function
- **Syntax directed method**: a method that applies local transformations to the source text, such as replacing a character with a string following the transliteration table
    - **syntax directed semantic translation**: a semi-formal approach based on a combination of syntax rules and semantic functions

It's worth to further expanding the definition of **syntax directed semantic translation**:

- the **syntactic representation** of the language is used to drive the semantic
- the following **procedure** is used:
    1. define a **set of attributes** of nonterminals of program
    2. define a **set of semantic equations** that determine how attributes can be evaluated
    3. define a **order** in which equations should be evaluated
    4. **construct a parse tree** that captures the syntactic structure of a program
    5. **traverse the tree** in order of evaluation of attributes
    6. **use equations to compute** said attributes

Analogies of Formal Language and translation Theory

- The **set of language phrases** corresponds to the **set of the pairs** *(source and destination strings)* that model the translation relation
- The **language grammar** becomes a **translation grammar,** which generates pairs of phrases
- The **finite state automaton** or the pushdown automaton becomes a **transducer automaton** or a syntax analyser that computes the translation relation

6.1.1 Translation Relation and Function

The translation can be formalized as a **binary relation** between the source and the target universal languages Σ^* and Δ^*, respectively: as such, it's a function whose domain is the subset of the cartesian product $\Sigma^* \times \Delta^*$. A **translation relation** ρ is a set of pairs of strings (x, y) with $x \in \Sigma^*$, $y \in \Delta^*$, and such that:

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

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

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

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

$$\tau : \Sigma^* \to \wp\,(\Delta^*)$$
$$\tau(x) = \{y \in \Delta^* \mid (x, y) \in \rho\}$$

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

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

Generally speaking, ρ is **partially defined**, as some strings in the source alphabet might not have a translation in the target alphabet; in order to make the function total it's necessary to add a special value `error` where the application of function τ to string x is undefined.

A particular case occurs when every source string has no more than one image, and the corresponding translation function is:

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

This case is important as it's the one that allows defining the **inverse translation** τ^{-1}: a function that maps a target string on the set of corresponding source strings:

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

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

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

Depending on the mathematical properties of τ, the following cases arise for a translation:

1. **total**: every source string has one or more images
2. **partial**: one or more source strings do not have any image
3. **single-valued**: no string has two distinct images
4. **multi-valued**: one or more source strings have more than one image
5. **injective**: distinct source strings have distinct images
   - $\to$ an alternative definition is that every target string corresponds to at most one source string: if only, in this case, the inverse translation τ^{-1} is single-valued

6. **surjective**: the image of the translation coincides with the range; every string over the target alphabet is the image of at least one source string

      $\rightarrow$ only in this case the inverse translation τ^{-1} is total

7. **bijective**: the translation is both *injective* and *surjective*

      $\rightarrow$ only in this case the inverse translation is *bijective* as well

6.1.2 Transliteration

The **transliteration** *(or alphabetic homomorphism)* is the simplest form of translation: each source character is transliterated to a target character or a string. The translation defined by an *alphabetic homomorphism* is single-valued, while this property does not necessarily apply to the inverse function.
Let $\oplus$ be any Greek letter. Then:

$$h^{-1}\left(\oplus\right) = \{\alpha, \ldots, \omega\}$$

If the homomorphism erases a letter *(maps it to the empty string)* the inverse translation is **multi-valued** because any string made of erasable characters can be inserted in any text position. If the inverse function is **single valued** too, then the transliteration is bijective and it's possible to reconstruct the source string from a given target string.

The transliteration *transforms* a letter into another one without any regard to the occurrence context.

6.1.2.1 Purely Syntactic Translation

When the source language is defined by a grammar, every syntactic component *(i.e. a subtree)* of the source language is individually *mapped* into an equivalent one in the target language.
A formal definition of Translation Grammar and Translation relation is shown in *(6.1)* and *(6.2)*

Definition 6.1 (Translation Grammar)**.** A **Translation Grammar** $G_T = (V, \Sigma, \Delta, P, S)$ is a context-free grammar that has as terminal alphabet a set $C \subseteq \Sigma^*\Delta^*$ of pairs (u, v) of source/target strings, also written as fraction $\frac{u}{v}$

Definition 6.2 (Translation Relation)**.** The **Translation Relation** ρ_G defined by grammar G_τ is:

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

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

6.1.3 Ambiguity of Source Grammar and Translation

As already observed, if the source grammar is **ambiguous**, a **sentence admits two different syntax trees**, each corresponding to a different target syntax tree. If the source grammar is not ambiguous, the translation can still be **multi valued**, if in the translation scheme different target rules are associated with the same source rule.
The next Property *(6.1)* gives a sufficient for avoiding ambiguity in the translation grammar.

Property 6.1 (Unambiguity Conditions for Translations)**.** Let $G_\tau = (G_1, G_2)$ be a **translation grammar** such that:

1. the source grammar G_1 is **unambiguous**
2. no two rules of target grammar G_2 **correspond to the same rule** of source grammar G_1

The translation specified by grammar G_τ is **single-valued** and defines a translation function.

6.1.4   Translation Grammar and Pushdown Automata

Much like the recognizer of context-free grammar, a transducer implementing a translation grammar needs a stack of unbounded length. A **pushdown transducer**, also known as a *IO automaton* is like a pushdown automaton, enriched with the capability to output zero or more characters at each move. A formal definition is shown in *(6.3)*.

> **Definition 6.3** (IO automaton). A **IO automaton** is a 8-tuple $(Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F)$:
>
> - Q is a finite set of **states**
> - Σ is the **source alphabet** *(input)*
> - Γ is the **pushdown stack alphabet**
> - Δ is the **target alphabet** *(output)*
> - δ is the state **transition** and **output function**
>   - $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to Q \times \Gamma^* \times \Delta^*$
> - $q_0 \in Q$ is the **initial state**
> - $Z_0 \in \Gamma$ is the **initial stack symbol**
> - $F \subseteq Q$ is the set of **final states**

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

$$q', a, Z$$

and it holds

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

then the automaton reads a from input and Z from the stack top, enters the state q'', pushes the string γ on the stack and outputs the string y.
If the automaton recognizes the string by empty stack, then the set of final states coincides with Q.
The subjacent automaton of the translator is obtained by erasing the target alphabet symbols and the output string actions from the definitions.

> **Definition 6.4** (Instantaneous Configuration). The **Instantaneous Configuration** of the pushdown transducer is defined as a 4-tuple $(q, y, \eta, z) \in (Q \times \Sigma^* \times \Gamma^* \times \Delta^*)$ where:
>
> - q is the **current state**
> - y is the remaining **portion *(suffix)* of the source string** x to be read
> - η is the **stack content**
> - z is the **string written to the output tape**, up to the current configuration

Moves   When a **move** is executed, a transition from a configuration to the next one occurs, denoted as $(q, x, \eta, \omega) \mapsto (p, y, \lambda, z)$. A computation is a chain of zero or more transitions, denoted by $\overset{*}{\mapsto}$.
The two possible moves depend on the current and next configuration and are shown in Table 10.

current configuration	next configuration	applied move
$(q, ax, \eta Z, z)$	$(p, x, \eta\gamma, zy)$	reading move $\delta\left(q, a, Z\right) = (p, \gamma, y)$
$(q, ax, \eta Z, z)$	$(p, ax, \eta\gamma, zy)$	spontaneous move $\delta\left(q, \varepsilon, Z\right) = (p, \gamma, y)$

Table 10: Applied moves of a pushdown transducer

The initial configuration and string acceptance conditions are the same described for pushdown automata *(Section 5.1)* and the computed translation τ is defined as follows *(assuming that the acceptance condition is*

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

6.1.5 From Translation Grammar to Pushdown Transducer

Translation schemes and pushdown transducers are two ways of representing language transformations: the former is a model suitable for the specification of the translation, while the latter is a model suitable for the implementation of the translation. Their equivalence is stated in Property 6.2.

> **Property 6.2** (Equivalence of translation grammar and pushdown transducer)**.** A translation relation is defined by a translation grammar or scheme if, and only if, it is computed by a *(eventually nondeterministic)* pushdown transducer.

Normalization of Translation Rules In order to simplify the rules, all the following hypotheses are made in the form of the source/target pairs $\frac{u}{v}$ that occur in the rules, where $u \in \Sigma^*$ and $v \in \Delta^*$:

1. For any pair $\frac{u}{v}$ it holds $|u| \leq 1$
   - $\rightarrow$ the **source** u is a **single character** $a \in \Sigma$ or the **empty string** ε
   - $\rightarrow$ the rule $\frac{a_1 a_2}{v}$ can be **replaced** by $\frac{a_1}{v} \frac{a_2}{\varepsilon}$

2. No rule may contain the following **substrings**:

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

   with $v_1, v_2 \in \Delta^*$

   - $\rightarrow$ if such combinations are present in a rule, they can be respectively **replaced** by the equivalent pairs

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

6.1.5.1 Predictive Pushdown Transducer Construction Algorithm

Let C be the **set of the pairs** of type $\frac{\varepsilon}{v}$ with $v \in \Delta^+$ and of type $\frac{b}{w}$ with $b \in \Sigma$ and $w \in \Delta^*$ occurring is some rule of the translation grammar.

The rules for constructing the transducer moves are shown in Table 11. Initially, the stack contains the axiom S and the reading head is positioned on the first character of the source string. At each step, the automaton *(non deterministically)* chooses an applicable rule and executes the corresponding move.

Note that this automaton does not need necessarily states, but they will be added later to improve the execution.

Details about the rules

- Rows $1, 2, 3, 4, 5$ apply when the stack top is a nonterminal symbol
   - $\rightarrow$ in case 2 the right part begins with a source terminal and the move is conditioned by its presence in the input string
- Rows $1, 3, 4, 5$ create spontaneous moves, which do no shit the reading head
- Rows $6, 7$ apply when the stack top is a pair.
   - $\rightarrow$ if that pair contains a source character *(row 7)* it must coincide with the current input character
   - $\rightarrow$ if that pair contains a target string *(rows 6, 7)*, the latter is output
- Finally, row 8 accepts the string if the stack is empty and the current character marks the end of the input string

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

Table 11: Construction algorithm for a predictive pushdown transducer

6.1.6 Syntax Analysis with Online Translation

A pushdown transducer created with the algorithm shown in Paragraph 6.1.5.1 is often non deterministic and generally not suitable for a compiler. A more efficient transducer can be built directly: a syntactic analyser is transformed into the corresponding syntactic transducer.

Given a transduction grammar, suppose that the underlying source grammar allows the construction of a deterministic syntax analyser. To compute the transduction, the syntax tree built by the analyser is directly translated as it's built.

As per the construction of the parser *(shown in Section 5.2)*, two different types of translations are possible: **top-down** and **bottom-up**:

- **Top-down analyser** *(LL)*, a parser can always be transformed into a transducer
- **Bottom-up analyser** *(LR)*, in order to be transformed into a transducer, every grammar rule must satisfy a special restrictive condition *(the write action can only occur at the end of the production)*

Top-down deterministic transducer Assuming that the source grammar G_1 is $ELL(k)$ with $k = 1$ *(ELL(1))*, by completing the corresponding parser with write actions *(see Section 5.2)*, a deterministic transducer can be built: this is the easiest way to create a translator.

The transducer can also be designed via a set of recursive procedures.

6.1.7 Regular Translation

A regular *(or rational)* translation expression *(or r.t.e. in short)* is a regular expression with the union, concatenation, star operator, and cross operator that has as arguments some string pairs (u, v), also written as $\frac{u}{v}$, where terms u and v are possibly empty strings, respectively, on the source and on the target alphabet

> **Definition 6.5** (Regular Translation Expression)**.** Let $C \subset \Sigma^* \times \Delta^*$ be the set of the pairs (u, v) occurring in the expression. The **Regular Translation** *(also called rational translation)* relation defined by the *r.t.e.* e_τ consists of the pairs (x, y) of the source/target strings such that:
>
> - there exists a string $z \in C^*$ in the regular set defined by the *r.t.e.* e_τ
> - strings x and y are the projections of string z on the first and second component, respectively

The set of source and target strings defined by an *r.t.e.* are regular languages; however, not every translation relation that has two regular sets as its source and target languages can be defined with an *r.t.e.*.

