Formal languages and compilers

Niccoló Didoni

September 2021

Contents

Part I

Introduction to formal languages

Chapter 1

Languages

1.1 Introduction

A formal language is a artificial language

- Whose **syntax** (i.e. how the words are placed in a sentence) and **semantics** (i.e. the rules of interpretation of the sentences) must be defined in an algorithmic way, therefore exists a procedure to verify the correctness of the grammar (syntax) of a sentence and define its meaning (semantics).
- Build over an alphabet.
- That uses a grammar (i.e. a set of axiomatic rules) to define the structure of a valid sentence.
- That uses an automata to test the meaning and translate the sentence.

A compiler can be seen as the sum of a grammar and an automata, in fact a compiler has to check if the syntax of a programm is correct and than translate the code to assembly.

Translation can also be used to define meaning, in fact if a phrase has meaning it means that can be translated (or interpreted) in another language or explained in another way in the same language.

1.2 Components of a language

1.2.1 Alphabet

The most elemental component of a language is his alphabet.

Definition 1 (Alphabet)**.** *An alphabet Σ is a finite set of elements (also known as letters)*

$$\Sigma = \{a_1, ..., a_k\}$$

Cardinality

The cardinality of an alphabet is the number of elements in the alphabet

$$|\Sigma| = k$$

1.2.2 String

Definition 2 (String)**.** *A string is an ordered set of atomic elements, possibly repeated.*

Let us take an alphabet

$$\Sigma = \{a, b, c\}$$

as an example. The following combinations of the elements of Σ are all strings

$$ab, abcac, aaaabc, bcbcbc, ccccc$$

Cardinality

The cardinality of a string is the number of letters in the string

$$|s| = |abcaa| = 5$$

Cardinality can also be restricted to a single letter. In this case only the cardinality is the number of times that letter is present in the string

$$|s|_a = |abcaa|_a = 3$$

Equality

Two strings are equal if their length is the same and they contains the same letters in the same order.

1.2.3 Language

Definition 3 (Language)**.** *A language L is a finite or infinite set of strings.*

For example

$$L = \{a, bb, bab\}$$

is a language.

Cardinality of a language

The cardinality of a language is the number of strings contained in the language

$$|L| = |\{a, bb, bab\}| = 3$$

Empty language

It is convenient to define the empty language Øas the language that contains no strings (i.e. the language which cardinality is 0)

$$|\emptyset| = 0$$

1.3 Operations on strings

1.3.1 Concatenation

The concatenation (or product) . allows to add the elements of a string y (in the same order) at the end of a string x

$$x.y = xy = abccc.accc = abcccaccc$$

The . notation can also be omitted.

Properties

The main properties of concatenation are

- The concatenation is **aggregative**

$$x(yz) = (xy)z = xyz$$

- The cardinality of the concatenated string is the **sum of the cardinality** of every string that has been concatenated

$$|xyz| = |x| + |y| + |z|$$

- The concatenation has a **neutral element** ε such that

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

The string ε has cardinality 0 and it's not the empty language $\emptyset$.

1.3.2 Substring

A string y is a substring of x if x can be written as the concatenation of a string p, y and another string s

$$x = pys$$

If both p and s aren't the empty string then the substring is a proper substring. The strings p and s are also substrings and are called **prefix** and **suffix**.

1.3.3 Mirroring

The operation of mirroring inverts the order of every letter if the string (it is like reading the string the other way around)

$$x^R = (abcaa)^R = aacba$$

Precedence

The mirroring operation is always executed before the concatenation then $aabc^R$ means mirroring only the last letter (c, which will result in c because the mirror of a string of one letter is the string itself). To mirror the whole string we have to use parentheses

$$(aabc)^R = cbaa$$

Concatenation

Mirroring the concatenation of two strings x and y is equivalent to concatenating in the opposite order the strings x and y mirrored

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

1.3.4 Repetition

The operation of repetition (or power) concatenates a string to herself n times

$$x^3 = (aa)^3 = aa.aa.aa = aaaaaa$$

The repetition can also be defined recursively as

$$x^n = x^{n-1}.x \ : n > 0$$

where the base case is

$$x^0 = \varepsilon$$

1.4 Operations on languages

The operations on the strings can also been applied to languages simply applying the operation on every string of the language.

1.4.1 Mirroring

Mirroring a language L means mirror every string of L

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

1.4.2 Prefix

The language $prefix(L)$ is made of all the prefixes of all the strings of L

$$prefix(L) = \{p | y = px \wedge y \in L \wedge p, x \neq \varepsilon\}$$

Prefix-free language

A language L is prefix-free if it does not contains any string which is a prefix of another string of L.

For instance

$$L = \{x | x = a^n b^n \wedge n > 0\}$$

is a prefix free language because if we remove the last letter of a string (of the last $k < 2n$ letters) we obtain a string that can be written in the following as $a^n b^n - k$ if $k < n$ or a^{2n-k}. In both cases the string is not of the form $x = a^n b^n \wedge n > 0$ thus it does not belongs to L.

On the other hand

$$L_2 = x | x = a^m b^n \wedge m > n > 0$$

is not prefix free. To demonstrate this we only need an example. Let $s_1 = a^4 b^2$, s_1 belongs to L_2 and if we remove a letter b at the end we obtain $s_2 = a^4 b^1$ which also belongs to L_2. The string s_2 is also a prefix of s_1 because obtained removing the last letter (in other words $s_1 = a^4 b^2 = s_2.b = a^4 b.b = a^4 b^2$).

1.4.3 Concatenation

The strings that belongs to the concatenation of two languages L' and L'' are obtained concatenating two strings of L' and L'' respectively

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

1.4.4 Repetition

As for the repetition of string the repetition of languages can be defined recursively, too

$$L^n = L^{n-1}.L \ : n > 0$$

where

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

Once again it is important to say that the empty language Øis different from the language that only has the empty string $\{\varepsilon\}$ (the first has cardinality 0, so no element in it, the second has cardinality 1).

Conventions

We define

- $\emptyset^0 = \{\varepsilon\}$
- $L.\emptyset = \emptyset.L = L$
- $L.\{\varepsilon\} = \{\varepsilon\}.L = L$

1.4.5 Star operator

The star operator applied to a language L generates the union of all the powers of a L

$$L^* = \bigcup_{i=0\ldots\infty} L^i$$

In other words the star operator let us concatenate the strings of a language between them as many times as we want.

For instance if

$$L = \{ab, ba\}$$

then

$$L^* = \{ab, ba, abba, baab, abab, ..., bababaababba, babaabab, ...\}$$

1.4.6 Set theoretic operations

As languages are sets (of strings) then it is possible to apply the set operations $(\cup, \cap, \neg)$ to languages, this operations are called set theoretic operations.

Among the set theoretic operations it is useful to analyse the complement $\neg$, in fact $\neg L$ is the language that contains all the strings not contained in L. To define such strings we have to define a universal language L_{uni}, i.e. the language containing every the possible string that can be obtained

from an alphabet Σ. Such language can be defined as the union of all the strings of length n that can be build from Σ

$$L_{uni} = \bigcup_{i=0...\infty} \Sigma^i = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 ...$$

Now we can define $\neg L$ as the set difference between L_{uni} and L

$$\neg L = L_{uni} \setminus L = L_{uni} - L$$

Importance of the alphabet

From the example of the set complement we can understand of defining not only a language but also the alphabet on which the language is based.

Chapter 2

Regular languages

2.1 Regular expressions

Before talking about regular languages we have to introduce regular expressions.

Definition 4 (Regular expression)**.** *A regular expression is a string r defined over the terminal alphabet Σ on which is possible to apply the symbols $*$, ., $\cup$, (in order of decreasing precedence) $\emptyset$, (and) according the following rules*

1. *The empty symbol is a regular expression.*

$$r = \emptyset$$

2. *An element of the alphabet Σ is a regular expression.*

$$r = a \in \Sigma$$

3. *The union of two regular expressions is a regular expression.*

$$r = (s \cup t)$$

   The symbol $\cup$ can be replaced with a vertical bar $|$.

4. *The concatenation of two regular expressions is a regular expression.*

$$r = s.t$$

5. *If we apply the star operator to a regular expression we obtain a regular expression.*

$$r = s^*$$

These operations are possible because every element of the alphabet can be seen as a singleton (set of just one element), so the operations are reduced to set operations.

2.1.1 Extended regular expressions

If we add to the operations allowed in a regular expression the set operation **intersection** $\cap$, **complement** $\neg$ and **difference** $\setminus$ we obtain an extended regular expression.

Notice that it is possible to obtain intersection, complement and difference using only the operations allowed in regular expressions (i.e. union, star and concatenation).

2.1.2 Interval repetition

For a regular expression it is possible to define an interval repetition. The syntax $[a]_m^n$ indicates the concatenation of the letter a at least m times and at most n times.

$$[a]_m^n = a^m \cup a^{m+1} \cup \cdots \cup a^n$$

If $n = 1$ and $m = 0$ the syntax is further simplified

$$[a] = [a]_0^1 = \varepsilon \cup a$$

2.2 Regular languages

We can use the regular expressions to define a regular language

Definition 5 (Regular language)**.** *A language is a regular if it is generated by a regular expression.*

The language $L(r)$ defined (or generated) by a regular expression r is

$$L(r) = \{x \in \Sigma^* : r \Rightarrow^* x\}$$

2.2.1 Family of regular languages

The family of regular languages (REG) is the collection of all the regular languages. The family of regular languages is closed with respect to the operations of **union** $\cup$, **concatenation** ., **cross** $+$, **star** $*$, **intersection** $\cap$, **complement** $\neg$ and **difference** $\setminus$ and **mirroring**.

Furthermore the family of regular expressions is the smallest family that

- Contains all the finite languages.
- Is closed with respect to concatenation, union and the star operator.

Family of finite languages

The family FIN of finite languages contains all the languages with a finite number of strings. It's important to notice that **every finite language is regular**, thus the family FIN is contained in the family REG of regular languages. Since regular languages can be infinite, FIN is strictly contained in REG.

$$FIN \subset REG$$

2.2.2 Equivalence

Two regular expressions are equivalent if they define (i.e. generate) the same language.

2.3 Subexpressions

Definition 6 (Subexpression)**.** *A subexpression of a regular expression r is a regular expression s contained in s that makes sense by itself.*

For instance consider the regular expression

$$r = B.(A \cup C)$$

The regular expression

$$A \cup B$$

is a subexpression of r.