6.1.8 2I Automaton

Since the set C of pairs occurring in an *r.t.e.* can be viewed as a new terminal alphabet, the regular language over C can be recognized via an *FSA*. The concept of **2I Automaton** *(two input automaton)* is a method used to define the translation and to build the translation function rigorously; it allows for dealing with lexical and syntactic translation in a unified way.

> **Definition 6.6** (2I Automaton)**.** A **2I Automaton** is a **finite automaton with two inputs**. It is defined by a set of states Q, the initial state $Q_0 \in Q$ and a set $F \subseteq Q$ of final states. The transition function δ is defined as follows:
>
> $$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Delta \cup \{\varepsilon\}) \to \wp(Q)$$

An automaton move includes the following actions:

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

The automaton recognizes the source and target strings if the computation ends in a final state after both tapes have been entirely scanned.

A transition from a state q_n to a state q_m while reading a pair (a, b) respectively from the source and from the target tape is denoted as $\xrightarrow{\frac{a}{b}}$ and it's represented in Figure 9.
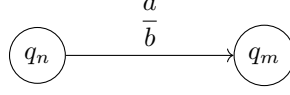
Figure 9: Example of a *2I automaton* transition

6.1.8.1 Automaton forms

By projecting the arc labels of a *2I automaton* on the first component, a new automaton with one input tape is obtained. The symbols in the tape are from the source alphabet Σ and the input automaton is **subjacent** to the original machine.

A *2I automaton* in **normal** form is characterized by the fact that each move reads exactly one character either from the source tape or from the target tape, but not from both. Arc labels are of the following types:

1. label $\dfrac{a}{\varepsilon}$ with $a \in \Sigma$, if one character from **source** is read
2. label $\dfrac{\varepsilon}{b}$, with $b \in \Delta$, if one character from **target** is read

The families of translation defined by *r.t.e.* and by finite *(eventually nondeterministic) 2I automata* coincide.

6.1.9 Nivat Theorem

The **Nivat Theorem** states some properties of the transition relation.

Theorem 6.1 (Nivat Theorem)**.** *The **Nivat Theorem states** the following* 4 *conditions are equivalent:*

1. *The translation relation ρ_τ is defined by a right-linear (or left-linear) translation grammar G_τ*
2. *The translation relation ρ_τ is defined by a 2I-automaton*
3. *The translation relation $\rho_\tau \subseteq \Sigma^* \times \Delta^*$ is regular*
4. *There exists and alphabet Ω, a regular language R over Ω and two alphabetic homomorphisms*

$$h_1 : \Omega \to \Sigma \cup \{\varepsilon\}$$
$$h_2 : \Omega \to \Delta \cup \{\varepsilon\}$$

such that:
$$\rho_\tau = \{(h_1(z), h_2(z)) \mid z \in R\}$$

6.1.10 Sequential Transducer

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

A formal definition is shown in *(6.7)*.

> **Definition 6.7** (Sequential Transducer)**.** A **Sequential Transducer** or *IO-automaton T* is a deterministic machine defined by a set Q of states, a source alphabet Σ and a target alphabet Δ, an initial state q_0 and a set $F \subseteq Q$ of final states. Furthermore, there are three single-valued functions:
>
> 1. The **state transition function** δ computes the next state
> 2. The **output function** η computes the string to be emitted by a move
> 3. The **final function** ϕ computes the last suffix to be appended to the target string at termination
>
> The domains and images of these three functions are the following:
>
> $$\delta : Q \times \Sigma \to Q \quad \eta : Q \times \Sigma \to \Delta^* \quad \phi : F \times \{\dashv\} \to \Delta^*$$

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

$$q \xrightarrow[\;u\;]{\;a\;} r$$

and means that in the state q, while reading character a, emits string u and move to the next state r. The final function $\phi(r, \dashv)$ means that when the source string has been entirely read, if the final state is r, then write string v.

For a source string x, the translation $\tau(x)$ computed by the sequential transducer T is the concatenation of two strings, produced by the output function and by the final one:

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

The machine is **deterministic** because the input automaton $\langle Q, \Sigma, \delta, q_0, F \rangle$ belonging to T is deterministic, and the output and final functions η and ϕ are single-valued.

The condition that the subjacent input automaton is deterministic does not ensure by itself that the translation is single-valued, because between two states of the sequential transducer T, for example, there may be two arcs labelled $\frac{a}{b}$ and $\frac{a}{c}$ which cause the output not to be unique.

A function computable via sequential state transducer *(IO-automaton)* is said to be a **sequential function**; the composition of two sequential functions is a sequential function.

6.1.10.1 Two opposite passes

Given a single-valued translation specified by a regular translation expression or by a *2I automaton*, it's not always possible to implement the translation via a sequential transducer such as a deterministic *IO-automaton*. However, in such cases the translation can be implemented via **two cascaded** *(deterministic)* **sequential passes**, each one being one way but scanning the string in opposite directions:

step 1. a sequential transducer scans **from left to right** and converts the **source string** into an intermediate string

step 2. another sequential transducer scans the **intermediate string from right to left** and produces the specified target string

6.2 Semantic Translation

None of the previously described syntactic translation methods can handle slightly more complicated translations, such as the translation of the conversion of a number from binary to decimal; they rely on devices that are too elementary to achieve such goals.

In order to make this kind of translation, a more powerful approach is needed, which is the **syntax-directed translation**. The term *directed* marks the difference from the purely syntactic methods explored in the previous Sections.

A semantic technique includes tree-walking procedures, which move along the syntax tree and compute some variables called **semantic attributes**: they represent the *meaning* (*or semantic*) of a certain source text.

It's important to notice that a syntax-directed method is not a formal model because attribute computing procedures are not formalized.

A syntax-directed compiler performs two cascaded phases:

phase 1. **Parsing** or syntax

phase 2. **Semantic evaluation** or semantic

(Phase 1) computes a **syntax tree**, usually condensed into an abstract syntax tree, containing the essential information for the next phase. The semantic phase *(Phase 2)* consists in applying a set of **semantic functions** on each node of the tree until all attributes have been evaluated; the set of evaluated attribute values is the **meaning** *(or translation)* and it will be found in the root of the tree.

This two-phase approach is called **two-pass compilation** and it's the most common and simplest method of compilation: the decoupling of the two phases allows the compiler designer to create them with more freedom.

6.3 Attribute Grammars

The meaning of a sentence is a set of attribute values, computed by the semantic functions and assigned to the nodes of the syntax tree. The syntax-directed translator contains the definition of the semantic functions, which are associated with the grammar rules. The set of grammar rules and associated semantic functions is called an **attribute grammar**.

For simplicity, the attribute grammar is defined with respect to an **abstract syntax**: a grammar that may be simpler than the real one, with the downside of being often ambiguous. However, this ambiguity does not prevent a single-valued translation, as the parser will pass to the semantic evaluator only one syntax tree.

Simpler compilers may combine the two phases into a single pass via a unique syntax *(the one of the language)*.

A formal definition of attribute grammar is shown in 6.8.

Definition 6.8 (Attribute grammar)**.** An **attribute grammar** is defined as follows.

1. A **context-free syntax** $G = (V, \Sigma, P, S)$ where V and Σ are the terminal and nonterminal sets, P is the production rule set and S is the axiom. It convenient *(albeit not mandatory)* to avoid having the axiom in any rule RP.
2. A **set of symbols**, the *(semantic)* attributes, associated with nonterminal and terminal syntax symbols. The set of the attributes associated with symbol $\oplus$ is denoted $\text{attr}\,(\oplus)$. The attribute set of a grammar is partitioned into two disjoint sets, the **left** and **right** attributes.
3. A **set of semantic functions** *(or rules)*.

   - each function is associated with a **production rule**

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

   where D_0 is a nonterminal and the other symbols can be **terminal** or **nonterminal**
   - the production p is called **syntactic support** of the function and it might be shared between different functions
   - the attribute σ associated with a symbol D_k is denoted by σ_k or σ_D if the syntactic symbol occurs exactly once in production p
   - a **semantic function** has the **form**

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

   function f assigns to the attribute σ of symbol D_k the value computed by the function body and its arguments can be any attributes of the same production p, excluding σ_k itself.
   - normally, semantic functions **are total in their domain** and written in a suitable notation, called semantic metalanguages, which can be informal
   - a function $\sigma_0 := f(\ldots)$ defines an **attribute**, qualified as **left**, of the nonterminal D_0, which is the *LP (or parent)* of the production
   - a function $\sigma_k := f(\ldots)$, $k \geq 1$ defines an **attribute**, qualified as **right**, of a symbol *(sibling or child)* D_k occurring in the RP
   - the same attribute cannot be in the **left in a function and right in another**
   - since terminal characters never occur in the left part, their attributes cannot be of the left type
4. The set $\text{fun}(p)$ of functions supported by production p must satisfy the following **conditions**:

   A. for each left attribute σ_0 of D_0, the exists in $\text{fun}(p)$ **exactly one function** defining the attribute
   B. For each right attribute δ_0 of D_0, **no function** exists in $\text{fun}(p)$ defining the attribute
   C. For each left attribute σ_i, $i \geq 1$, **no function** exists in $\text{fun}(p)$ defining the attribute
   D. For each attribute δ_i, $i \geq 1$, there exists in $\text{fun}(p)$ **exactly one function** defining the attribute

   - The left attributes σ_0 and the right ones δ_i with $i \geq 1$ are called **internal** for production p because they are defined by functions supported by p
   - The right attributes δ_0 and left attributes σ_i with $i \geq 1$ are called **external** for production p because they are defined by functions supported by other productions

5. Some attributes can be initialized with constant values or with values computed by external functions. This is often the case for the so-called lexical attributes, those associated with terminal symbols. For such attributes, the grammar does not specify a computation rule.

6.3.1 Dependence Graph

If the grammar is specified via a translation, it abstracts from the details of tree traversing procedures: the attribute evaluation program can be automatically constructed from the functional dependencies between attributes, supposing that the bodies of the semantic functions are given.

To prepare for such construction, the functional dependencies are formalized in progressively less abstract ways

via directed graphs:

> **Definition 6.9** (Dependence Graph of a Semantic Function)**.** The nodes of **Dependence Graph of a Semantic Function** are the arguments and results of the function considered, and there is an arc from each argument to the result.

> **Definition 6.10** (Dependence Graph of a Production p)**.** The **Dependence Graph of a Production** p, denoted by dep_p, collects the dependence graphs for all the functions supported by the production considered.

> **Definition 6.11** (Dependence Graph of a Decorated Syntax Tree)**.** The **Dependence Graph of a Decorated Syntax Tree** is obtained by pasting together the graphs of the individual productions that are used in the tree nodes.

A grammar is called **acyclic** *(or loop-free)* if the dependence graph of the tree is acyclic for every sentence. A Property relative to the correctness is given in 6.3.

> **Property 6.3** (Correct Attribute Grammar)**.** Given an attribute grammar satisfying the conditions of Definition 6.8, the following holds: if the attribute dependence graph of the tree is acyclic, the system of equations corresponding to the semantic function has exactly one solution.

Under the acyclicity condition, the equations can be ordered in such a way that each semantic function is applied after the functions that compute its arguments; this produces a value for the solution since the functions are total. The solution thus obtained is unique, as in a system of linear equations.
In order to provide a total order of nodes, the topological ordering method can be used. However, this is a rather inefficient way of computing the solution, since it would be necessary to apply the sorting algorithm even before computing the attribute values.

Checking whether a given grammar is acyclic is another problem. Since the source language is usually infinite, the acyclicity test cannot be performed by the exhaustive enumeration of all the trees; an algorithm that determines if an attribute grammar is acyclic exists but is $\mathcal{NP}$-complete and thus not used in practice. It is more convenient to test certain sufficient conditions.