2.3.1 Choice

Definition 7. *A choice is a regular expression that corresponds to a possible instance of another regular expression.*

For instance e is a choice for the regular expressions e^*, e^+ or e^n (where e^+ is the same as $e^n : n >= 1$).

2.3.2 Derivation

Given a regular expression r we can replace a subexpression s with one of its choices to obtain another regular expression (a derived regular expression) r' and we write

$$r \Rightarrow r'$$

The symbol $\Rightarrow$ indicates the derivation relation.

The derivation can be applied in sequence until there are no choices for each subexpression.

2.3.3 Ambiguity

Sometimes it is possible to obtain the same string applying a difference sequence of derivation (e.g. we have applied different choices or the same choices but in a different order). In these cases we say that the original regular expression is ambiguous.

Most regular expressions are ambiguous but given a regular expression it is always possible to find a non ambiguous (equivalent) regexp. The disadvantage of finding a non ambiguous regular expression is that the expression is usually much more complex.

Checking ambiguity To check if a regular expression is ambiguous we can generate a new regular expression in which we add a different numeric subscript to every letter. For instance $R = a(a|b)^*c$ is transformed to $R_\# = a_1(a_2|b_3)^*c_4$. Regular expression R is ambiguous if $R_\#$ generates two strings x and y that, after removing the subscripts, are the same.

2.4 Lists

A list is a sequence of elements whose length isn't fix in advance. A list can be generated using the star or the cross operator

$$e^+ \vee e^*$$

where e is a terminal or compound object.

In general a list is generated by the regular expression

$$ie(se)^*f$$

where

- i is the initial token.
- e is the letter that represent an element of the list.
- s is the letter that separates the elements of the list.
- f is the token that marks the end of the list.

For instance in C a block of code is a list of instructions separated by a semicolon that begins with an open curly bracket and ends with a close curly bracket

```
{
    istr1;
    istr2;
    istr3;
}
```

If we do not consider spaces and newlines we can verify that the list can be generated by the regular expression

$$ie(se)^*f = \{istr_i.(;.istr_i)^*\}$$

2.4.1 Nested lists

A nested list is a collection of up to k (with k finite) lists. Every list can be used in another list

$$\begin{aligned}
&list_k = i_k.e_k.(s_k.e_k)^*.f_k \\
&list_{k-1} = i_{k-1}.list_k.(s_{k-1}.list_k)^*.f_{k-1} \\
&\vdots \\
&list_2 = i_2.list_3.(s_2.list_3)^*.f_2 \\
&list_1 = i_1.list_2.(s_1.list_2)^*.f_1
\end{aligned}$$

A nested list can be used, for instance, to parse a mathematical expression, in fact in a mathematical expression some operations have an higher priority. For instance multiplication and division have an higher priority than sum and subtraction, for this reason we could define a nested list of

two levels. The level 2 recognises multiplication and division while the level 1 recognises sum and difference

$$\begin{aligned} prod &= i.n.(s_p.n)^*.f \\ sum &= i.prod.(s_s.prod)^*.f \end{aligned} \tag{2.1}$$

where

$$\begin{aligned} n &= \{1...9\}\{0...9\}^* \\ s_p &\in \{\cdot, /\} \\ s_s &\in \{+, -\} \end{aligned}$$

It is easy to verify for example that the example 2.2 can be generated by the regular expression 2.1.

$$3 + 6 \cdot 5 + 7 + 1 + 6/3 \tag{2.2}$$

In nested lists the lowest level lists (lists with a small index) are more semantic levels.

Chapter 3

Free grammars

Definition 8 (Context free grammar)**.** *A context free grammar (or simply a free grammar, Backus Normal Form grammar, Type 2 grammar) is a quartet*

$$< \Sigma, V, A, P >$$

where

- Σ *is an alphabet of terminal symbols.*
- V *is an alphabet of non terminal symbols.*
- A *is an axiom. The axiom is a string made of non terminal and (optionally) terminal symbols.*
- P *is a set of production rules (or simply rules). Every rule has a left hand side that has to contain only non terminal symbols and a right end side that can contain both. The rule can be used to replace the left end side with the right end side of the rule.*

To derive a string from a grammar we have to start from the axiom A and use the rules in P to build a string with only terminal symbols. For instance consider the grammar

$$\begin{aligned} < \{a,b\}, \{F\}, F, \{ & \\ & F \to \varepsilon, \\ & F \to aFa, \\ & F \to bFb, \\ \} > & \end{aligned} \tag{3.1}$$

that generates strings that are palindrome (and have an even number of letters). Starting from the axiom F we could apply the second rule twice, the third rule once, the second rule again and then the first rule to obtain the string *aabaabaa*

$$F \Rightarrow^2 aFa \Rightarrow^2 aaFaa \Rightarrow^3 aabFbaa \Rightarrow^2 aabaFabaa \Rightarrow^1 aaba\varepsilon abaa = aabaabaa$$

Expression power Free grammars are more powerful than regular expression. The example 3.1 demonstrates this, in fact it is not possible to define a regular expression that generates all the palindrome strings for $\Sigma = \{a, b\}$ (an finite state automaton cannot has no memory and to generate a palindrome string it needs to remember all the letters that have been written to a certain point).

3.1 Lists

A list can be generated by a free grammar. For instance the grammar 3.2 generates a list of palindrome strings.

$$
\begin{aligned}
< \{a, b\}, \{L, P\}, L, \{ & \\
& L \to LP, \\
& L \to P, \\
& P \to \varepsilon, \\
& P \to aPa, \\
& P \to bPb, \\
\} >
\end{aligned}
\tag{3.2}
$$

3.2 Reduced form

A grammar is in a reduced form if

- Every non terminal symbol is reachable from the axiom. If it is not so those symbols that are not reachable are useless.
- Every non terminal symbol generates a non empty set of strings.

Both conditions have to be true.

If a grammar is not in a reduced form it is possible to simplify it. Grammars could also be circular (i.e. we return to the a certain string if we apply a certain rule) but also in this case it always possible to remove such circularity. For instance a grammar that contains the rules $r_1 = X \to XY$ and $r_2 = Y \to \varepsilon$ can bring to circularity ($X \Rightarrow^{r_1} XY \Rightarrow^{r_2} X$).

3.3 Free language

A context-free language (or simply free language) is the language generated by a free grammar. In other words a language is context free if and only if it can be generated by a free grammar.

Equivalence Two grammars G_1 and G_2 are equivalent if and only if they generate the same language.

$$G_1 \equiv G_2 \iff L(G_1) = L(G_2)$$

3.4 Infinite free grammars

A free grammar can be infinite, i.e. it can generate an infinite number of strings. A language $L(G)$ is infinite if and only if grammar G

- It is in **reduce form**.
- It doesn't admit **circular derivation** (obtain the same string after applying a rule).
- It admits **recursive derivations**.

Definition 9 (Recursive derivation)**.** *A recursive derivation is a derivation in which the right side of the rule contains the non terminal letter on the left side of the rule (e.g. in $E \to E + T$ the non-terminal E on the left side is used on the right side, too).*

In general a grammar is recursive if with a series of derivation we can obtain

$$A \Rightarrow^n xAy$$

In particular

- If $n = 1$ (i.e. there exists a rule $AtoxAy$), the derivation is immediately recursive.
- If the non-terminal is on the right most side of the substitution (i.e. $y = \varepsilon$) the rule is a right recursive (e.g. $A \to xA$).
- If the non-terminal is on the left most side of the substitution (i.e. $x = \varepsilon$) the rule is a left recursive (e.g. $A \to Ay$).

For instance the grammar 3.3 is infinite because all the rules are recursive (the first two rules are left immediate recursive rule while the last one is an indirect recursive rule).

$$G = \begin{cases} V = \{E, T, F\} \\ \Sigma = \{i, +, *\} \\ E \\ P = \{ \\ \quad E \to E + T \mid T, \\ \quad T \to T * F \mid F, \\ \quad F \to E \mid i \\ \} \end{cases} \tag{3.3}$$

3.5 Syntax tree

A syntax tree represents the derivation process that allows to transform the axiom of the grammar in a string.

Structure In a syntax tree

- The root is the axiom.
- Every node represent a non terminal letter.
- Every leaf represent a terminal letter.

For instance if we expand a node E using the rule $E \to E + T$ then we have to add three child nodes (one of which is a leaf) for E, $+$ (the leaf, in fact it cannot be further expanded) and T.

The set of leaves is called frontier. Scanning the frontier from left to right we get the string generated from the reduction.

Skeleton tree A syntax tree only shows which rules have been used but not the oreder in which they have been used.

A tree can be simplified removing the name of the inner nodes and leaving just the frontier.

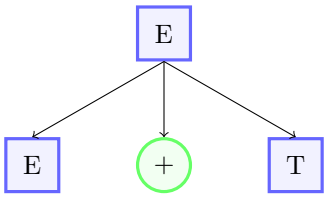


Figure 3.1: The syntax tree for the rule $E \to E + T$

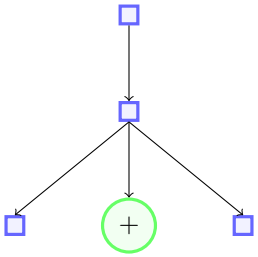


Figure 3.2: A branch of a skeleton tree.

Condensed tree A tree can be further simplified removing the branches that have limited information like linear branches (i.e. the branches generated by a rule $E \to T$).

3.6 Dyck language

Brackets (and correct parenthesising) are fundamental in programming languages. For this reason it is important to introduce parenthesis languages, that is languages whose terminal alphabet is only made of brackets and that have rules that allow only good parenthesises sequences. The paradigm of such languages is called Dyck language. An example of a Dyck language is the one generated by the grammar with the alphabet

$$\Sigma = \{(,), [,]\}$$

and the rules

$$S \to (S)S \mid [S]S \mid \varepsilon$$

Regularity A Dyck language isn't regular. To demonstrate this statement we can notice that the language

$$L = \{(^n)^n | n \geq 1\}$$

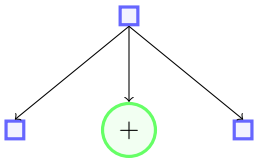


Figure 3.3: A condensed version of the tree 3.2.

is a subset of a Dyck language (L only allows sequences of opened parentheses followed by the same number of closed parentheses while a Dyck language also allows strings like (()()) which aren't generated by $(^n)^n$).