6.3.2 One-Sweep Semantic Evaluation

A fast evaluator should be able to compute the values of all the attributes of a tree in a single pass: a tree can be traversed via a depth-first search, which permits the evaluation of the attributes with just one sweep over the tree.
Let N be a node of the tree and $N_1, \dots, N_r$ its children, while t_i is the subtree rooted at node N_i. A depth-first algorithm first visits the tree root. Then, in order to visit the subtree t_N rooted at node N, it recursively proceeds as follows:

1. it performs a depth-first visit of the subtrees $t_1, \dots, t_r$ in an order corresponding to some permutation of $1, \dots, r$
2. evaluates the attributes according to the following principles:
   - before entering and evaluating a subtree t_N, it computes the right attributes of node N *(the root of the subtree)*
     - $\rightarrow$ the attributes are then passed as input parameters of the procedure that implements the visit
     - $\rightarrow$ procedure calls with input parameters passing are the *"descending phase"* of the visit
   - at the end of visit of subtree t_N, it computes the left attributes of node N
     - $\rightarrow$ the attributes are the output parameters of the procedure that implements the visit
     - $\rightarrow$ procedure return an output parameter passing are the *"ascending phase"* of the visit
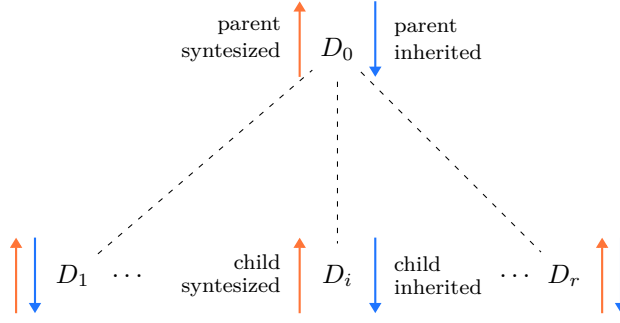
Figure 10: One-sweep semantic evaluation

Not all grammars are compatible with this procedure *(named **one-sweep**)* because more intricate functional dependencies may require several visits of the same node.

An illustration of this procedure is given in Figure 10.

6.3.2.1 One-Sweep Grammar

For each production $p : D_0 \rightarrow D_1 D_2 \ldots D_r$ with $r \geq 0$, it's necessary to introduce a new relation between the symbols of the right part of the production. Then it will be possible to build a directed graph, called **sibling graph** and denoted by sibl_p, that collects the relations between the symbols of the right part of the production p. The nodes of sibl_p are the symbols $\{D_1, \ldots, D_r\}$ of the production, and has arcs $D_i \rightarrow D_j$, with $i \neq j$, $i, j \geq 1$ if in hte dependence graph dep_p there is an arc $\sigma_i \rightarrow \delta_j$ from an attribute of symbol D_i to an attribute of symbol D_j.

The nodes of the sibling graph are not the same as those of the dependence graph: the former are syntactical symbols, the latter are attributes.

The next definition states the conditions that make a grammar suitable for one-sweep evaluation.

Definition 6.12 (one-sweep grammar). A grammar satisfies the one-sweep condition if, for each production $p : D_0 \rightarrow D_1 D_2 \ldots D_r$, $r \geq 0$ that has a dependence graph dep_p, the following clauses hold at the same time:

    A. Graph dep_p contains no circuit (is acyclic)
    B. Graph dep_p does not contain a path $\lambda_i \rightarrow \ldots \rightarrow \rho_i$, $i \geq 1$ that goes from a left attribute λ_i to a right attribute ρ_i of the same symbol D_i, where D_i is a sibling of D_0
    C. Graph dep_p contains no arc $\lambda_0 \rightarrow \rho_i$, ≥ 1, from a left attribute of the father node D_0 to a right attribute of a sibling node D_i
    D. The sibling graph sibl_p contains no circuit (is acyclic)

Explanation of the conditions

(A) necessary for the grammar to be acyclic
(B) if such a path existed, it would be impossible to compute the right attribute ρ_i before visiting the subtree t_i, because the value of the left attribute λ_i is computed at the end of the visit of t_i (against the order of the depth-first visit)
(C) the value of attribute ρ_i would not be available when the visit of the subtree t_i is started
(D) allows to topologically sort the child nodes

6.3.2.2 Construction of the One-Sweep Evaluator

The procedure visits the subtrees, computes, and returns the left attributes of the root of the subtree. For each production $p : D_0 \rightarrow D_1 D_2 \ldots D_r$, $r \geq 0$:

1. choose a topological order *(TOS)* of the nonterminals $D_1, D_2, \ldots D_r$ with respect ot the sibling graph sibl_p
2. for each symbol D_i, $1 \leq i \leq r$, choose a topological order *(TOR)* of the right attributes of symbol D_i with respect to the dependence graph dep_p
3. choose a topological order *(TOL)* of the left attributes of symbol D_0 with respect to the dependence graph dep_p

The three orders *TOS*, *TOR* and *TOL* prescribe how to arrange the instructions of the procedure that implements the visit of the subtree t_N.

To better understand the procedure, check the example on Page 454 of the textbook.

6.3.3 Combined Syntax and Semantic Analysis

It's possible to combine syntax tree construction and attribute computation, trusting the parser with the duty to invoke the semantic functions. In the following consideration, a pure *BNF* grammar is assumed, as using the *EBNF* productions would make it more complicated to specify the correspondence between syntax symbols and attributes. There are three typical situations for the , depending on the nature of the source language, shown in Table 12.

source language		tool
regular	lexical analysis with lexical attributes	`flex, lex`
$LL(k)$	recursive descent parser with attributes	
$LR(k)$	shift-reduce parser with attributes	`yacc, bison`

Table 12: Combined syntax and semantic

6.3.3.1 Lexical Analysis with Attribute Evaluation

The **lexical analyser** *(or scanner)* segments the source text into the lexical elements, called **lexemes** *(or tokens)*, such as identifiers, integers and strings; lexemes are the smallest substrings that can be invested with some semantic meaning. Each technical language uses a finite set of lexical classes, represented by regular formal language, like a regular expression or a set of strings.
Technical languages have two different levels of syntactic specifications:

- **lexical** *(lower)*, defining the form of the lexemes
- **syntactic** *(higher)*, assuming that the lexemes are given, and considering them as terminal symbols of the grammar

Some lexemes may carry meaning, like a semantic attribute, that is computed by the lexical analyser.

6.3.3.1.1 Attributed Recursive Descent Translator
Assuming that a syntax is suitable for deterministic top-down parsing *(as shown in Section 5.5)*, the attribute evaluation can proceed with parsing if the functional dependencies of the grammar obey additional conditions beyond the one-sweep grammar conditions:

- the one-sweep algorithm visits in depth-first order the syntax tree, in an order that can be different from the natural one
- the top-down parser constructs the tree in the natural order

In order to combine the two procedures is necessary to exclude any functional dependency that would make the two procedures visit the tree in different orders, as stated in Definition 6.13.

> **Definition 6.13** (L-condition)**.** A grammar satisfies the condition **L** if, for each production $p : D_0 \to D_1 \ldots D_r$, it holds:
>
>   a. The one-sweep condition *(Definition 6.12)* is satisfied
>   b. The sibling graph sibl_p *(Paragraph 6.3.2.1)* contains arc $D_j \to D_i$, $j > i \geq 1$

The condition (b) prevents a right attribute of node D_i to depend from any attribute of a node D_j placed to its **right** in the production p.

The next Property *(6.14)* relates the *L-condition* and deterministic parsing.

> **Definition 6.14** (Attribute Grammar and Deterministic Parsing)**.** Consider an attribute grammar G, where:
>
>   • the syntax satisfies the $LL(k)$ condition
>   • the semantic rules satisfy the L-condition
>
> It's possible to construct a top-down deterministic parser with attribute evaluation able to compute the attributes of G at parsing time.

6.4   Static Analysis

Static Analysis is a technique used by compilers to check some properties of the source code, before the compilation phase. The different analyses are categorized according to the purpose of the check, for example:

  • **verification**, to examine the program correctness
  • **optimization**, to improve the efficiency of the program
  • **scheduling** and parallelizing, to change the execution order of the program in order to exploit the parallelism

Such cases use a **control-flow graph**, a directed graph similar to a program flowchart. It's convenient to view this graph as describing the state-transition function of a finite automaton: static analysis is performed by analysing it, using various techniques.
It's important to notice that the control-flow graph does not represent the entire source language, but only the source code of a single program: static flow analysis must not be confused with syntax-driven translation.

6.4.1   Program as an Automaton

The control-flow graph is an abstraction of a program, that can be represented by a *FSA*; its component is now described.

  • Each **node** represent an instruction
    • the instructions are simpler than the ones of the source language
    • typical instructions are assignment, jump, arithmetic operations, ...
    • the operands are all simple variables and constants
  • Each **arc** represents a **possible control flow**
    • if an instruction p is followed by q, there is a directed **arc** from p to q
    • **unconditional** instructions have at **most one successor**
    • **conditional** instructions have **two** *(or more)* **successors**
    • an **instructions** with two or more **predecessors** is a **confluence of arcs** in the graph
  • The **first instruction** of the program is the entry point and corresponds to the **initial node**
  • The **last instruction** of the program is the exit point and corresponds to the **final node**

A control-flow graph is not a faithful representation of the program, as it misses:

- the `true`/`false` value determining the successor of a conditional instruction
- the node representing a `goto` instruction, as it's just an arc to the successor instruction
- any operation performed by an instruction, as they are replaced by an abstraction
  - a value assignment is said to define a variable
  - if a variable occurs in the RP of an expression or boolean condition, it's said to be **used**
  - a node representing a statement p in the graph is associated with **set** def(p) of **defined variables** and **set** use(p) of **used variables**

Language of the control-flow graph The language of the control-flow graph is defined in 6.15.

Definition 6.15 (Language of the control-flow graph)**.** Let A be a finite state automaton, represented by a control-flow graph. Its terminal alphabet is the set I of program instructions, each represented by a 3-tuple:

$$\langle \texttt{label, defined variables, used variables} \rangle$$

The language $L(A)$ recognized by the automaton contains the strings over the alphabet I that label the paths from the initial node *(the entry point)* to the final node *(the exit point)*: its strings represent a sequence of program instructions the machine can execute when the program is run.

Furthermore, language L is **local** *(Definition 4.12)*.

Conservative approximation The automaton can specify only an approximation of the valid execution path of a program. Not all paths are executable by the program, as no syntax analysis is performed on the conditions selecting the successor's nodes in conditional instructions. For example, consider the following instruction.

```
1: if x^2 >= 0 then
2: instruction_2 else
3: instruction_3
```

The formal language accepted by the automaton contains two paths:

$$\{1\,2,\ 1\,3\}$$

even though the second path is not executable, as the condition is always true.

As a consequence, static analysis may sometimes reach pessimistic conclusions *(it may discover never executed code paths)*. It's not decidable whether a path of a control-flow graph will ever be executed by the program, as the problem can be reduced to the halting problem.

The decision to analyze all recognized paths is a conservative approximation to program: it might diagnose non-existing errors or erroneously assign resources, but real errors will never be missed. The method is then defined as **error safe**, despite being possibly inefficient.

A usual hypothesis in static analysis is that the automaton is clean *(Section 4.1.2, page 27)*, and every instruction is on a path from the entry point to the exit point. If this is not the case, anomalies may happen:

- some executions don't reach the exit point
- some instructions are never executed *(dead code)*

6.4.2 Liveness of a variable

A professional compiler performs several passes of analysis to optimize the code. One of the most important is the analysis of **variable liveness** *(Definition 6.16)* that determines how long the value of a variable is needed.

> **Definition 6.16** (Variable liveness). A variable a is **live** on the exit from a program node p if in the program control-flow graph there exists a path from p to a node q *(not necessarily distinct from p)* such that:
>
>   • the path does not traverse an instruction $r \neq q$ that define a
>   → if r **defines** a, then $a \in \mathrm{def}\,(r)$
>   • instruction q **uses** a
>   → if q uses a, then $a \in \mathrm{use}\,(q)$
>
> For brevity, the variable liveness condition is called **live out**.

In other words, a variable is live-out of a certain node if some instruction that may be successively executed makes use of the value that variable has in the former node. If the variable gets reassigned before the next use, it's not live-out of the node.

More precisely, a variable is **live-out** for a node if it is live on any **arc leaving** from that node. Similarly, a variable is **live-in** for a node if it is live on some **arc entering** the node.

Computing Liveness Intervals   Let I be the instruction set. Let $D(a)(I) \subseteq I$ and $U(a)(I) \subseteq I$ be the sets of instructions that define and use variable a, respectively.