The language L isn't regular (a finite state automaton cannot recognise it, we need an automata with memory) therefore also a Dyck language isn't regular.

3.7 Closures

A free grammar is closed with respect to union $\cup$, concatenation . and star $*$ operations (i.e. if we apply such operations to free grammars we obtain a free grammar). A free grammar is also closed for mirroring.

On the other hand a free grammar is not closed for intersection $\cap$ and complement $\neg$.

3.7.1 Union

The union of two grammars G_1 and G_2 is a grammar G defined by

- $\Sigma = \Sigma_1 \cup \Sigma_2$
- $V = \{S\} \cup V_1 \cup V_2$
- S
- $P = \{S \to S_1 | S_2\} \cup P_1 \cup P_2$

3.7.2 Concatenation

The concatenation of two grammars G_1 and G_2 is a grammar G defined by

- $\Sigma = \Sigma_1 \cup \Sigma_2$
- $V = \{S\} \cup V_1 \cup V_2$
- S
- $P = \{S \to S_1.S_2\} \cup P_1 \cup P_2$

3.7.3 Star

If we apply the star operation to a grammar G_1 we obtain a grammar G defined by adding the rules

- $S \to S.S_1$
- $S \to \varepsilon$

to the rules of G_1.

If we want to apply the cross operation we simply have to add only the first rule.

3.8 Free languages

The family LIB of free languages is the collection of free languages. The family LIB strictly contains the family REG of regular languages.

$$REG \subset LIB$$

3.9 Ambiguity

Ambiguity can be divided in syntactic or semantic ambiguity. We will focus on syntactic ambiguity.

Definition 10 (Grammar ambiguity)**.** *A string is ambiguous if and only if it admits (i.e. can be generated by) to different syntactic trees. A grammar is ambiguous if it generates at least one ambiguous string.*

The trees can differ even if only two nodes have a different name.

Compilers Compilers are designed trying to avoid ambiguous grammars.

Finding ambiguous grammars A grammar that has a double recursive rule (e.g. $E \to E + E$) is always ambiguous, but in general there isn't an algorithm to determine if a grammar is ambiguous (the problem is undecidable).

Furthermore in general there's no way to get rid of ambiguity (it is possible only in some cases).

Ambiguity degree The ambiguity degree of a phrase x in the language $L(G)$ is the number of different syntax tree that x admits. The ambiguity degree can be infinite.

The ambiguity degree of a grammar G is the maximum ambiguity degree of the string generated by G.

3.9.1 Ambiguity of two-sided recursion

If a rule or a derivation (i.e. a sequence of rules) generates a rule with both right and left side recursion (e.g. $E \to E + E|e$), then the grammar is ambiguous. To show this property let us consider phrase $P = e + e + e$. P can be generated expanding the leftmost E or the rightmost E

- $E_0 \to E_1 + E_2 \to E_1 + E_3 + E_4 \to e + e + e$.
- $E_0 \to E_1 + E_2 \to E_3 + E_4 + E_2 \to e + e + e$.

This type of ambiguity can be removed by splitting the generation of the lists.

3.9.2 Ambiguity of the union

The grammar G obtained by the union of G_1 and G_2 is ambiguous if and only if $L_1(G_1)$ and $L_2(G_2)$ share some strings.

A string x that belongs to the union of L_1 and L_2 is ambiguous because G contains the set of rules of G_1 and G_2. Only the the strings that belong to $L_1 \setminus L_2$ or to $L_2 \setminus L_1$ would be generated in a non ambiguous way because they can be generated only with the rules of one grammar.

3.9.3 Inherently ambiguous languages

A language is inherently ambiguous if every grammar that generates it is necessarily ambiguous. In other words a language is inherently ambiguous if all the equivalent grammars are ambiguous. Example of inherently ambiguous grammars are

$$L = \{a^i b^j c^k \mid i, j, k \geq 0 \wedge (i = j \vee j = k)\}$$

and

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

3.9.4 Ambiguity of the concatenation

The grammar G obtained from the concatenation of two grammars G_1 and G_2 can be ambiguous if a suffix of a string of $L_1(G_1)$ is a prefix of a string of $L_2(G_2)$.

3.10 Transformations

We can transform a grammar to improve it and so that the resulting grammar has some properties (and is equivalent to the original one). Before introducing some type of transformations we have to define when two grammars are equivalent (we want to transform a grammar in an equivalent one). In particular we can distinguish between two types of equivalence

- **Weak equivalence**. A grammar G_1 is weakly equivalent to a grammar G_2 if both generate the same language. Weak equivalence is **decidable**.
- **Strong equivalence**. A grammar G_1 is strongly equivalent to a grammar G_2 if and only if G_1 and G_2 are weakly equivalent and at least the condensed trees of the strings of $L_1(G_1)$ and $L_2(G_2)$ are identical (i.e. if it's possible to assign the same structure to each string of the languages generated by the grammars). Strong equivalence is **undecidable** (but in some cases).

Strong equivalence implies weak equivalence, in fact if two grammars are strongly equivalent, they also are weakly equivalent.

$$strong \Rightarrow weak$$

3.10.1 Non terminal expression

A grammar can be transformed in a weakly equivalent one with non terminal expressions. This means that all the rules that contain a non terminal letter in the right term are transformed in rules that only have terminal letters in the right end side. A grammar with non terminal expressions is said to be in **normal form**.

For instance the grammar with the rules

$$A \to \alpha B \gamma$$
$$B \to \beta_1 | \beta_2 | \beta_3$$

is (weakly) equivalent to the grammar with the rules

$$A \to \alpha \beta_1 \gamma$$
$$A \to \alpha \beta_2 \gamma$$
$$A \to \alpha \beta_3 \gamma$$

3.10.2 Right member axiom removal

In some cases an axiom is present in the right side of a rule. For instance if E is an axiom we could have a rule $E \to E + T$. We can remove such structure adding a new axiom E_0 and adding a rule that transforms the axiom in the original rule

$$E_0 \to E + T$$

3.10.3 Nullable non terminal letter

A non terminal letter is nullable if and only if it might generate the ε string.

$$A \text{ nullable } \Leftrightarrow A \Rightarrow^{+} \varepsilon$$

Let $Null$ be the set that contains all the nullable non terminal letters. A non terminal letter A belongs to $Null$ if it appears in one of the following types of rule

- $A \to \varepsilon$
- $A \to A_1 A_2 \dots A_n \ : \forall A_i, A_i \in Null \wedge A_i \in V \setminus \{A\}$

Let's further examine these rules

- A letter A belongs to $Null$ if there exists a rule that directly maps A in ε.
- A letter A belongs to $Null$ if there exists a rule that transforms A in a concatenation of non terminal letters A_i that differ from A and that belong to $Null$.

Removing nullable letters To make all letters non nullable we have to

1. Compute the set $Null$. In other words we have to find all the nullable non terminal letters.
2. For each rule P,add the alternative rules that can be obtained from rule P by deleting each nullable non-terminal that occurs in the right member of the rule, in all possible combinations. For instance if we have a rule $A \to aA|\varepsilon$ we can replace such rule with $A \to aA|a$ (that does not contain the null string ε).
3. Remove the empty rules $A \to \varepsilon$, for every non terminal $A \neq$S.
4. Remove the rule $S \to \varepsilon$ if and only if the language does not contain ε.
5. Put the grammar into reduced form and remove the circular derivations.

3.10.4 Copy rule removal

Copy rule A copy rule is a rule that transforms a non terminal letter in another non terminal one

$$A \to B \quad : A, B \in V$$

Algorithm The algorithm used to remove copy rules (i.e. to create a set of rules P' without copy rules) is the following

1. Generate the copy set (i.e. the set of all non terminal letter that are the left part of a copy rule). The set of copy rules can be generated recursively, in fact

   $$A \in Copy(A)$$

   and

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

2. Remove the copy rules

   $$P' = P \setminus A \to B : A, B \in V$$

3. Add the rules

$$P' = P' \cup \left\{ A \to \alpha : \alpha \in \left((\Sigma \cup V)^* \setminus V \right) \wedge (B \to \alpha) \in P \wedge B \in Copy(A) \right\}$$

Removing the copy rules shortens the derivation tree of the strings generated by the grammar without copy rules. The grammar G' obtained removing the copy rules is weakly equivalent to the grammar G.

Reasons for the removal Removing copy rules

- Helps with the construction of an automaton that recognises the grammar.
- Also removes cyclic rules like $A \to B$, $B \to A$.

3.10.5 Transformation of left recursion to right recursion

Grammars with left recursive rules can't be recognised by top-down parsers. For this reason it's useful to transform such left recursive rules in right recursive ones.

Immediate recursion Immediate recursion

$$AtoAb_1|Ab_2|...|Ab_n$$
$$Atoc_1|c_2|...|c_k$$

is easier to eliminate because we simply have to transform such rules in

$$Atoc_1A'|c_2A'|...|c_kA'|c_1|c_2|...|c_k$$
$$A'tob_1A'|b_2A'|...|b_nA'|b_1|b_2|...|b_n$$

The grammar G generates a derivation chain with non terminal letters always on the left-hand side of the string

$$A \Rightarrow Ab_1 \Rightarrow Ab_3b_1 \Rightarrow Ab_2b_3b_1 \Rightarrow c_1b_2b_3b_1$$

while the grammar G' generates strings that have the same structure (a c_x letter followed by an arbitrary sequence of b_x letters) but using only right recursive rules

$$A \Rightarrow c_1A' \Rightarrow c_1b_2A' \Rightarrow c_1b_2b_3A' \Rightarrow c_1b_2b_3b_1$$

Non immediate left recursion Some rules might not be immediately left recursive but still left recursive because they generate a derivation chain like

$$A \Rightarrow Ab$$

3.10.6 Chomsky normal form

The Chomsky normal form only allows two types of rule

- $A \to BC \quad : B, C \in V$
- $Atoa \quad : a \in \Sigma$

If the empty string ε, the rule

$$S \to \varepsilon \quad : S \text{ axiom}$$

has to be in the language, but S can't appear in the right side of any rule.

3.11 Grammars and regular expressions

3.11.1 Linear grammars

Some particular types of grammars, called linear grammars, are very useful for regular expression. A linear grammar contains only one non terminal letter for each rule. In general every rule of a linear grammar $G = \{\Sigma, V, P, S\}$ can be written as

$$A \to vXu : v, u \in \Sigma^* \wedge X \in (V \cup \{\varepsilon\})$$

Notice that linear grammars generate a syntax tree that can't have more than one bifurcation (i.e. from a node we can have as many leaves as we want but only one internal node).

Linear rules can be further restricted imposing that only v or u are present. More specifically we can define

- **Right-linear rules** $AtovX$ that keep growing on the right (i.e. expanding the non terminal that is on the right).
- **Left-linear rules** $AtoXu$ that keep growing on the left.

Unilinear grammars If a grammar contains only right-linear rules or only left-linear rules it is called **unilinear** (or of type 3).

Unilinear grammars generate trees that grow only on the right (if the rules are right-linear) or on the left (if the rules are left-linear) of the root.

Strict unilinear grammars Unilinear grammars can be further restricted imposing that only one terminal per rule can be present. Practically the rules of a strict unilinear grammar have the following form (respectively for right or left linear grammars).

$$A \to aB \qquad\qquad A \to Ba$$

3.11.2 From regular expressions to grammars

Regular languages are a subset of free languages so it's possible to generate a free grammar from a regular expression. In particular, to transform a regular expression in a grammar, we have to

1. Divide the regex in sub-expressions.
2. For each sub expression create a rule.

3.11.3 Form grammars to regular expressions

A regular expression can always be transformed in a grammar using Table 3.1.

Table 3.1 allows us to map every regular into an equivalent grammar, thus the regular language generated by a regular expression can also be generated by a grammar. Since a grammar generates a context-free language, we can state that the family REG of regular languages is strictly included in the family of context free languages CF.

$$REG \subset CF$$

Regex	Grammar rule
$r_0 = r_1 \cup r_2$	$E_0 \to E_1 \vert E_2$
$r_1 = r_3^*$	$E_1 \to E_1 E_3 \vert \varepsilon$
$r_1 = r_3^+$	$E_1 \to E_1 E_3 \vert E_3$
$r_1 = r_3.r_4$	$E_1 \to E_3 E_4$
$r_1 = a$	$E_1 \to a$
$r_1 = \varepsilon$	$E_1 \to \varepsilon$

Table 3.1: Table to transform grammars in regular expressions.

3.11.4 Linear language equations

Given a linear grammar we can build a regular expression (thus showing that linear grammars generate regular languages). In particular we can build for each rule an equation in which each non terminal is replaced with the language generated by the non terminal. For instance a rule $A \to aB$ can be transformed in the equation $L_A = aL_B$.

The linear system of equation made by the rules of the grammar can be solved by like a normal system (e.g. by substitution) and recursive rules can be solved using the Arden identity.

Arden identity Let us consider an equation in the unknown language L_X

$$L_X = L_K L_X \cup L$$

where

- L_K is a non empty language.
- L is any language.

The equation has one and only one solution

$$L_X = L_K^* L$$

in fact if we replace this solution in the equation we obtain

$$\begin{aligned} L_K^* L &= L_K L_K^* L \cup L \\ L_K^* L &= L_K^+ L \cup L \\ L_K^* L &= (L_K^+ \cup \varepsilon) L \\ L_K^* L &= L_K^* L \end{aligned}$$

thus $L_X = L_K^* L$ is solves the equation.

3.11.5 Regular expressions in grammars

A regular expression can be added in the right part of a rule. For instance we could write

$$A \to (Ab)^+$$

instead of

$$A \to Ab|AbAb|AbAbAb...$$

A grammar in which a rule contains a regular expression is called **extended grammar** and still generates a free language.

Tree An extended grammar can generate trees in which a node can have an arbitrary number of children.

Ambiguity Extended grammars, like free grammars, can be ambiguous but in this case even the regular expressions used in the rules can be ambiguous. If one regex is ambiguous the grammar is ambiguous.

Part II

Automata

Chapter 4

Introduction

Automata are the devices associated to grammars that allows to translate and recognise free languages.

Compilers and parser contain some types of automata. An automata can recognise a language but can't create the derivation tree so we need add some other devices to create the derivation tree.

Chapter 5

Finite State Automata

5.1 Description

An automaton is a deterministic recognition algorithm whose

- Domain is the set of strings over the alphabet Σ.
- Image is a binary value that says if the input string is recognised or not (i.e. is generated by the grammar related to the automaton).

5.1.1 General model

A Finite State Automaton can be represented with

- An input tape that contains the input string that has to be recognised by the automaton. The input tape is read by the input head one letter at a time, from right to left and only once.
- The control unit that moves the input head.

Configuration A configuration specifies at any time what the automaton has done and what it still has to do. In particular a configuration contains

- The part of the tape that hasn't been read yet.
- The position of the input head.
- The current state of the control unit.

A final configuration is a special state in which the automaton goes if the input string is accepted. A translation is related to each final configuration. If the automaton is deterministic the string has just one translation, otherwise it could have multiple translations.

Determinism A Finite State Automaton is deterministic, in fact at each step of the recognising algorithm the automaton has at most one possible move. The algorithm also has to end at some point independently from the length of the input string. Determinism is important because we want to ensure that a single translation corresponds to an input string.

State transition graph A deterministic Finite State Automaton can be represented as a state-transition graph in which

- A node represents a state of the automaton.
- An arc represents a move of the automaton.

The graph as a unique initial state and one or more final state. Any path that connects the initial node to a final node is a valid string of the language.

5.2 Deterministic finite state automata

A deterministic finite state automata is defined as a set of five elements

- The finite, non empty set of states Q.
- The input alphabet Σ.
- The transition function $\delta(Q \times \Sigma) \to Q$ that given a state and a letter of the input alphabet returns the next state to which the FSA should go. For instance if

  $$\delta(A, a) = B$$

  then if the FSA is in the state A and reads the letter a then it has to go to the state B.
- The initial state $q_0 \in Q$.
- The set of final states $F \subseteq Q$.

Language recognition A language is recognised by a deterministic finite state automata if and only if the FSA, after processing all the string, is in a final state.

Finite state languages The family of languages recognised by finite state automata is called the family of finite state languages.

5.2.1 Clean form

Sometimes an automaton may have useless states that do not contribute in recognising a string. Before analysing such useless state we have to introduce some definitions

Accessible state A state q is accessible (or reachable) from a state p if there is a sequence of transitions that moves the automaton from state q to state q.

More specifically a state q is accessible (or reachable) if it is accessible from the initial state q_0.

Postaccessible state A state q is postaccessible (or defined) if some final state can be reached from state q.

Useful state A state q is useful if it is both accessible and postaccessible (i.e. it is placed on a path that connects the initial state to the final state).

Clean form An automaton A is in the clear form (or in reduced form) if every state of A is useful. Every FSA has an equivalent clean form.

5.2.2 Minimisation

Minimal automaton For every finite state automaton we can always find a weakly equivalent FSA (i.e. that recognises the same language) with the minimum number of states. Such automaton is unique.

Indistinguishable states A state p is indistinguishable from state q if and only if for all input letters x

- Either both next states $\delta(p, x)$ and $\delta(q, x)$ are final states
- or neither one is final

Two indistinguishable states can be merged in one state.

Distinguishable states When minimising an automaton it is more useful to consider the states that are distinguishable.

A state p is distinguishable from a state q if and only if

- p is final and q is not final (or vice versa).
- $\delta(p, a)$ is indistinguishable from $\delta(q, a)$.

Distinguishable table We can use distinguishable states to build a distinguishable table, that is a lower triangular table in which we put

- All states but the first on the rows.
- All states but the last on the columns.

For each cell C_{ij} in the table we can write

- A x if the states i and j are distinguishable.
- A $\sim$ if the states i and j are indistinguishable.
- The states to which i and j go for corresponding input. For instance if node 1 after receiving input x goes to 2 and node 3, after receiving input x, goes to 4, then we will write the couple $(2, 4)$ in the cell C_{13}.

After building the cell we have to scan all the cells until there are only xs or $\sim$. In particular

- If one of the couples of states in the cells is an indistinguishable state, we remove the couples and write a x. For instance if in a cell C_{ab} we have $(1, 3)$ and in the table the cell C_{13} has a x, we write a x over the cell C_{ab} to indicate that the states a and b are distinguishable (like the states 1 and 3).
- If one of the couples of states in the cells is a distinguishable state, we remove the couples and write a $\sim$. For instance if in a cell C_{ab} we have $(1, 3)$ and in the table the cell C_{13} has a $\sim$, we write a $\sim$ over the cell C_{ab} to indicate that the states a and b are indistinguishable (like the states 1 and 3).

Once we have found the indistinguishable states we have to draw an indistinguishable graph in which

- States are nodes.
- If a state A is indistinguishable from the state B we have an arc from A to B.

The nodes that form a complete polygons (i.e. polygons in which all nodes are connected to each other) can be merged in a single node.

5.3 Non deterministic finite state automata

Non deterministic finite state automata are FSA that allow more than one move for any input letter (e.g. from a state A, if we read x we can go to state B and to state C).

Formally a non deterministic finite state automata is a set of five elements

- The finite, non empty set of states Q.
- The input alphabet Σ.
- The transition function $\delta(Q \times \Sigma) \to \mathcal{P}(Q)$ (where $\mathcal{P}(Q)$ is the power set of Q) that given a state and a letter of the input alphabet returns the next states (as a set) to which the FSA should go. For instance if
$$\delta(A, a) = B$$
then if the FSA is in the state A and reads the letter a then it has to go to the state B.
- The initial state $q_0 \in Q$.
- The set of final states $F \subseteq Q$.

Chapter 6

Automata conversions

6.1 From automata to regex

Given a finite state automaton it's always possible to obtain the regular expression that generates the strings recognised by such automaton.

To generate the regex we have to generate the equivalent generalised automaton, i.e. an automaton that can have a regex as transition.

6.1.1 Node elimination (BMC) algorithm

Before applying the algorithm to obtain a finite state grammar we should check that the automaton has a

- Initial state with outgoing arcs only.
- Final state with incoming arcs only.

In case this property isn't respected we can add

- An initial state connected to the initial state of the automaton with a spontaneous transition (ε-move).
- A final state F connected to the final state of the automaton. State F is reached via a spontaneous transition from the final state of the automaton.

To create the equivalent generalised automaton we have to remove states one at a time until we have only two states. The regular expression on the arc that connects the two nodes is the regex that generates the language recognised by the original automaton.

In particular when we want to remove a node Q we have to replace Q with a transition that goes

- From every node i_x connected to Q with an incoming arc in Q.
- To every node o_y connected to Q with an outgoing arc from Q.