Variable a is live-out of an instruction p if and only if for the language $L(A)$ accepted by the automaton, the following condition holds: language $L(A)$ contains a sentence $x = upvqw$, where:

   • u, w are arbitrary **sequences of instructions** *(possibly empty)*
   • p is any **instruction**
   • v is a **possibly empty instruction sequence** not containing a definition of a
   • q is an **instruction** that uses a

The above conditions are formalized by the equation:

$$u, w \in I^* \wedge p \in I \wedge v \in (I \setminus D\,(a)) \wedge q \in U\,(a)$$

The set difference contains all the instructions that do not define a, while q uses a. The set of all strings x that meet this condition, defined as L_p, is a subset of the language $L(A)$ recognized by the automaton. Language L_p is regular because it can be defined by the following intersection:

$$L_p = L(A) \cap R_p$$

where R_p is the regular language defined by the regular expression:

$$R_p = I^* p \,(I \setminus D\,(a))^* U\,(a)\, I^*$$

The definition of R_p and L_p prescribe that letter p must be followed by a letter q taken from set $U\,(a)$ and all the letter between p and q must not be in set $D\,(a)$.

To check if a variable is a live-out of node p, it is sufficient to check if $L_p \neq \emptyset$; to achieve that goal it's sufficient to build the recognizer as the product of machine A and the recognizer of R_p. If no path connecting the input node to the final node exists, then L_p is empty.

Such a procedure is however not efficient, considering the big number of variables and instructions in real-world programs; a more efficient technique is the **data-flow analysis**. This method examines all paths from any instruction to another instruction that uses the same variable.

6.4.2.1 Data-Flow equations

The liveness computation is expressed as a system of data-flow equations: consider a node p of a program A. A first equation expresses the relation between the variables live-out (denoted $\mathrm{live_{out}}\,(p)$) and those live-in (denoted by $\mathrm{live_{in}}\,(p)$) of the node; a second equation expresses the relation between the variables live-out of a node and those live-in for its successors. $\mathrm{succ}\,(p)$ denotes the set of *(eventually immediate)* successors of node p and by $\mathrm{var}(A)$ the set of all the variables of program A.

The equations are shown in Definition 6.17.

> **Definition 6.17** (Data-Flow Equations). For each final node p:
>
> $$\text{live}_{\text{out}}(p) = \emptyset$$
>
> For any other node p:
>
> $$\text{live}_{\text{in}}(p) = \text{use}(p) \cup (\text{live}_{\text{out}}(p) \setminus \text{def}(p))$$
>
> $$\text{live}_{\text{out}}(p) = \bigcup_{q \in \text{succ}(p)} \text{live}_{\text{in}}(q)$$

6.4.2.2 Solution of Data-Flow Equations

Given a control-flow graph with a number $|I| = n \geq 1$ of nodes, the resulting system has $2 \cdot n$ equations with $2 \cdot n$ unknown variables: $\text{live}_{\text{in}}(p)$ and $\text{live}_{\text{out}}(p)$ for each node $p \in I$. Each unknown is a set of variables, and the solution to be computed is a pair of vectors, each one containing n sets.

To solve the system of equations, the following iteration is used, by taking the empty set as the initial approximation $(i = 0)$ for every unknown:

$$\forall\, p \in I \quad \text{live}_{\text{in}}(p) = \emptyset \quad \text{live}_{\text{out}}(p) = \emptyset$$

At any iteration i, for each equation of the system in 6.17, the unknowns in the right-hand side are replaced with the values computed in the previous iteration; the new iteration $i + 1$ is then calculated.

The iteration stops when the values computed in the last iteration are equal to those computed in the previous iteration *(a fixed point is reached)*; this solution is termed the **least fixed point solution** of the transformation that computes a new vector from one of the preceding iterations.

A **finite** number of iterations is always sufficient to compute the least fixed point solution, because:

- every set $\text{live}_{\text{in}}(p)$ and $\text{live}_{\text{out}}(p)$ is **finite**, since the number of variables is finite
- every iteration step either increases the cardinality of the above sets of variables or leaves them unchanged, as the equation is **monotonic**
- when the solution does not change anymore, the algorithm **terminates**

6.4.2.3 Application of Liveness Analysis

Memory allocation Liveness Analysis is best applied to decide if two variables can reside in the same memory cell: if two variables are live-in in the same program instruction, then both variables must be stored in different memory cells. The two variables then **interfere**.

Conversely, if two variables do not **interfere**, then they can be stored in the same memory cell.

Useless instructions An instruction defining a variable is **useless** is the value assigned to the variable that is never used by any instruction: the value is not live-out for the defining instruction. Therefore, to verify that the definition of a variable a by instruction p is useless, it is sufficient to check if $\text{live}_{\text{out}}(p) \not\subseteq \{a\}$.

6.4.3 Reaching Definition Analysis

Another basic applied type of static analysis is the search for a variable definition that reaches a given instruction. Consider an instruction that assigns a constant value to variable a. The compiler examines the program to see if the same constant can be replaced for the variable in the instructions using a, with the advantage of:

1. replacing the variable with a constant value, reducing the number of memory accesses and increasing the speed of the program
2. obtaining an expression where all the operands are constant, which can be evaluated at compile time

The transformation 2. is termed **constant propagation** and will be analysed later *(6.4.3.1)*.

A formal definition of the Reaching Definition problem is given in 6.18.

Definition 6.18 (Reaching Definition)**.** The **definition** of a variable a at instruction q *(denoted as a_q)* **reaches** the input of an instruction p (not necessarily different from q) if there exists a path from q to p that does not contain any redefinition of a.

Instruction p can then see and use the value of the variable a defined in instruction q.

Given an automaton A with instruction set I, the Definition 6.18 can be expressed as a regular expression over the language $L(A)$ or via a system of data-flow equations.

Definition 6.19 (Reaching Definition, via regular expression)**.** Definition a_q reaches instruction p if language $L(A)$ contains a sentence of the form $x = uqvpw$, where:

- u, w are arbitrary sequence of instructions *(possibly empty)*
- p is any instruction
- v is a sequence of instructions that do not contain any definition of a *(possibly empty)*
- q is an instruction that defines a

This condition is represented via the following regular expression:

$$u, \ w \in I^* \wedge q \in D(a) \wedge v \in (I \setminus D(a))^* \wedge p \in I$$

where p, q may coincide.

Definition 6.20 (Reaching Definition, via data-flow equations)**.** If node p defines variable a, any other definition a_q of the same variable in another node q, with $q \neq p$ is **suppressed** by p. The set of definitions suppressed by instruction p is the following:

$$\begin{cases} \mathrm{sup}(p) = \emptyset & \text{if } \mathrm{def}(p) = \emptyset \\ \mathrm{sup}(p) = \{a_q \mid q \in I \wedge q \neq p \wedge a \in \mathrm{def}(q) \wedge a \in \mathrm{def}(p)\} & \text{if } \mathrm{def}(p) \neq \emptyset \end{cases}$$

Data-Flow Equations   For the initial node 1:

$$\mathrm{in}(1) = \emptyset \tag{A}$$

For any other node $p \in I$:

$$\mathrm{out}(p) = \mathrm{def}(p) \cup (\mathrm{in}(p) \setminus \mathrm{sup}(p)) \tag{B}$$

$$\mathrm{in}(p) = \bigcup_{\forall\, q\, \in\, \mathrm{pred}(p)} \mathrm{out}(q) \tag{C}$$

Similarly to the liveness equations *(Definition 6.16)*, the reaching definition system can be solved by iteration until the computed solution converges to a fixed point; initially, all sets are empty.

The set out' denotes the elements of out reaching the exit node starting from node q, but with their subscripts deleted.

Explanation

- Equation A assumes that no variable is passed as input parameters to the subprogram
  - $\rightarrow$ otherwise, $\mathrm{in}(1)$ would be the set of input parameters
- Equation B inserts into exit from p all the local definitions of p and the definitions reaching the entrance to p, provided the latter are not suppressed by p
- Equation C states that any definition reaching the exit of some predecessor node reaches also the entrance to p

6.4.3.1 Constant Propagation

The constant propagation problem is the search for constant expressions that can be evaluated at compile time.

> **Definition 6.21** (Constant Propagation). In the instruction p, it is safe to replace with a constant k any variable a used in p if the following conditions hold:
>
> 1. There exists and instruction $q : a := k$, such that a_q reaches p
> 2. No other definition a_r of variable a reaches the entrance of p, with $r \neq q$

6.4.3.2 Availability of Variables and Initialization

A basic correctness check performed by a compiler is to verify that all the variables are initialized before their first use; more generally, a variable used in some instruction must be available at the entrance of that instruction. A definition of availability is shown in Definition 6.22, under the assumption that the subprogram has no input parameters.

> **Definition 6.22** (Variable Availability). A variable a is **available** at the entrance of instruction p *(just before its execution)* if in the program control-flow graph every path from the initial node 1 to the entrance of p contains a statement that defines variable a.

6.4.3.3 Badly Initialized Variables

A formal definition of badly initialized variables is given in 6.23.

> **Definition 6.23** (Badly Initialized Variables). An instruction p is not well initialized if the following predicate holds:
> $$\exists\, q \in \mathrm{pred}(p) \text{ such that } use(p) \nsubseteq \mathrm{out}'(q)$$

The condition says that there exists a node q predecessor of p such that the definition of reaching its exit does not include all the variables used in p. Therefore, when the program execution runs on a path through q, one or more variables used in p don't have a value.

7   Laboratory

The laboratory has the following requirements:

1. Familiarity of the `C` programming language
2. Being able to use a standard `UNIX` compilation toolchain
3. Know how a `C` `struct` is represented in memory

7.1   Regular Expressions

The `POSIX` compatible regular expression library is a standard library of the `C` programming language. It contains an extended set of functions to build and manipulate regular expressions.

Table 13 shows the basic character sets used in regular expressions, Table 14 shows the composition rules of regular expressions, and Table 15 shows some useful sets of characters.

syntax	*matches*
x	the character x
.	any character except newline
[x,y,z]	any character in the set x, y, z
[^x,y,z]	any character not in the set x, y, z
[a-z]	any character in the range a-z
[^a-z]	any character not in the range a-z

Table 13: Basic character sets

syntax	*matches*
R	the regular expression R
R S	the concatenation of R and S
R|S	the alternation of R and S
R*	zero or more occurrences of R
R+	one or more occurrences of R
R?	zero or one occurrence of R
R{n}	exactly n occurrences of R
R{n,}	at least n occurrences of R
R{n,m}	at least n and at most m occurrences of R

Table 14: Composition of Regular Expressions

7.2   Lexical Analysis

The purpose of the lexical analysis is:

1. to **recognize** the tokens of the language
2. to *(possibly)* **decorate** the tokes with additional informations

Such analysis is performed through a **scanner**, which basically is just a big *FSA*. Since coding a scanner is a hard task, scanner generators based on regular expressions such as `FLEX` are used.