The regex on the arc has to replicate the string recognised by the sub-automaton $i_x \to Q \to o_y$.

Notice that the node elimination algorithm (also called BMC from Brzozowski and McCluskey) can generate different yet equivalent regular expressions.

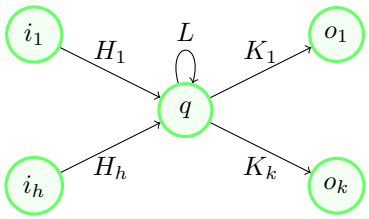


Figure 6.1: A section of a generalised finite state automaton.

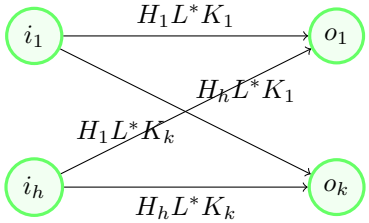


Figure 6.2: A simplification of the generalised finite state automaton in figure 6.1.

Example Consider the graph in figure 6.1. To eliminate q we can directly connect the nodes i_x to every node o_x. For instance consider the first couple $i_1 \to o_1$. In this case the arc that connects the two nodes must have the regex $H_1L^*K_1$. The simplification for every state is represented in figure 6.2.

Determinism This algorithm can be used for both deterministic and non deterministic automata.

6.2 From regular expressions to non-deterministic automata

Given a regular expression we can build a non-deterministic automaton that recognises the strings generated by the regex. It's important to remember that every unilinear grammar can be transformed in a non ambiguous one (i.e. in a deterministic one).

To get a deterministic automaton we need to eliminate

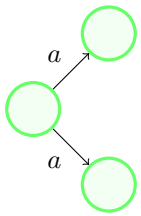
- **Spontaneous transitions** (also called ε-moves).
- **Multiple transitions with the same label**.

6.2.1 Removing spontaneous transitions

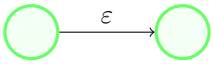
Sometimes we only need to remove ε-transitions from an automaton to obtain an automaton that can recognise the strings generated from a linear grammar.

Copy rules

To remove ε-transitions we can recognise that such transitions are equivalent to copy rules in a grammar, in fact going from a state A to a state B without reading a letter is like applying the rule $A \to B : A, B \in V$.



(a) Multiple transitions.



(b) ε-move.

Figure 6.3: Types of non-determinism.

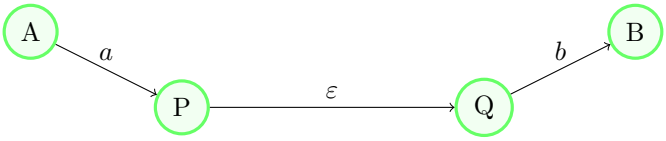


Figure 6.4: An automaton with an empty transformation.

If we see the problem in these terms then it's easy to solve it, in fact we can apply the algorithm to simplify copy rules. Even after applying the algorithm the automaton might not be in the reduced form so we might have to further reduce it.

Propagation rules

Empty transitions can be removed also acting directly on the automaton. In particular we have to cut the ε-arc and add new arcs to reconnect the nodes that have been disconnected after cutting the ε-arc. The new arcs can be generated using

- The **forward propagation rule**.
- The **backward propagation rule**.

To analyse the propagation rules consider the (sub)automaton in figure 6.4 in which the arc between P and Q is cut.

Forward propagation rule The forward propagation rule states says that the node A has to be connected with the node Q. In other words the transaction between A and P is replicated from A to Q. If the node P is the initial state of the automaton than Q becomes an initial state.

Backward propagation rule The backward propagation rule states says that the node P has to be connected with the node B. In other words the transaction between Q and B is replicated from P to B. If the node Q is a final state of the automaton than Q also becomes a final state.

Non determinism Both rules eliminate empty transactions and allow to obtain an equivalent automaton. On the other hand such rules can worsen the non-determinism of the automaton, in fact

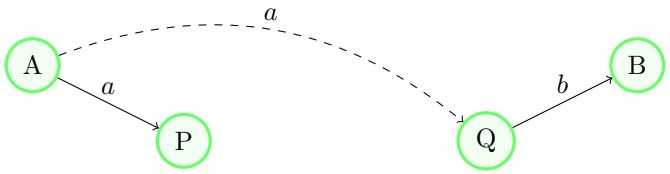


Figure 6.5: Forward propagation rule applied to the automaton in figure 6.4.

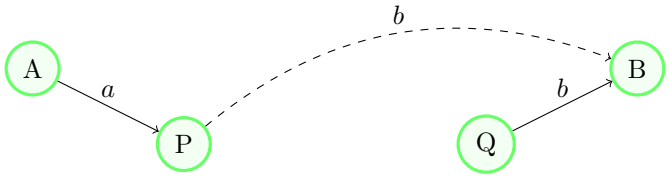


Figure 6.6: Backward propagation rule applied to the automaton in figure 6.4.

we duplicate rules with the same label. For instance in figure 6.5 we have added a transformation that exists from A and read the letter a, but such transformation already existed (from A to P).

6.2.2 Local testable languages

Local testable languages are a subset of regular languages that are easy to recognise even without an automaton. A string belongs to a local testable language if

- It begins with a letter that belongs to the set

$$Ini(L) = \{a \in \Sigma \mid a\Sigma^* \cap L \neq 0\}$$

  of initial letters.

- It ends with a letter that belongs to the set

$$Fin(L) = \{a \in \Sigma \mid \Sigma^* a \cap L \neq 0\}$$

  of final letter.

- It contains only couple of letters that belongs to the set

$$Dig(L) = \{x \in \Sigma^2 \mid \Sigma^* x \Sigma^* \cup L \neq 0\}$$

Only regular languages can be locally testable (in other words local testable languages are also regular).

For every language we can define such sets but only local testable languages are entirely described by them. For instance the language L generated by

$$a^n b^n$$

is not regular (hence not locally testable) but we can still define the three sets, in particular

- $Ini(L) = \{a\}$

- $Fin(L) = \{b\}$
- $Dig(L) = \{aa, ab, bb\}$

The language $L = a^n b^n$ isn't completly described by Ini, Fin and Dic because we can find a string that satisfy the set constraints but that isn't generated by $a^n b^n$. For instance abb

- Starts with a.
- Ends with b.
- All the couples ab, bb are in Dig.

but isn't generated by $a^n b^n$.

Local testable languages A language L is a local testable language if and only if

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

Transformation to an automaton The sets used in local testable language can be transformed in an automaton, in particular we build

- An initial state q_0 ($\cup\Sigma$).
- A final state for every letter in the Fin set.
- A transaction from the initial state q_0 to a state a that reads a if a is in the Ini set.

$$a \in Ini(L) \Rightarrow q_0 \rightarrow^a a$$

- A transaction from the a state a to a state b that reads b if the couple ab is in the Dig set.

$$ab \in Dig(L) \Rightarrow a \rightarrow^b b$$

Linear regular expressions A linear regular expression has no repeated generator. For instance

- $(abc)^*$ is linear because every letter appears only once in the regular expression (not in one of the generated strings).
- $(ab)^*a$ isn't linear because the letter a is used multiple times.

If a regular expression is linear then it is also local testable. The vice versa is not true.

$$LINEAR \Rightarrow LOCALTESTABLE$$

How to find the sets

The sets Ini, Fin and Dig can be defined recursively.

Ini The Ini set is defined as

- $Ini(\emptyset) = \emptyset$
- $Ini(\varepsilon) = \emptyset$
- $Ini(a) = \{a\}$
- $Ini(e_1e_2) = Ini(e_1)$ (unless e_1 can be nullable, in such case $Ini(e_1e_2) = Ini(e_1) \cup Ini(e_2)$)
- $Ini(e_1 \cup e_2) = Ini(e_1) \cup Ini(e_2)$
- $Ini(e_1^*) = Ini(e_1)$

Fin The Fin set is defined as

- $Fin(\emptyset) = \emptyset$
- $Fin(\varepsilon) = \emptyset$
- $Fin(a) = \{a\}$
- $Fin(e_1e_2) = Fin(e_2)$
- $Fin(e_1 \cup e_2) = Fin(e_1) \cup Fin(e_2)$
- $Fin(e_1^*) = Fin(e_1)$

Dig The Dig set is defined as

- $Dig(\emptyset) = \emptyset$
- $Dig(\varepsilon) = \emptyset$
- $Dig(a) = \emptyset$
- $Dig(e_1e_2) = Fin(e_1).Ini(e_2)$
- $Dig(e_1 \cup e_2) = Dig(e_1) \cup Dig(e_2)$
- $Dig(e_1^*) = Dig(e_1) \cup Fig(e_1).Ini(e_1)$

6.2.3 Berry-Sethi algorithm

Given a non deterministic automaton, the Berry-Sethi algorithm can be used to build a deterministic automaton equivalent to the original one. The algorithm can be applied also starting directly from the automaton.

Alphabet To build the automaton using the BS algorithm we have to rewrite the regular expression that generates the language recognised by the automaton. In particular we want to eliminate all duplicate letters in the regex by adding a subscript to all the letters in Σ. For instance the regex

$$e = (ab)^*a$$

becomes

$$e_\# = (a_1b_2)^*a_3$$

Call $\Sigma_\#$ the new alphabet that contains the letters with subscript (in our example $\Sigma_\# = \{a_1, b_2, a_3\}$).

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

Table 6.1: Followers table

Followers Define the set $Fol(c_\#)$ of the followers of $c_\# \in \Sigma_\#$ in this way

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

and

$$Fol(a_i) = \{b_j \mid a_i b_j \in Dig(e_\# \dashv)\}$$

where

- $\wp$ is the powerset.
- $Fol(\dashv) = \emptyset$

In practice the set Fol is a different way to express the digram set Dig, and it is usually presented in the form of a table. Even more practically the set of followers of a letter l_i is the set of letters that can be found after l_i in a valid string.

Algorithm The Berry-Sethi algorithm works as follows

1. Each state is denoted by a subset of $\Sigma_\# \cup \dashv$.
2. The algorithm examines the states, and constructs their destination states and outgoing moves, by the subset construction.
3. Set $Ini(e_\# \dashv)$ is the initial state, final states contain $\dashv$, and the state set Q starts with the initial state.