In a compiler, the scanner prepares the input for the parser:

syntax	matches
(R)	capture group or override precedence
^R	match at the beginning of the line
R$	match at the end of the line
\t	tab character
\n	newline character
\w	a word *(same as [a-zA-Z0-9_])*
\d	a digit *(same as [0-9])*
\s	a whitespace character *(same as [\t\s\n])*
\W	a non-word character
\D	a non-digit character
\S	a non-whitespace character

Table 15: Regular expression utilities

1. it **detects** the tokens of the language
2. it **cleans** the input
3. it **adds** information to the tokens

Words Words cannot be enumerated in artificial languages, as there are too many of them *(despite being bounded)*. However, technical words are simpler than natural words:

- Their structure is simple
- They follow specific rules
- They are *(normally)* a regular language

> **Property 7.1** (C identifiers)**.**
>
> 1. The **first** character must be a **letter** or an **underscore**
> 2. The **following** characters must be **letters**, **digits** or **underscores**

7.3 **FLEX**

FLEX is a lexical analyser generator:

→ as **input** it takes a specification file of the scanner
← as **output** it generates a C source code file implementing the scanner

The workflow of FLEX *(illustrated in Figure 11)* is the following:

1. The **specification** file is **written**
2. The **specification** file is **compiled** with FLEX
3. The **generated** C source code file is **compiled** with cc

The generated parsers implement a nondeterministic *FSA*, later made deterministic via the Berry-Sethi algorithm, which tries to match all possible tokens at the same time; as soon as one is recognized:

1. The **semantic action** is executed
2. The **stream** skips past the end of the token
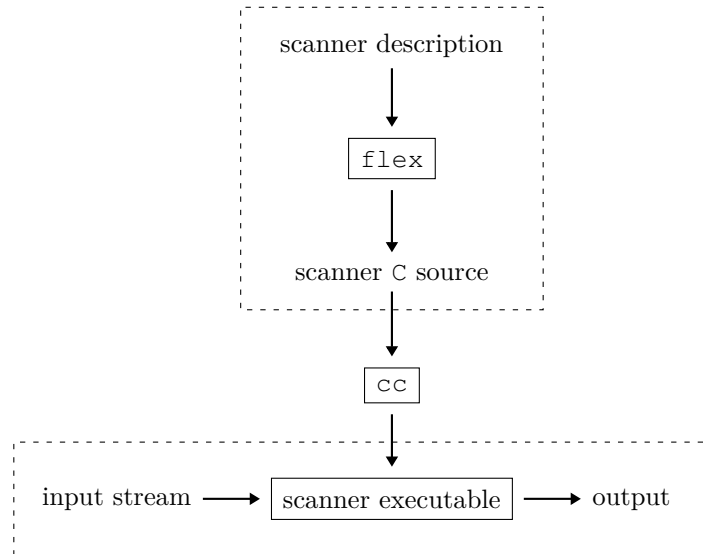3. The **automaton** reboots

Figure 11: Workflow of FLEX

7.3.1 Specification File

A FLEX file is structured in three sections, separated by the `%%` token:

- the **definitions**: declare useful *r.e.*
- the **rules**: bind *r.e.* combinations to actions
- the **user code**: C code

Definitions A definition associates a name with a set of characters; *r.e.* can be used to define character sets, with quotes *(symbol ")* used for literal strings. They are recalled by putting their names in curly braces *(symbol { })*. Definitions behave like C macros.

Rules A rule represents a full token to be recognized:

- the **token** is described by a *r.e.*
- it exploits definitions to be more readable
- it defines a **semantic action** to be made at each match
  - semantic actions are executed **every time the rules matches** and can access the matched string
  - global variables:
    - **> char** `*yytext` contains the **matched string**
    - **> int** `yyleng` contains the **length of the matched string**

Simple applications put the business logic inside the semantic actions; more complex applications that use a separate parser *(for example compilers)* do the following:

1. **assign** a **value** to the recognized token
2. **return** the token **type**

User Code User C code is copied to the generated scanner **as is**.
Arbitrary code can be put inside definitions and rules sections by escaping from the FLEX syntax within `%{` and `%}`. This feature is usually used for header inclusions, global variables, and function definitions.

7.3.2   Generated Scanner

While FLEX generates a scanner, **it is not a scanner itself**; the generated scanner is a C source code file called lex.yy.c, exporting the following variables and functions:

- **> FILE** *yyin: the input file
    - it can assume the value stdin to read from the standard input
- **> int** yylex(**void**)
    - it parses the input until:
        1. a semantic function returns; the return value is the same as the one in the action
        2. the file ends; the return value is 0
- **> void** yyerror(**char** *msg)
    - prints an error message
- **> int** yywrap(**void**)
    - it is called when the end of the file is reached
    - it allows opening a new file and continue scanning from there
    - return values:
        - ‣ 0: continue scanning
        - ‣ 1: stop scanning
    - it can be disabled with the instruction %%option noyywrap in the specification file

Behaviour   Important behaviours of the scanner:

- **Longest matching rule** - if the scanner matches **more than one string**, the rules that generates the **longest** one is chosen
- **First rule** - if the scanner matches **more than one string with the same length**, the **first** one is chosen
- **Default action** - if the scanner finds no rule, the next character in input is considered matched implicitly and printed to the standard output

7.3.3   Multiple scanners

In order to support multiple scanners, rules can be marked with the same name of the associated scanner *(the start condition)*; special actions are used to switch between scanners.
A start condition S is used to mark rules with as a prefix like <S>RULE; it marks rules as active when the scanner is running the scanner S.

- The * start condition matches **every** start condition
- The initial start condition is called **INITIAL**
- The start conditions are stored as **integers**
- The current start condition is stored in the variable **YY_START**

Start conditions can be:

- **> exclusive** - declared as %x S;
    - **disables unmarked rules** when the scanner is in the start condition S
- **> inclusive** - declared as %s S;
    - **enables unmarked rules** when the scanner is in the start condition S
    - the INITIAL condition is inclusive

Command BEGIN(S) switches to the start condition S; command ECHO copies the content of the matched string yytext to the standard output.

7.4 Syntactic Analysis - **BISON**

The purpose of the **Syntactic Analysis** is:

1. **identify** the grammar structures of the input
2. **verify** the syntactic correctness of the input
3. **build** a derivation tree of the input

It's important to notice that syntactic analysis does not determine the meaning of the input *(that is the task of semantic analysis)*. The syntactic analysis is performed over a stream of terminal symbols *(tokens)* generated by the lexical analysis; nonterminal symbols are only generated through the reduction of grammar rules.

BISON BISON is the standard tool used to generate *LR* parsers and is designed to work with FLEX; the generated parsers use the *LALR*(1) algorithm, a variant of *LR*(1).
Features of the parser:

- the pilot automaton is described by a *FSA*
- the parsing stack is used to keep the parsers state at runtime
- it acts as a typical shift-reduce parser

7.4.1 Specification File

A BISON file is structured in four sections, separated by the %% token:

- the **prologue**: include headers, global variables, and function definitions
- the **definitions**: declare tokens, operator precedence, nonterminal types
- the **rules**: declare the grammar rules
- the **user code**: C code

Definitions Different syntactic elements can be defined inside the definitions section:

- **tokens**, via the %token keyword
    - each token is assigned a numbers
    - the lexer uses said numbers to identify tokens
- **grammar** rules, in the *BNF* notation
    - if not specified, the *LP* of the first rule is the **axiom**
    - rules are specified via the **syntax**

Just like FLEX, BISON allows specifying semantic actions in grammar rules; semantic action is expressed in C instructions are can be specified at the end of each rule alternative. They are executed when the rule they are associated with has been completely recognized: the order of execution of the actions is bottom-up with respect to the syntactic tree.

```
[ RULE_LP ] : [ RULE_RP ] [ ... ]
              [ { SEMANTIC_ACTION } ]
            | [ ... ]
            ;
```

Code 8: Definition section of a BISON file

Semantic actions can also be placed in the middle of a rule; however, BISON normalizes the grammar in order to have only actions only at the end of each rule.

Semantic values By associating a variable to each token *(or non-terminal)* parsed, it is possible to keep track of what each of them represents.

- **tokens**: the value is assigned in the lexer
- **nonterminals**: the value is assigned in the semantic action

The **%union** declarator specifies the entire collection of possible data types:

- type specification for **terminals** *(tokens)* is done in the **token declaration**
- type specification for **non-terminals** is done in the special **%type instruction**

```
%union {
  [ c_type ] [ VARIABLE_NAME ];
  [ ... ]
}

[ %token <c_type> TOKEN_NAME ]
[ ... ]

[ %type <c_type> NON_TERMINAL_NAME ]
[ ... ]
```

Code 9: Semantic value declaration

About the union type union is a type of compound data defined by C, not by BISON; the latter uses this data structure to associate multiple types with semantic values.
Unions are like the struct type, but assigning a value to one item invalidates all the others: that's because unlike struct *(that allocate their items sequentially)*, unions allocate their items in the same memory location.

Accessing semantic values The semantic value of each grammar symbol in a production is stored in a variable called $i, where i is the position of the symbol in the production.

- $$ corresponds to the **semantic value** of the rule itself
- Mid-rule actions count in the numbering
- Mid-rule actions have additional restrictions:
  - the variable $$ cannot be used since it points to the semantic value of the action itself
  - it's not possible to access values of symbols that come later

```
section : LSQUARE // $1
        | ID // $2
        | RSQUARE // $3
        | { printf("section: %s", $2); } // $4
        | options // $5
        | { $$ = create_sections($2, $5); } // $6
        ;
// the variable $$ contains the value of section
```

Code 10: Semantic value usage

Interface   The generated parser is a C file ending with the `.tab.c` extension, as well as a header file ending with the `.tab.h` extension. The main parsing instruction is **int** `yyparse`(**void**), the same as the `yyparse` function in FLEX; however, compared to the latter, BISON provides a `yylval` global variable containing the semantic value of the last token returned by the lexer: its type is the one specified in the `%union` declaration.

7.4.2   Integration of **FLEX** and **BISON**

Steps to integrate FLEX and BISON:

- In the **FLEX source**:

    step 1. include the `.tab.h` header file
    step 2. assign the semantic value of the token to the correct member of the `yylval` variable
    step 3. return the token identifiers declared in the BISON file

- In the **BISON source**:

    step 4. declare and implement the `main()` function

- When **compiling**:

    step 5. generate the FLEX scanner by invoking FLEX
    step 6. generate the BISON parser by invoking BISON
    step 7. compile the C file produced by bison and flex together

The command lines to execute (step 5) to (step 7) are shown in 11; an illustration of the workflow is shown in 12.

```
flex scanner.l
bison parser.y
cc -o out lex.yy.c parser.tab.c
```

Code 11: Command lines to execute (step 5) to (step 7)

7.4.2.1 Precedence and Associativity Declarations in **BISON**

Precedence   BISON allows specifying the precedence of operators in a grammar, via the `%precedence` instruction in the **definitions** section of the file. It lists the tokens in order of **growing precedence**, grouping them by line:

↓ tokens that come **first** have the **lowest precedence**
↑ tokens that come **last** have the **highest precedence**
↔ tokens listed in the **same line** have the **same precedence**.

Associativity   The associativity of an operator can be expressed by replacing the `%precedence` instruction with one of the following:

> **%left** for left associativity
> **%right** for right associativity
> **%nonassoc** for non-associativity

```
[ %precedence | %left | %right | %nonassoc ] TOKEN [ TOKEN ... ] // lowest
[ ... ]
[ %precedence | %left | %right | %nonassoc ] TOKEN [ TOKEN ... ] // highest
```

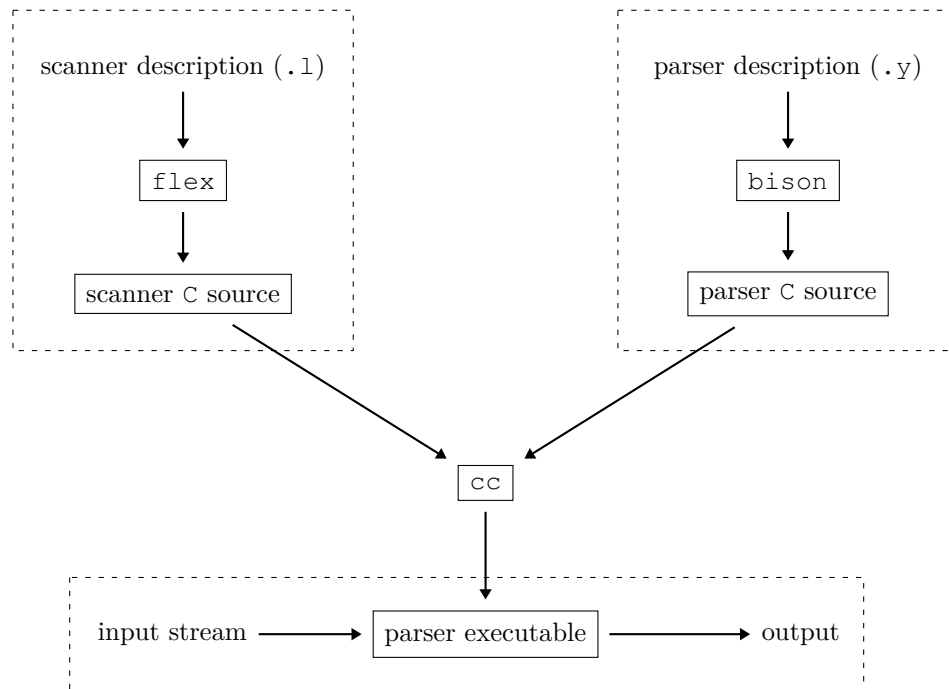Code 12: Precedence and Associativity declaration

Figure 12: Workflow of `FLEX` and `BISON`

7.4.3 Notes

It's important to notice that the produced program does not compute the value of the expressions in the input file; it just produces C code that computes the value of the expressions. The computation happens when the C code is compiled and executed: since `cc` is only a compiler, no computation happens in this stage.

In an **interpreter**, execution of the analysed parsed commands happens immediately after the parsing phase; in a **compiler**, commands are rewritten in another language without being executed. The definitions of parse time and compile time are shown in 7.1 and 7.2.

Definition 7.1 (Compile Time). Compile time computation is the computation performed in the compiler to produce the compiled output

Definition 7.2 (Run Time). Run time computation is the computation performed by the compiled output when executed

7.5 Compilers

The purpose of a **compiler** is to translate a program written in a certain language into a semantically equivalent program written in another language. Each compiler is written in a pipeline, where each stage applies a transformation to the input program and produces an output program; each stage has its purpose:

source language

↓

front-end

↓

middle-end

↓

back-end

↓

target language

front-end: converts from **source language** into an **intermediate form**
middle-end: performs **optimizations** on the intermediate form
back-end: converts from the **intermediate form** into **target language**
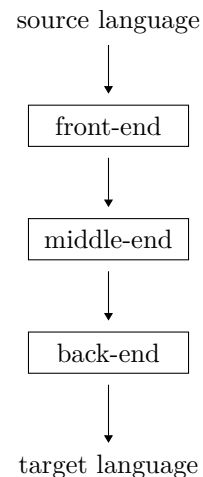
7.6 **ACSE**

ACSE is a simple compiler that:

→ as **input** accepts a C like **source language** called LANCE
← as **output** emits a RISC-like **assembly language** called MACE

It's bundled with two tools:

1. asm, an **assembler** *(from assembly to machine code)*
2. MACE, a **simulator** for the machine code

   - it **emulates** a fictional machine called MACE
   - it won't be used in this course

7.6.1 **LANCE**

LANCE is the source language recognized by ACSE. It offers a very small subset of C99 instructions, including:

> **>** standard set of **arithmetic/logic/comparison operators**
> **>** reduced set of **control flow** statements *(only while, do-while, if)*
> **>** only one **scalar** type *(int)*
> **>** only one **aggregate** type *(arrays of int)*
> **> no functions**
> **> reading** and **writing** to standard input/output via the functions:

   - read(var) to **read an integer** from **standard input** and store it in var
   - write(var) to **write** the value of var to **standard output**

A LANCE source file is composed of two sections:

1. **variable** declarations
2. **program** body as a list of statements

7.6.2 Compilation Process

ACSE compilation is performed in steps:

Front-end

1. the source code is **tokenized** by a FLEX-generated scanner
2. the stream of tokens is **parsed** by a BISON-generated parser
3. the code is **translated** to a temporary intermediate representation by the semantic actions of the parser

Middle-end   There's no middle-end in ACSE; the intermediate representation is directly piped to the back end without any optimization.

Back-end

1. the **intermediate representation** is normalized to account for physical limitations of the `MACE` machine
2. each instruction is printed out producing the `MACE` assembly code

7.6.2.1 Intermediate Representation

The **Intermediate Representation** (`IR`) is the data representation used in a compiler to represent the program. In `ACSE` it is composed of two main parts:

1. the **instruction list**

   > instructions are `RISC`-like assembly
   > all the syntactic details are abstracted
   > latter analysis of the program is simpler

2. the **variable table**

The data is stored in an unbounded number of registers in an unbounded number of memory locations.

Registers   Registers are the variables of intermediate language. All registers have the same type *(32-bit signed integer)*; the instruction using the register is responsible for the interpretation of the value as a pointer. There are two special registers:

> **zero** *(R0)* always containing the value 0

   > each write to `R0` is **ignored**

> **PSW**, also called **status word**, implicitly read and written by the arithmetic instructions; contains flags that are used by other instructions:

   > `N`, set if the result of the last arithmetic operation is **negative**
   > `Z`, set if the result of the last arithmetic operation is **zero**
   > `C`, set if the last arithmetic operation needed a **carry**
   > `V`, set if the last arithmetic operation **overflowed**

Immediates   Immediates are constant values encoded directly in the instruction.

Labels   Labels are constant pointers to a given memory location. They are used both for conditional jumps and for pointing to the location of statically allocated data; the actual address is computed by the assembler.

7.6.3   Instructions

Instruction formats and **Addressing modes** are shown respectively in Tables 16 and 17.
The **conditional jump instructions** are shown in Table 18. All the flags checked are part of the `PSW` register; Table 19 shows the j**ump instructions that allow direct numerical comparison**. The instructions displayed in the latter need a `SUB` operation before the branch; refer to Code 13 for an example.

type	*operands*	*example*
ternary	1 destination and 2 source registers	`ADD R3, R1, R2`
binary	1 destination and 1 source registers, 1 immediate operand	`ADD R3 R1 #4`
unary	1 destination register, 1 address operand	`LOAD R1 L0`
jump	1 address operand	`BEQ L0`

Table 16: Instructions Format

type	syntax	notes
register direct	R<n>	n is a register number
register indirect	(R<n>)	the data contained in the n-th register
symbolic address	L<n>	n is a label number
	<label id>	the name of the label is explicitly used
immediate	#<value>	value is a constant integer

Table 17: Addressing Modes

instruction	branch condition	test
BT	always	–
BF	never	–
BPL	if positive	!N
BMI	if negative	N
BNE	if not zero	!Z
BEQ	if zero	Z
BVC	if no overflow	!V
BVS	if overflow	V
BCC	if no carry	!C
BCS	if carry	C

Table 18: Conditional jump instructions

instruction	branch condition	test
BNE	if not equal	!Z
BEQ	if equal	Z
BGE	if greater or equal	(N && V) || (!N && !V)
BLT	if less than	(N && !V) || (!N && V)
BGT	if greater than	(!Z && (N && V) || (!N && !V))
BLE	if less or equal	(Z || (N && !V) || (!N && V))

Table 19: Numerical branch instructions

```
SUBI R0 R2 #3 // the value is saved in R0 to be discarded
BGT L0 // branches if R2 < 3
```

Code 13: Conditional jump example

7.6.4 Variable Table

The **variable table** contains the list of all the variables declared in the program. A variable can be a scalar or an **array**:

- **scalars** have a register containing where they are **stored** and an initial **value**
- **arrays** have a **size** and a **label** to a memory location where they will be **stored**

Each variable has a unique identifier equal to the name of the variable itself.

7.6.5 LANCE grammar

A LANCE source file is split into two sections:

1. **variable declarations**, where the root-nonterminal is called `var_declarations`
2. **list of statements**, where the root-nonterminal is called `statements`

A statement is a syntactic unit of an imperative programming language that expresses some actions to be carried out. They can be classified as:

- **Simple** statements, which are the basic statements of the language and cannot be further decomposed
- **Compound** statements, which are statements that can be decomposed into simpler statements

The basic grammar rules expressed in *BNF* are shown in Code 14.

```
program : var_declarations statements
var_declarations : var_declaration var_declarations
                 | empty
statements : statement statement
           | empty
code_block : statement
           | LBRACKET statements RBRACKET
var_declaration : ...
statement : ...
```

Code 14: LANCE grammar in BNF

The simplest statement is `return`: it exists the program. In asm is translated into a HALT instruction; a BISON example is shown in Code 15.

```
statement : /* ... */
          | control_statement
          | /* ... */
;


control_statement : /* ... */
                  | return_statement SEMI
;


return_statement : RETURN
                 {
                     /* insert an HALT instruction */
                     gen_halt_instruction(program);
                 }
;
```

Code 15: `BISON` return

The function `gen_halt_instruction` is a standard `C` function. It's defined in `axe_gencode.h` and it's implemented in `axe_gencode.c`; it generates an `HALT` instruction and inserts it into a linked list called **program list** placed inside the global `program` structure.
The implementation of said function in `C` is shown in Code 16.

```
void gen_halt_instruction(t_program_infos *program) {
  t_axe_instruction *inst = malloc(sizeof(t_axe_instruction));
  inst->opcode = HALT;
  program->instructions = addLast(program->instructions, inst);
}
```

Code 16: `asm` return in `C`

7.6.6 The Program instance

The instruction list is contained in a **global structure** called **program**. Its declaration happens at the top of the `Acse.y` file; it contains the intermediate representation of the program being compiled as well as other contextual informations.
Its definition is shown in Code 17.

```
typedef struct t_program_infos {
  t_list *variables;
  t_list *instructions;
  /* ... */
  int current_register;
} t_program_infos;
```

Code 17: `program` structure

7.6.7   The Variable instance

The syntax of variable declarations is similar to the C one; its grammar is shown in Code 18.

```
variable_declaration : TYPE declarator_list SEMI
declarator_list : declarator
                | declarator COMMA declarator_list
declarator : IDENTIFIER
           | IDENTIFIER LBRACKET expression RBRACKET
           | IDENTIFIER ASSIGN NUMBER
```

Code 18: Variable grammar

Variable identifiers

The token for variable identifiers is called IDENTIFIER: its semantic value is a C string with the name of the variable. Such string is dynamically allocated by the lexer: it must be freed in the semantic actions.
Code 19 shows the Acse.y file where the token is defined and Code 20 shows the Acse.lex file where the token is recognized. Code 21 shows the implementation of the C strdup() function.

```
%union {
  ...
  char *string;
  ...
}

%token <string> IDENTIFIER
```

Code 19: Content of Acse.y file

```
ID [a-zA-Z_][a-zA-Z0-9_]*
%%
{ID} {
  yylval.string = strdup(yytext);
  return IDENTIFIER;
}
```

Code 20: Content of Acse.lex file

```
char *strdup(const char *s) {
  char *d = malloc(strlen(s) + 1);   // Space for length plus nul
  if (d == NULL) return NULL;        // No memory
  strcpy(d,s);                       // Copy the characters
  return d;                          // Return the new string
}
```

Code 21: `stdrup()` function

7.6.7.1 Variable creation

Each declared variable must be added to the `variable` list: it associates the register or memory location where the variable or array *(respectively)* is stored with its name. The function to add a variable to the list is shown in Code 22.

```
void createVariable(
  t_program_infos *program,
  char *id, // the variable identifier
  int type, // type (always INTEGER_TYPE)
  int isArray, // 1 if the variable is an array, 0 otherwise
  int arraySize, // the size of the array (if isArray == 1)
  int init_value // the initial value of the variable (if isArray == 0)
);
// the identifier is freed by the function
```

Code 22: Add variable to the list

7.6.7.2 Variable access

Creating a variable only tells ACSE that it exists; no space is reserved for it in the memory. The compiler knows where its value will be found at runtime.

In ACSE all every **variable** is stored in a **register**. The signature of the function used is shown in Code 23; it returns the identifier of the register where the variable is stored.

The register identifier is the number of the register: register `R<n>` is identified by the number `n`. Make sure to never do arithmetic on the register identifier: it is not a pointer to the register.

```
int get_symbol_location(
  t_program_infos *program,
  char *id // the variable identifier
  int genLoad // always 0
);
// returns the identifier of the register containing the variable
// the identifier is not freed by the function
```

Code 23: Get variable from the list

7.6.7.3 Variable assignment

To assign a value to a variable, the compiler must first load the value in a register; an instruction to assign a value to that register is then generated.

The `read` statement assigns a value to a variable: at runtime, the value will be read from the standard input via the `READ` instruction. The variable identifier is then freed.