Example Consider the regular expression

$$e = (a|bb)^*(ac)^+$$

As first step we have to transform e in $e_\#$ adding a subscript to every letter

$$e_\# \dashv = (a_1|b_2b_3)^*(a_4c_5)^+ \dashv$$

Next we have to define the followers of each letter in $\Sigma_\#$ Then we have to define the sets Ini, Fin and Dig. In this example

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

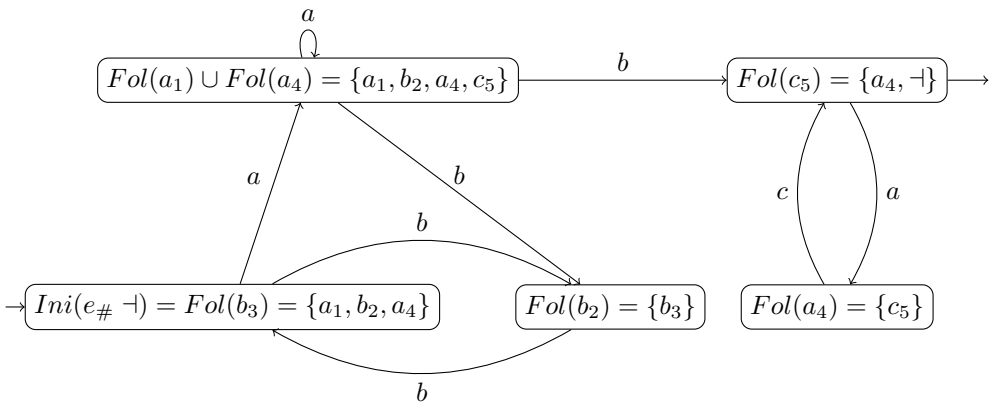


Figure 6.7: The deterministic automaton that recognises $(a|bb)^*(ac)^+$

Let's now build the deterministic automaton. In particular we start from the initial state $0 = \{a_1, b_2, a_4\}$ and consider the possible transitions. From 0 we can read a or b (since 0 contains a_i and b_i). Let us consider transition a first. Since 0 contains a_1 and a_4 we create a state 1 with the union of the followers of a_1 and a_4 (using the followers table). In particular we create state $1 = \{a_1, b_2, a_4, c_5\}$. States 0 and 1 are connected with a transition labelled with a. If we do the same with b_2 we obtain a state $2 = \{b_3\}$ reached from 0 with a transition labelled b. Notice that if a state contains $\dashv$, such state has to be final. The process has to be repeated until all states have been visited to finally obtain the automaton in Figure 6.7.

Chapter 7

Push down automata

Push down automata (PDA) are a generalisation of finite state automata.

Relation with free grammars A push down automata is equivalent to a free grammar.

Syntax analysis Push down automata are used in syntax analysis, in particular an analyser has to

1. Obtain a push down automaton from a grammar G. The PDA has to recognise G.
2. Recognise a string s using the push down automaton.
3. Build the syntax tree of s.

7.1 Definition

A push down automaton adds memory to a finite state automaton. In particular, the memory is an infinite tape on which only two operations are possible

- **Push**. The operation `push(B)` adds a symbol B at the end of the tape. It's also possible to push a sequence of letters `push(A, B, C)`. In this case the last letter added is C.
- **Pop**. The operation `pop` removes the letter at the end of the tape. When an automaton reads from memory the letter read is popped. Thus to expand the stack we have to insert back the letter popped and then add the new letter.
- **Empty**. The operation `empty` checks if the memory is empty. Usually we use a special letter Z_0 to indicate the start of the tape, thus when the automaton pops the letter Z_0 it knows that the stack is empty. The Z_0 symbol should be used only to indicate the start of the tape.

In other words the memory is a stack.

7.1.1 Formal definition

A push down automaton is a defined by

$$< Q, \Sigma, \Omega, \delta, q_0, Z_0, F >$$

where

- Q is the **set of states** Q (same definition of finite state automata).
- Σ is the alphabet of **input letters**.
- Γ is the alphabet of **memory letters**.
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to Q \times F^*$ is the **transformation function**. The transformation function takes as input a state q, an input letter a and a memory letter Z and returns the next states and the letters to write to memory (p_i, γ_i).

  $$\delta(q, a, Z) = \{(p_1, \gamma_1), ..., (p_n, \gamma_n)\}$$

  The function returns just one state if the automaton is deterministic, otherwise it might return a set of states.
- $q_0 \in Q$ is the **initial state** of the FSA.
- $Z_0 \in \Omega$ is the special symbol to represent the **bottom of the stack**.
- $F \subseteq Q$ is the set of **final states**.

Configuration A configuration is represented by

- The **current state** of the automaton.
- The **input** that still has to be analysed.
- The **content of the stack**.

Transformation A transformation brings the automaton from a configuration to another.

End of recognition We can define when a push down automaton ends it's execution using two different approaches

- Empty stack. According to this method the PDA ends the execution whenever the stack is empty. This method is used in syntax analysis.
- Final state. According to this method the PDA ends the execution whenever the last letter in input is read.

7.1.2 Spontaneous moves

It's important to analyse spontaneous moves in push down automata because some moves that would be spontaneous for finite state automata aren't so for push down automata.

Deterministic epsilon-moves If an automaton has two transformations that read ε from input but read two different letters from memory (as in example 7.1), the automaton is **deterministic**, in fact the letter read in memory allows us to decide which transformation to execute even if the letter read from input is the same.

In a FSA these transactions would have made the automaton non deterministic.

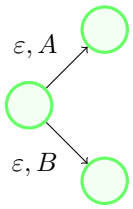


Figure 7.1: A non-spontaneous move for push down automata.

Non determinism with an epsilon move Let us consider now an automaton in figure 7.2 that can

- Read a letter a from input and a letter A from memory.
- Read nothing from input (i.e. ε) and a letter A from memory.

This automaton is **non-deterministic** in fact we can't decide which move to make. If in the second transformation we replace the ε letter with anoter letter (different from the one read in the first transformation, a) the PDA is deterministic.

In this case the stack doesn't help in removing the non determinism because it's always possible to read ε from input.

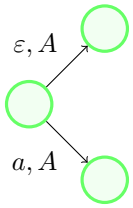


Figure 7.2: A spontaneous move for push down automata.

Part III

Syntax analysis

Chapter 8

From grammars to push down automata

8.1 Introduction

Syntax analysis allows to recognise an input string generated by a grammar and build the corresponding syntax tree. This actions are executed by an, ideally deterministic, push down automata.

8.1.1 Structural relation between grammars and push down automata

In general the moves of a push down automaton aren't related to the rules of the grammar that generates the same language recognised by the PDA.

Structural relation Ideally we would like to have a function that maps grammar rules to moves of a PDA. Finding such map allows us to build a **structural relation** between the grammar and the push down automata.

Definition of syntax analyser Structural relation is fundamental for syntax analysers, in fact a syntax analyser is a push down automaton that

- Is in structural correspondence with with a grammar.
- Is able to generate the syntax tree of the recognised strings.

A syntax analyser should be deterministic to be implemented on a computer, in fact an implementation of a non-deterministic automaton would recognise a string in exponential time.

8.2 Naive solution

The solution we are going to describe allows to map a grammar to an automaton in a systematic way but doesn't ensure to create a deterministic automaton. For this reason this is just a starting point.

8.2.1 Mapping rules to moves

To systematically map a grammar to an automaton we have to map the rules of the grammar to the moves of the automaton.

Non terminal rule A rule that contains only non terminals is mapped to a move that

- Reads and pops from memory the non terminal A on the left side of the rule.
- Pushes in reverse order all the non terminals on the right side of the rule.

Practically the rule

$$A \to BA_1 \dots A_n$$

is mapped in the move

```
mem = A
pop
push An, ..., A1, B
```

Terminal rule A rule that contains one terminal on the right side of the rule is mapped to a move that

- Reads and pops from memory the non terminal A on the left side of the rule.
- Reads the terminal form input.
- Pushes in reverse order all the non terminals on the right side of the rule.

Practically the rule

$$A \to bA_1 \dots A_n$$

is mapped in the move

```
mem = A
input = b
pop
push An, ..., A1
```

Empty rule A rule that transforms a non terminal with an empty string is mapped to a move that

-
- Reads and pops from memory the non terminal A on the left side of the rule.

Practically the rule

$$A \to \varepsilon$$

is mapped in the move

```
mem = A
pop
```

RULE	MOVE
$A \to BA_1 \dots A_n$	mem = A; pop; push $A_n, ..., A_1, B$
$A \to bA_1 \dots A_n$	mem = A; input = b; pop; push $A_n, ..., A_1$
$A \to \varepsilon$	mem = A; pop
–	input = ε; mem = Z_0; end
–	input = b; mem = b; pop

Table 8.1: A summary of all the transformations to map a grammar to a PDA.

Final move If the letter ε is read from input and the stack is empty than the PDA stops and accepts the input string.

Terminal letter If the PDA reads in input a terminal letter and reads the same letter from memory then i pops such letter from the stack.

Chapter 9

Syntax analyser

A syntax analyser, also known as a parser, is a machine that

1. Reads from source a string s.
2. If s belongs to $L(G)$
   - Exhibits a derivation (i.e. the steps to obtain the string).
   - Builds a syntax tree. A syntax tree is better than a derivation because it is independent from the derivation order.
3. Else stops execution and notifies the error, possibly with an explanation.

Forest If the string s read from the parser is ambiguous then the parser returns a forest, that is, a set of syntax trees.

Types of parser There exist two types of parsers

- **Bottom-up**. A bottom-up parser expands the rightmost non final node first. This means that the tree is created in inverse order.
- **Top-down**. A top-down parser expands the leftmost non final node first. This means that the tree is created in direct order. A top-down parser can be non deterministic so it might fail (remember that we are working on a computer, which is a deterministic machine). In such cases we have to use bottom-up parsers.

9.1 Extended form grammars

Some grammars might be in extended form. In such cases we have to use multiple automata, i.e. a network of automata, each of which codes a rule of the grammar.

The automata in the network

- Can be non minimal, because we want that the input state to have no incoming arcs.
- Have to be deterministic.

9.1.1 From automata to push down automata

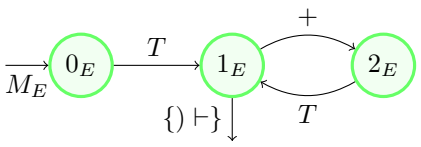
So far we have only talked about networks of automata, but we know that a syntax analyser is a push down automata, thus we need a stack. To jump from an automaton to a push down automaton we can see each automaton as a procedure that can recursively call the another procedures. When a function is called a function activation is pushed on the stack. What we have created is a push down automaton.