The grammar of the `read` statement is shown in Code 24. the family of functions `gen_xxx_instruction()` allows to generate ternary instructions.

```
statement : /* ... */
          | read_write_statement SEMI
          | /* ... */
;
read_write_statement : read_statement
                     | write_statement
                     | /* ... */
;
read_statement : READ LPAREN IDENTIFIER RPAREN
                 {
                  int location;
                  location = get_symbol_location(program, $3, 0);
                  gen_read_instruction(program, location);
                  free($3);
                 }
;
```

Code 24: Read grammar

7.6.7.4 Sharing variables between semantic actions

Generally speaking, sometimes it's necessary to declare a variable that is accessible to multiple sematic actions. There are several way to achieve this goal:

1. Use a **global variable**
   - ✓ Easy to implement
   - ✗ Doesn't work when the semantic actions are nested

2. Use a **global stack**
   - ✓ Works with nested semantic actions
   - ✗ Requires a lot of boilerplate code

3. Re purpose a symbol's **semantic value** as a variable
   - ✓ Works with nested semantic actions
   - ✓ Fairly easy to implement
   - ✗ Does not work when the variable must be accessed from multiple rules *(in this case, fall back to the global stack)*

7.6.8 Constant

The token for constant integers is called `NUMBER`: its semantic value is the integer value that appears in the `LANCE` source code.

Code 25 shows the `Acse.y` file where the token is defined and Code 26 shows the `Acse.lex` file where the token is recognized.

```
%union {
  ...
  int number;
  ...
}

%token <number> NUMBER
```

Code 25: Content of `Acse.y` file

```
DIGIT [0-9]+
%%
{DIGIT}+ {
  yylval.number = atoi(yytext); // atoi() is a standard C function converting ASCII to
    INTEGER
  return NUMBER;
}
```

Code 26: Content of `Acse.lex` file

7.6.9 Expressions

In `LANCE`, expressions appear in:

- Right-hand-side *(RHS)* of assignments
- Array indices
- Conditions in control statements

Almost all `C` operators are supported in `LANCE`:

- basic **arithmetic** operators: +, −, *, /
- **bitwise** operators: &, |, «, »
- **logical** operators: &&, ||, !
- comparison operators: ==, !=, <, >, <=, >=

The grammar of expressions is shown in Code 27, and the operator precedence is shown in Code 28. Note that:

- operators & and | have **lower precedence** than comparisons
- unary MINUS does not work well since it is left associative and has the same priority as PLUS
  - to fix this behaviour, additional parentheses around the expression are required
  - − 1 * 2 must be written as (− 1) * 2

```
exp : NUMBER
    | IDENTIFIER
    | NOT_OP exp // NOT_OP !
    | exp AND_OP exp // AND_OP &
    | exp OR_OP exp // OR_OP ||
    | exp PLUS exp // PLUS +
    | exp MINUS exp // MINUS -
    | exp MUL_OP exp // MUL_OP *
    | exp DIV_OP exp // DIV_OP /
    | exp LT exp // LT <
    | exp GT exp // GT >
    | exp EQ exp // EQ ==
    | exp NOTEQ exp // NOTEQ !=
    | exp LTEQ exp // LTEQ <=
    | exp GTEQ exp // GTEQ >=
    | exp SHL_OP exp // SHL_OP <<
    | exp SHR_OP exp // SHR_OP >>
    | exp ANDAND exp // ANDAND &&
    | LPAR exp RPAR // LPAR (, RPAR )
    | MINUS exp // the ambiguity of MINUS arises here
;
```

Code 27: Expression grammar

```
%left OROR
%left ANDAND
%left OR_OP
%left AND_OP
%left EQ NOTEQ
%left LT GT LTEQ GTEQ
%left SHL_OP SHR_OP
%left PLUS MINUS
%left MUL_OP DIV_OP
%right NOT_OP
```

Code 28: Expression precedence

7.6.10 Semantic Actions

7.6.10.1 Temporary Registers

The intermediate language can only represent operations between 2 registers at most; for more complex expressions, **temporary registers** are used. Those are not associated with variables and are used to store **intermediate results**.
ACSE provides a function to retrieve a register identifier never seen before in the translated intermediate representation; it will be used as a temporary register. The implementation of said function is shown in Code 29.

```
int getNewRegister(t_program_infos *program) {
  int result;
  result = program->current_register;
  program->current_register++;
  return result;
}
// nothing is added to the list of instructions
```

Code 29: Register identifier

Retrieving a new temporary register does not generate any instruction in the intermediate representation: book-keeping of register identifiers is done by the ACSE compiler *(at compile time)*. In theory, all infinite registers exist *a priori* in the intermediate representation; the only executed operation is deciding which register identifier is used for a given expression.

7.6.10.2 Constant assignment

The instructions to put a constant in a register are:

> **>** `gen_load_immediate()`, which creates a new register and puts the constant in it
> **>** `gen_move_immediate()`, which puts the constant in any register

Both their implementations are shown in Code 30.

```
// put a constant in a new register
int gen_load_immediate(t_program_infos *program, int imm) {
  int imm_register;
  imm_register = getNewRegister(program);
  gen_move_immediate(program, imm_register, imm);

  return imm_register;
}
// put a constant in any register
void gen_move_immediate(t_program_infos *program, int dest, int imm) {
  gen_addi_instruction(program, dest, 0, imm);
}
```

Code 30: Constant assignment

7.6.10.3 Code Generation

Basic code generation for Expressions   The code generation for expressions is shown in Code 31:

- constants are placed in new temporary registers
- registers associated with variables are used directly
- the register identifier of the subexpressions is used for the parentheses

```
exp : NUMBER
    {
      $$ = getNewRegister(program);
      gen_addi_instruction(program, $$, REG_0, $1);
    }
    | IDENTIFIER
    {
      $$ = get_symbol_location(program, $1, 0);
      free($1);
    }
    | LPAR exp RPAR
    {
      $$ = $2;
    }
    | /* ... */
```

Code 31: Expression code generation grammar

Basic code generation for Operators   Many operators directly translate into a single MACE instruction; Code 32 shows the code generation for some of them.

```
exp : /* ... */
    | exp PLUS exp
    {
      $$ = getNewRegister(program);
      gen_add_instruction(program, $$, $1, $3, CG_DIRECT_ALL);
    }
    | exp AND_OP exp
    {
      $$ = getNewRegister(program);
      gen_andb_instruction(program, $$, $1, $3, CG_DIRECT_ALL);
    }
    | exp OROR exp
    {
      $$ = getNewRegister(program);
      gen_orl_instruction(program, $$, $1, $3, CG_DIRECT_ALL);
    }
```

Code 32: Operator code generation

Code generation for Comparison Operators   Comparison operators require more complex code sequences using the set instructions: SLT, SGT, SLE, SGE and SEQ. They implement the same logic as branching, but instead of jumping they set a register to 1 or 0.
Code 33 shows the code generation for some comparison operators.

101

```
exp : /* ... */
    | exp LT exp
    {
      $$ = getNewRegister(program);
      gen_sub_instruction(program, REG_0, $1, $3, CG_DIRECT_ALL);
      gen_slt_instruction(program, $$);
    }
    | exp GT exp
    {
      $$ = getNewRegister(program);
      gen_sub_instruction(program, REG_0, $1, $3, CG_DIRECT_ALL);
      gen_sgt_instruction(program, $$);
    }
    | /* ... */
```

Code 33: Comparison operator code generation

7.6.11   Constant Folding

ACSE implements a **constant folding** optimization: instead of computing constant expressions at runtime, it computes them at compile time. This makes the execution of the program faster at the expense of a longer compilation time.
For example, the expression:

```
a + 3 * 4 - 2 + c
```

Is evaluated at compile time and replaced by the constant:

```
a + 10 + c
```

Running code at compile time is permitted only when the result does not change the behaviour of the program in any observable circumstance; the process of computing an expression value is invisible to the programmer and the compiled program will work in the same way.

7.6.11.1 Implementation of Constant Folding

The idea is to have a double meaning for the semantic value of exp:

- the **register identifier** of the expression
- the **constant value** of the expression

In each action, the operands are checked:

- if **both** are constant
    1. the result is computed at **compile time**
    2. the result expression is **replaced** by the constant
- if at least **one of them** is not a constant
    1. if there's a constant, it is moved to a register
    2. the code that will compute the result at runtime is generated
    3. the result expression is replaced by the register identifier which will contain the result

The double meaning is implemented via a new type of semantic value, called t_axe_expression; Code 34 shows the definition of the new type in the acse_struct.h header file, and Code 35 shows the grammar rules that use it, as defined in the acse.y file.
Code 36 shows the use of the helper function create_expression to create a new expression.

102

```
typedef struct t_axe_expression {
  int value;
  int expression_type; // IMMEDIATE (constant) or REGISTER (variable)
} axe_expression;
```

Code 34: Constant expression struct in `acse_struct.h`

```
%union {
  /* ... */
  t_axe_expression expr;
  /* ... */
}

%type <expr> exp
```

Code 35: Constant expression grammar rules in `acse.y`

```
t_axe_expression create_expression(int value, int expression_type) {
  t_axe_expression expr;
  expr.value = value;
  expr.expression_type = expression_type;
  return expr;
}
```

Code 36: Helper function to create a new expression in `acse_struct.h`

Expression with constant folding   Code 37 shows the code generation for expressions with constant folding.

```
exp : NUMBER
    {
      $$ = create_expression($1, IMMEDIATE);
    }
    | IDENTIFIER
    {
      int location = get_symbol_location(program, $1, 0);
      $$ = create_expression(location, REGISTER);
      free($1);
    }
    | LPAR exp RPAR
    {
      $$ = $2;
    }
    | /* ... */
```

Code 37: Expression with constant folding code generation

Operators with constant folding ACSE offers two helper functions to generate code for operators with constant folding, both defined in the `axe_expression.h` file:

> `handle_bin_numeric_op()` for arithmetic and logical operations
> `handle_binary_comparison()` for comparison operations

Both their signatures are shown in Code 38, while their usage in operators grammar is shown in Code 39; note that in the latter, non-binary operators are handled without any helper function.

```
/*
 * Valid values for binop variables are:
 * ADD SUB MUL DIV ANDL ORL EORL
 * ANDB ORB EORB SHL SHR */
t_axe_expression handle_bin_numeric_op(t_axe_expression left, t_axe_expression right,
  int binop);
/*
 * Valid values for condition are:
 * _LT_ _GT_ _EQ_ _NOTEQ_ _LTEQ_ _GTEQ_ */
t_axe_expression handle_binary_comparison (t_axe_expression left, t_axe_expression
  right, int condition);
```

Code 38: Constant expression helper functions in `axe_expression.h`

```
exp : /* ... */
    | exp AND_OP exp { $$ = handle_bin_numeric_op(program, $1, $3, ANDB); }
    | exp OR_OP exp { $$ = handle_bin_numeric_op(program, $1, $3, ORB); }
    | exp PLUS exp { $$ = handle_bin_numeric_op(program, $1, $3, ADD); }
    | exp MINUS exp { $$ = handle_bin_numeric_op(program, $1, $3, SUB); }
    | exp MUL_OP exp { $$ = handle_bin_numeric_op(program, $1, $3, MUL); }
    | exp DIV_OP exp { $$ = handle_bin_numeric_op(program, $1, $3, DIV); }
    | exp LT exp { $$ = handle_binary_comparison(program, $1, $3, _LT_); }
    | exp GT exp { $$ = handle_binary_comparison(program, $1, $3, _GT_); }
    | exp EQ exp { $$ = handle_binary_comparison(program, $1, $3, _EQ_); }
    | exp NOTEQ exp { $$ = handle_binary_comparison(program, $1, $3, _NOTEQ_); }
    | exp LTEQ exp { $$ = handle_binary_comparison(program, $1, $3, _LTEQ_); }
    | exp GTEQ exp { $$ = handle_binary_comparison(program, $1, $3, _GTEQ_); }
    | exp SHL_OP exp { $$ = handle_bin_numeric_op(program, $1, $3, SHL); }
    | exp SHR_OP exp { $$ = handle_bin_numeric_op(program, $1, $3, SHR); }
    | exp ANDAND exp { $$ = handle_bin_numeric_op(program, $1, $3, ANDL); }
    | exp OROR exp { $$ = handle_bin_numeric_op(program, $1, $3, ORL); }
    | /* ... */
```

Code 39: Constant expression operators via helper functions

7.6.11.2 Assignment

Since the right-hand-side of an assignment is an expression, there are two cases:

1. the expression is a constant: the value is materialized and assigned
2. the expression is a register: the code to copy the value from the register to the variable is generated