Example To better understand how to create a network of automata consider the following grammar

$$E \to T(+T)^*$$
$$T \to F(\times F)^*$$
$$F \to a|(E)$$

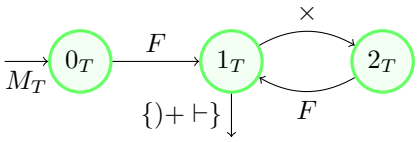
Since we have three rules we can create a network of three automata, one for each rule (9.1, 9.2 and 9.3).

Every time the first automaton should read a non-terminal T, it calls the automaton that recognises the non-terminal T. The same can be said when the automaton T has to recognise the non-terminal F, in fact in this case the it calls the automaton F.



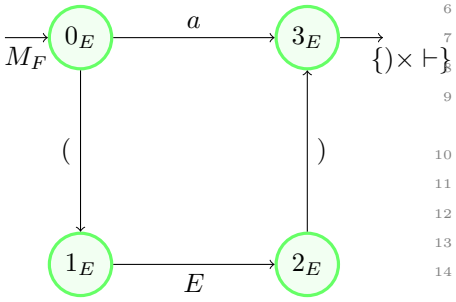

```
1          procedure E:
2              call T
3              while (cc = +):
4                  cc <- next
5                  call T
6              if ( cc in {(, end
       }):
7                  return
8              else error
9
```

Figure 9.1: Automaton generate from rule $E \to T(+T)^*$.




```
1          procedure T:
2              call F
3              while (cc = x):
4                  cc <- next
5                  call F
6              if ( cc in {)+, end}):
7                  return
8              else error
9
```

Figure 9.2: Automaton generate from rule $T \to F(\times F)^*$.




```
1          procedure F:
2              if ( cc = a ):
3                  cc <- next
4              elseif ( cc = ( ):
5                  cc <- next
6                  call E
7                  if ( cc = ) )
8                      cc <- next
9                      if ( cc in
       {)x, end} ):
10                         return
11                     else error
12                 else error
13             else error
14
```

Figure 9.3: Automaton generate from rule $F \to a|(E)$.

Chapter 10

ELR

ELR(k) are a set of bottom-up syntax analysers that depend on the parameter k, a ELRs with bigger k are slightly more powerful. We will consider ELR(1).

Stack The stack contains the read characters and the visited states, in particular we can perform two different moves

- **Shift**. A shift move is performed to push an element on the stack.
- **Reduce**. A reduce move is performed to pop some elements from the stack and replace them with another set of elements.

Followers The followers of a state S are the terminal characters that allow the automaton to end its execution. In practise the followers are represented on the outgoing arrow that represents the end of execution in a final state.

Forward reading The automaton should be able to read in advance the next character in input to decide which move to perform, in fact

- If the next character is in the followers set then the automaton should reduce the stack.
- Otherwise the automaton should shift the stack.

For instance consider the grammar seen in the introduction

$$\begin{aligned} E &\to T(+T)^* \\ T &\to F(\times F)^* \\ F &\to a|(E) \end{aligned}$$

translated using the automata 9.1, 9.2 and 9.3. If such parser would have to recognise the string

$$a \times (a + a)$$

it would execute the following phases

1. The stack is initialised with $0_S, 0_T$ because the string can either start with the axiom or a T non-terminal.

2. When the parser reads a it executes automaton T that executes automaton F and adds $a, \{3_E\}$ to the stack (shift).
$$S = \{\{0_E, 0_T\}, a, \{3_E\}\}$$
3. The parser looks forwards to see $\times$ as next character and because $\times$ is in the followers of automaton F, then the reduce operation is executed. In particular a is popped out from the stack and F is added. The parser now is in state 1_T.
$$S = \{\{0_E, 0_T\}, F, \{1_T\}\}$$
4. The parser shifts character $\times.\{2_T\}$.
$$S = \{\{0_E, 0_T\}, F, \{1_T\}, \times, \{2_T\}\}$$
5. The parser reads character a and calls automaton F.
6. Automaton F adds a to the stack.
$$S = \{\{0_E, 0_T\}, F, \{1_T\}, \times, \{2_T\}, a, \{3_E\}\}$$
7. Automaton F reads forward and because the next character (i.e. $+$) is in the followers of F, then a reduce operation is executed. Character a is popped from the stack and it's replaced with F.
$$S = \{\{0_E, 0_T\}, F, \{1_T\}, \times, \{2_T\}, F, \{1_T\}\}$$
8. The parser is now in state 1_T and because the next character is in the set followers a reduce is executed. In particular characters $F \times F$ are replaced with T. The parser is now in state 1_E and the stack only contains the initial state and the non terminal T.
$$S = \{\{0_E, 0_T\}, T, \{1_E\}\}$$
9. The parser reads character $+$ and adds it to the stack (shift move). The parser is now in state 2_E.
$$S = \{\{0_E, 0_T\}, T, \{1_E\}, +, \{2_E\}\}$$
10. The parser reads character a and goes calls automaton T that calls automaton F.
11. Automaton F adds a to the stack.
$$S = \{\{0_E, 0_T\}, T, \{1_E\}, +, \{2_E\}, a, \{3_E\}\}$$
12. F reads forward the character $\vdash$ and because it is in the set of followers then a reduce operation is executed, in particular a is replaced with F.
$$S = \{\{0_E, 0_T\}, T, \{1_E\}, +, \{2_E\}, F, \{1_T\}\}$$
13. The symbol $\vdash$ is also a follower of T thus another reduce operation is executed.
$$S = \{\{0_E, 0_T\}, T, \{1_E\}, +, \{2_E\}, T, \{1_E\}\}$$
14. Finally $\vdash$ is also in the set of followers of E thus the characters $T + T$ are reduced.
$$S = \{\{0_E, 0_T\}\}$$
15. The stack of the automaton only contains the initial state thus the parser ends the execution.

Shifting multiple states In some cases we have to shift multiple states in the stack because the parser, even being deterministic, can found itself in different states of the automaton. Thus the states on top of the stack represents all the possible states in which the parser can be. When a symbol is read the next step is computed starting from each of the possible states on top of the stack (i.e. the computations from all the states are executed in parallel) and the full derivation can be obtained only after the terminal symbol $\vdash$ is read.

To better understand this concept consider the following grammar

$$S \to aE|E$$
$$E \to (aa)^+$$

that generates strings with an even or odd number of as. The parser is made of two automata (because the grammar has two rules) that are shown in figure 10.1.

Figure 10.1: Automata for multiple states example.

When the parser reads an a, it can go to states

- 1_S if automaton E is not called ($0_S \to_a 1_S$).
- 1_E if automaton E is called ($0_S \to_\varepsilon 0_E \to_a 1_E$).
- 0_E if letter a is read and automaton E is called after looking ahead ($0_S \to_a 1_S \to_\varepsilon 0_E$).

All states are valid thus are added to the stack

$$S = [\{0_S, 0_E\}, a, \{1_S, 1_E, 0_E\}]$$

10.1 Systematic construction

Until now we have built ERL parsers without a proper structured process. Because ELR are used in computers it would be better do describe a process that allows us, given a grammar, to systematically (i.e. algorithmically) build a parser.

10.1.1 Pilot

A pilot is a finite graph that groups and merges the network of automata defined from each rule of the grammar. The pilot also contains the information needed to represent the state (or states, in fact when the automaton is in a state, multiple execution threads might be active) in which the parser can be.

In other words the pilot is a data structure that contains all the information needed by the parser to recognise a string. In particular the pilot contains

- The states in which the parser can be.
- Terminal symbols that are in the followers set of each state.

The pilot drives the push down automaton.

Initials set Given a state q_A of the machine M_A, we call $Ini(q_A)$ the set of initials of the language that we can recognise starting from state q_A.

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

The set of initials can be computed recursively, in particular

- a belongs to $Ini(q_A)$ if there exists a transition from q_A to another node that reads a.

$$q_A \xrightarrow{a} r_A \Rightarrow a \in Ini(q_A)$$

- a belongs to $Ini(q_A)$ if there exists a transition from q_A that calls another automaton B and a is in the initial set of the first state 0_B of automaton B.

$$q_A \xrightarrow{B} r_A \wedge 0_B \xrightarrow{a} 1_B \Rightarrow a \in Ini(q_A)$$

- a belongs to $Ini(q_A)$ if there exists a transition from q_A that calls another automaton B and the language starting from 0_B is nullable (i.e. the empty string can be derived) and a is in the initial set of the first state r_A.

$$q_A \xrightarrow{B} r_A \wedge 0_B \xrightarrow{*} \varepsilon \wedge a \in Ini(r_A) \Rightarrow a \in Ini(q_A)$$

Item An item is an elementary unit of which the pilot is made of. An item contains

- One or more **states** q (i.e. the states in which the parser is).
- and one or more terminal symbols a that represent the followers. Such symbols are called **look-ahead symbols**. Look-ahead symbols are useful when the parser can do multiple moves, in fact such symbols can be used to decide the correct move.

$$\langle q, a \rangle$$

Closure A closure of a set of items C returns the set of items that can be directly reached starting from C. More formally an item $\langle 0_B, b \rangle$ belongs to $closure(C)$ if

$$\begin{cases} \exists \langle q, a \rangle \in C \\ \exists q \xrightarrow{B} r \\ b \in Ini(L(r).a) \end{cases}$$

A state of the pilot contains an item C and its closure.

To better understand what a closure is and how to obtain one, consider the following network (figure 10.2).

The closure of item $C = \langle 0_E, \vdash \rangle$ is

$$\langle 0_E, \vdash \rangle, \langle 0_T, \{a, (, \vdash\} \rangle$$

because

- There exists a transition $0_E \xrightarrow{T} 1_E$ and 0_T is the first state of the automaton T.

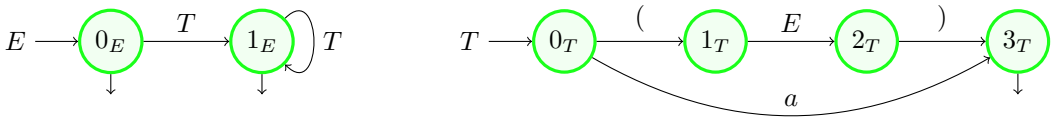


Figure 10.2: The network used for closure example.