The code to handle both cases is shown in Code 40.

```
assign_statement : IDENTIFIER ASSIGN exp
                   {
                       int location;
                       location = get_symbol_location(program, $1, 0);

                       if ($3.expression_type == IMMEDIATE) {
                         /* constant */
                         gen_move_immediate(program, location, $3.value)
                       } else {
                         /* register */
                         gen_add_instruction(program, location, REG_0, $3.value, 0);
                       }

                       free($1);
                   }
                   ;
```

Code 40: Assignment code generation

7.6.12 Arrays

7.6.12.1 Array access

Arrays are stored in memory locations identified by labels, constant pointers to the first element of the array. The compiler does not know the address of the label, but it is able to access to the element at a given index by offsetting the memory location pointed by the label.
The intermediate representation of the MOVA instruction executes the following steps:

1. load the address into a register
2. add the offset of the desired element to the address
3. use indirect addressing to read or write the element

Furthermore ACSE provides two helper functions to handle array access, defined in axe_array.h:

> **>** loadArrayElement() to load an element from an array
> **>** storeArrayElement() to store an element into an array

The code to handle array access is shown in Code 41.

```
/* ID is the variable identifier of the array
 * index is the expression that evaluates to the index of the element to access
 * data is the expression that evaluates to the value to store
 * Return the identifier of the register which will contain the value read from the
   array element
 */
int loadArrayElement(t_program_infos *program, char *ID, t_axe_expression index);
int storeArrayElement(t_program_infos *program, char *ID, t_axe_expression index,
  t_axe_expression data);
```

Code 41: Array access code generation

7.6.12.2 Arrays in expressions

When an array appears in an expression:

1. generate a LOAD to the array element into a new register
2. the expression semantic value is that register identifiers

The grammar rule for array access is shown in Code 42.

```
exp : /* ... */
    | IDENTIFIER LSQUARE exp RSQUARE
      {
        int reg;
        reg = loadArrayElement(program, $1, $3);
        $$ = create_axe_expression(reg, REG);
        free($1);
      }
    ;
```

Code 42: Array access grammar

7.6.12.3 Array assignment

Value assignment to array is handled by the `storeArrayElement()` function; the grammar rule for array assignment is shown in Code 43.

```
assign_statement : IDENTIFIER LSQUARE exp RSQUARE ASSIGN exp
                  {
                      storeArrayElement(program, $1, $3, $6);
                      free($1);
                  }
                  | /* ... */
                  ;
```

Code 43: Array assignment grammar

7.6.12.4 Checking variable properties

Inside ACSE, both arrays and scalars are variables; as such, it might need necessary to check the properties of a variable, such as its type or its size. The function to retrieve this information is `getVariable()`, shown in Code 44; the definition of the `t_axe_variable` structure is shown in Code 45.

```
t_variable *getVariable (t_program_infos *program, char *ID);
```

Code 44: Variable properties

```
typedef struct t_axe_variable {
  int type;
  int isArray; // 1 if the variable is an array, 0 otherwise
  int arraySize, // the size of the array (if isArray == 1)
  int init_value // the initial value of the variable (if isArray == 0)
  char *ID; // the identifier of the variable
  t_axe_label *labelID; // the label of the variable
} t_axe_variable;
```

Code 45: Variable structure

7.6.12.5 Checking if a variable is an array

A common pattern is checking id a given identifier is associated to an array; such implementation is shown in Code 46.

```
char *the_id;
t_axe_variable *v_ident = getVariable(program, the_id);
if (!v_ident->isArray) {
  // the variable is not an array
  yyerror("the variable is not an array");
  YYERROR;
}
```

Code 46: Checking if a variable is an array

7.6.13 Control statements

In intermediate language, control statements are represented by branches, via jump instructions. Branch destinations are represented by labels, which are used to identify the target of a jump instruction.
There are two kinds of branches:

- → **Forward**, where the label is **after** the branch
- ← **Backward**, where the label is **before** the branch

Since ACSE is a syntactic directed translator, labels that appear after the branch must be also generated after the branch: the labels must be allocated without being generated *(they must be not added to the instruction list)*.
ACSE provides 3 primary functions to create labels:

- **>** newLabel(), which creates a new label without inserting into the new instruction list
- **>** assignLabel(), which assigns a label to a given instruction
- **>** assignNewLabel(), which creates a new label and assigns it to a given instruction,

The signatures of these functions are shown in Code 47; the instructions to create labels are shown in the next Paragraphs.

```
// create a new label
t_axe_label *newLabel(t_program_infos *program);
// assign a label to an instruction
void assignLabel(t_program_infos *program, t_axe_instruction *instruction, t_axe_label
  *label);
// create a new label and assign it to an instruction
t_axe_label *assignNewLabel(t_program_infos *program) {
  t_axe_label *label = newLabel(program);
  assignLabel(program, instruction, label);
  return label;
}
```

Code 47: Label functions

Creating forward labels

1. Create the label with the instruction newLabel()
2. Generate the branch passing the newly created label structure
3. Insert the label in the program by using the instruction assignLabel() just before generating the destination instruction

```
t_axe_label *label = newLabel(program);
gen_bt_instruction(program, label, 0);

/* ... */

assignLabel(program, instruction, label);
```

Code 48: Creating forward labels

Creating backward labels

1. Create and insert the label in the program with the instruction `assignNewLabel()`
2. Generate the branch instruction when needed

```
t_axe_label *label = assignNewLabel(program);
label = assignNewLabel(program);

/* ... */

gen_bt_instruction(program, label, 0);
```

Code 49: Creating backward labels

7.6.13.1 `if` statement without `else`

The grammar for the `if` statement without `else` is shown in Code 50. The expression iside the parenthesis is called **condition**; the code block is executed only if the condition is not equal to zero.
The instruction in the intermediate representation is a `BEQ` on the `Z` flag.

```
if_statement : IF LPAR exp RPAR block;
```

Code 50: Grammar for the if statement

Three semantic actions are needed:

1. One for the code before the body
2. One for the body
3. One for assigning the label to the end of the body

Parsing the `exp` nonterminal does not always results in code generation since the expression can be a variable or a constant.
The label must be shared between the first two actions; the code grammar for the `if` statement is shown in Code 51.

```
%union {
  /* ... */
  t_axe_label *label;
  /* ... */
}
%token <label> IF
/* ... */
if_statement : IF LPAR exp RPAR { /* ... */ }
             | code_block { /* ... */ }
             ;
```

Code 51: Code for the if statement

Conditional jumps and constant expression   When the conditional expression is of immediate type, the expression can be evaluated at compile time. Since said expression is constant, it's possible to know in advance *(at compile time)* if the branch will never *(or always)* be taken: the **conditional** branch can be replaced by an **unconditional** one.

7.6.13.2 `if` statement with `else`

Since the `else` statement is optional, a new nonterminal is needed to handle the case in which it is not present; such grammar is shown in Code 52.

```
if_statement : if_stmt
             | if_stmt ELSE code_block
             ;
if_stmt : IF LPAR exp RPAR code_block
        ;
```

Code 52: Grammar for the if statement with else

7.6.13.3 `while` statement

This statement repeats the contained code block until a given condition is not satisfied; this is the simplest looping construct in the language.
Its grammar is shown in Code 53.

```
while_statement : WHILE LPAR exp RPAR code_block;
```

Code 53: Grammar for the while statement

A loop is implemented by via an unconditional backward branch at the end of the loop and a conditional one at the start. Before each iteration, the condition is checked; if not satisfied, the execution of the program continues after the code block.
Due to this structure, it's necessary to share informations between the two semantic actions:

- the label for breaking out of the loop at the end

- the label for continuing the loop at the start of the loop

The definition of the `while` statement, as implemented in `axe_struct.h` is shown in Code 54; the semantic actions are shown in Code 55.

```
typedef struct {
  t_axe_label *break_label;
  t_axe_label *continue_label;
  t_axe_code_block *code_block;
} t_axe_while_statement;
```

Code 54: Code for the while statement

```
%union {
  /* ... */
  t_axe_while_statement *while_statement;
  /* ... */
}
/* ... */
%token <while_statement> WHILE
/* ... */
```

Code 55: Semantic actions for the while statement

7.6.13.4 `do while` statement

This statement is similar to the `while` statement, but the condition is checked after the execution of the code block. It simplifies the control flow graph, since only one label is needed.

The branch instruction is implemented via a `BNE` instruction because it must be taken if the condition is not satisfied. As such, the body is executed always at leat once.

The grammar is shown in Code 56.

```
do_while_statement : DO code_block WHILE LPAR exp RPAR;
```

Code 56: Grammar for the do while statement

8 Notes on the previous chapters

8.1 Language Families

8.1.1 Closure Properties

The closure of languages under the set operations, Kleene star and concatenation is shown in Table 20.

language family	concatenation	union	intersection	difference	complement
REG	✔	✔	✘	✘	✔
CF	✔	✔	✘	✘	✘

Table 20: Language Families closures

8.2 Generative Grammars

8.2.1 Known Languages

The following Subparagraphs contain the definition of some known languages; unless specified, all the grammars are context-free and non-ambiguous.

Palindrome Language The palindrome language is represented by the grammar:

$$S \rightarrow \varepsilon \mid aSa \mid bSb$$

Dyck Language A right-recursive grammar defining the Dyck language is:

$$S \rightarrow \varepsilon \mid (S)S$$

A left-recursive one is:

$$S \rightarrow \varepsilon \mid S(S)$$

An alternative is represented by the grammar:

$$S \rightarrow ((S))^*$$

8.2.2 From context-free to regular

A right or left unilinear grammar can be transformed into a regular grammar by mapping each grammar rule to an automaton transition.

8.3 Parsing

8.3.1 Advantages of Top-down Parsers over Bottom-up Parsers

- **Anticipated decision**
  - $\rightarrow$ each m-state base has only a single item
  - $\rightarrow$ the parser can decide which rule to apply to a phrase as soon as it finds the leftmost character of the rule, without waiting for the reduction

- **No need for stack pointers**
  - $\rightarrow$ once the parser has decided which rule to apply, it can immediately start the reduction
  - $\rightarrow$ there's no need to push on the stack the state path followed
  - $\rightarrow$ it suffices to push on the stack the sequence of machines followed

- **Further simplification of the pilot**

#	language	finite transducer	pushdown transducer
1	$L \in REG$	$\tau(L) \in REG$	
2	$L \in REG$		$\tau(L) \in CF$
3	$L \in CF$	$\tau(L) \in CF$	
4	$L \in CF$		$\tau(L)$ not always $\in CF$

Table 21: Closure Properties of Translation

→ every m-node of the pilot graph that contains many candidates is split into as many nodes
→ the kernel-equivalent nodes are merged and their lookahead sets are combined into one
→ new arcs, named *call arcs*, are added to the pilot graph: they represent the transfers of control from one machine to another one in the net
→ each call arc is labelled with a set of terminals, named *guide set*, determining the parser decision to transfer control to the parsed machine

8.3.2 Drawing the Pilot graph

- The **initial state** of the axiom has **no base** and its closure contains the **terminal character** ⊣
- A state without transitions labelled with a terminal has an empty closure set
- The **closure** of a m-state contains the **nonterminal characters** labelling the arches that leave the machine states inside it
- The **base** of an m-state contains the **terminal characters** labelling the arches that leave the machine states inside it
- The **lookahead** set contains all the **terminal characters that can be read** after the m-state base

[Determining if the machine net is ELL(1)]Determining if the machine net is $ELL(1)$
The machine net is not $ELL(1)$ if any of the following conditions is true:

- the guide sets of each exit arrow of the machine net are not disjoint
- the base of a m-state has more than one item
- any m-state has more than one transition
    → in this case, the graph is also missing the STP

[Determining if the machine is ELR(1)]Determining if the machine is $ELR(1)$
The machine is $ELR(1)$ if there is no shift-reduce conflict.

8.3.3 Simplification of a $ELL(1)$ Pilot

- A call arc from state q_A to state 0_B is added if and only if a machine M_A has a transition $q_A \xrightarrow{B} r_A$
- Two or more initial states in the closure of a m-state originate a **call chain**
- Transitions with the 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, thanks to the STP: they are into a one-to-one correspondence with the non-initial states of the machine net

8.4 Translation

8.4.1 Closure Properties of Translation

Given a language L and a transducer τ, the closure of the two of them with respect to the translation is shown in Table 21.

8.5 `BISON`

8.5.1 Notes on `BISON`

Usually, the needed associativity is **left**.