- $\vdash$, (and a are initial letters for the language generated by $L(0_E)$, in fact from 1_E we can obtain $(.\Sigma^*$, $a.\Sigma^*$ or ε (because 1_E is also final).

The closure of item $C = \langle 1_T, \vdash \rangle$ is

$$\langle 1_T, \vdash \rangle, \langle 0_E, \{)\} \rangle, \langle 0_T, \{a, (,)\} \rangle$$

because

- There exists a transition $1_T \xrightarrow{E} 2_T$ and 0_E is the first state of the automaton E.
-) is the only initial letter for the language generated by $L(2_T)$ $\vdash$, in fact from 2_T we can only obtain $).\Sigma^*$.
- There exists a transition $0_E \xrightarrow{T} 1_E$, 0_E is in the closure and 0_T is the first state of the automaton T.
-), (and a are initial letters for the language generated by $L(1_E).)$, in fact from 1_E we can obtain $(.\Sigma^*$, $a.\Sigma^*$ or $\varepsilon.)$ (because 1_E is also final but we have to append a) at the end).

Transition function To formalise the pilot's graph we have to introduce a transition function

$$\vartheta(\langle q, a \rangle) = \langle q', a' \rangle$$

that given an item returns the next item. The transition function represents the possible shift operations that can be done by the parser.

If we consider item $C = \langle 1_T, \vdash \rangle$ and its closure $closure(C) = \langle 1_T, \vdash \rangle, \langle 0_E, \{)\} \rangle, \langle 0_T, \{a, (,)\} \rangle$

10.1.2 Pilot construction algorithm

To build the pilot we have to (consider network in Figure 10.2)

1. Start from the initial symbol of the network and add a new item. The new item should have only the empty string in the lookahead. In our example the initial state is 0_E, thus the initial item is $\langle 0_E, \vdash \rangle$.
2. Compute the closure of the initial state. The first item (call it I_0) becomes $\langle 0_E, \vdash \rangle, \langle 0_T, \{\vdash, (, a\} \rangle$.

The initialisation phase is over, now we have to repeat the following steps until we can't all possible transitions and states have been analysed. Consider an item I_n

1. For every state S_i in item I_n we consider all possible outgoing transitions (both final and non-final).

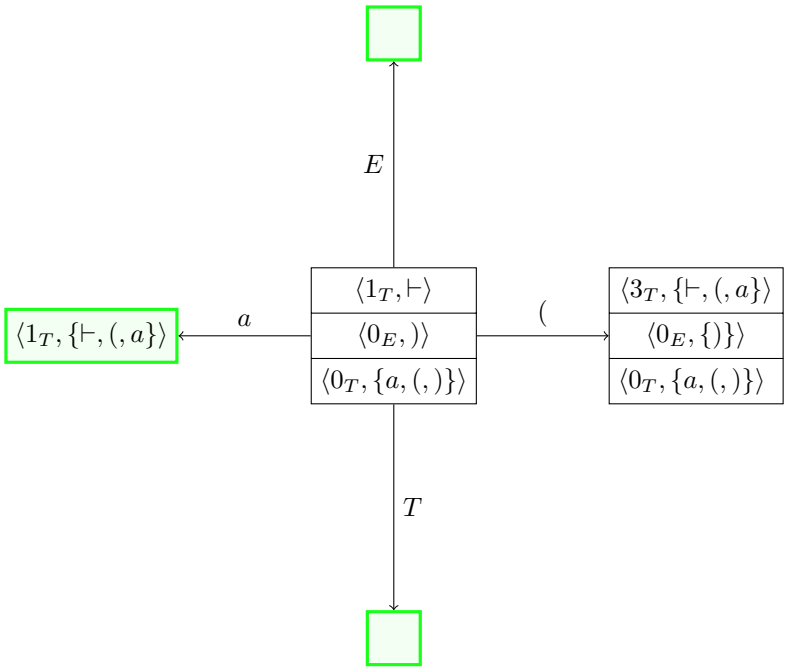


Figure 10.3: Outgoing transitions from a state in the graph.

2. For every transition we create a new item with the state reached by the transition and the same lookahead set. For instance if we consider state 0_T in I_0 we can go to 1_T reading (and to 3_T reading a, thus we build two new items

   - $\langle 1_T, \{\vdash, (, a\}\rangle$
   - $\langle 3_T, \{\vdash, (, a\}\rangle$

   If a transition takes us to an already present we should not build a new item but reuse the one present instead. Two items are equal if they contains the same states and for each states the lookahead sets are the same.

3. Finally we have to compute the closure of all new states created at the previous step.

10.2 ERL conditions

ERL conditions makes determinism possible. We can recognise three different conditions

- Shift-reduce conflict.
- Reduce-reduce conflict.
- Convergence conflict.

If condition 1 and 2 are satisfied (i.e. no conflict exists) then the pilot is deterministic.

10.2.1 Shift-reduce conflict

A pilot contains a shift-reduce conflict when it contains a state where there is a final item with a certain lookahead symbol and from the same state there is an outgoing arc with the same label of the lookahead symbol. In this case the pilot can both reduce or shift.

10.2.2 Reduce-reduce conflict

A pilot contains a reduce-reduce conflict when a state contains two final items that share the same lookahead symbol. In this case the pilot can perform multiple reductions but the input tape can't help in deciding which reduction to execute.

10.2.3 Convergence conflict

A convergence conflict happens when two items go to the same item reading the same symbol. Usually conflicts (i.e. multiple transitions are possible from a state) can be solved using lookahead sets, thus a convergence conflict only happens when two items have the same lookahead sets.

The only way to solve convergence conflicts (that lead to non-determinism) is to replace the grammar with an equivalent one that doesn't lead to non-determinism. Furthermore if a grammar is ambiguous then the pilot's graph will surely have a conflic.

10.2.4 Conflicts in graph and network

If a conflict is present in the pilot graph, then the conflict is also present in the network. The same isn't true in the opposite direction.

$$\text{conflict in graph} \Rightarrow \text{conflict in network} \qquad \text{conflict in network} \not\Rightarrow \text{conflict in graph}$$

10.3 Complexity

Time and space complexity for deterministic parsers is linear with respect to the length of the input string. In particular let us call

- n_T the number of terminal shifts.
- n_N the number of non-terminal shifts.
- n_R the number of reductions.

The number of non-terminal shifts is equal to the number of reductions

$$n_N = N_R$$

in fact after every non-terminal shift a reduction is performed. The number of terminal shifts is the number of characters n in the input string

$$n_T = n$$

If we sum the total number of operations we obtain

$$n_{tot} = n_T + n_N + n_R = n + 2n_R$$

Furthermore the number of reductions is always linearly bounded to the letters in the input string n.

$$n_{tot} = n + 2kn = (2k + 1)n = kn + C \quad \Rightarrow \mathcal{O}(n)$$

Finally space complexity is always smaller than time complexity, thus it is also linear.

Chapter 11

ELL

ELL(k) is a family of top-down syntax analysers that doesn't need a pilot (thus a graph) like ELR but can be driven by directly by the net. For this reason ELL is a simplified version of ELR.

We are going to analyse ELL(1) like we did for ELRs and we are simply going to call it ELL. As for ELR the parameter k represents how many forward transitions we can consider for the lookahead set. In $k = 1$ we consider only the immediate transition from a state to the successive one.

Top-down approach ELL build the syntax tree using a top-down approach (opposite to ELR that used a bottom up approach). This means that the tree is built starting form the root and it's expanded to the leafs.

Abstract behaviour ELL uses directly the network build with the rules' automaton. In particular ELL calls one automaton of the net (like in procedure calling, in fact we said that each automaton can be represented as a procedure) and the procedure called is added to the stack.

Multiple threads ELR was capable to follow multiple threads because of the graph. On the other hand ELL follows the network thus it can only follow one thread. If we consider the procedure calling abstraction we can say that a program can only have a single thread (because processes defined from automaton can't create threads).

Power ELL is strictly less powerful than ELR

$$ELL <_{power} ELR$$

and if a the ELL analyser is deterministic also the corresponding ELR (i.e. the ELR built from the same network) is also deterministic.

$$\text{ELL deterministic} \Rightarrow \text{ELR deterministic}$$
$$\text{ELL deterministic} \not\Leftarrow \text{ELR deterministic}$$

In other words if a grammar is ELL then it is also ELR but the opposite is not always true.

$$\text{ELL grammar} \Rightarrow \text{ELR grammar}$$
$$\text{ELL grammar} \not\Leftarrow \text{ELR grammar}$$

Another important property is that a language is ELR if it admits at least one (among all infinite grammars) ELR grammar. We can also add that any deterministic language is ELR-1 but there exists languages that have a deterministic automaton but that are not ELL (e.g. $a^n b^m : 0 \leq m \leq n$).

11.1 ELL condition

Initially we said that ELL is a simplified version of ELR. This means that there exists some condition that a grammar (or a network) has to verify so that the ELL is deterministic.

Generally if the ELR graph has only single transitions then the ELL can be built, but it can't be built if there is a leftmost recursive (direct or indirect) derivation. More formally the ELL conditions are

- There are **no leftmost derivations**.
- The **ELR pilot satisfies the first two ELR conditions**.
- The pilot graph **doesn't have multiple transitions**.

This means that before building a ELL syntax analyser we have to

1. Build the ELR pilot graph.
2. Verify the ELL conditions.
3. Get rid of the pilot and build the ELL analyser.

11.2 ELL construction

Example To better understand how ELL construction works lets introduce the following grammar

$$\Sigma = \{(, a,)\}$$
$$E \to T^*$$
$$T \to (E)|a$$

The network generated from such grammar is shown in figure 11.1.

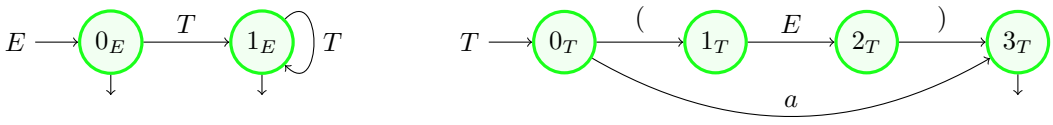


Figure 11.1: The network of the example ELL grammar.

PCFG To build an ELL analyser we have to build a so called PCFG that connects the automata in the network. In particular we draw a connection between two states if the former transfers the computation to the latter. In other words to automata are connected if the procedure of an automaton calls the procedure of the other one. The PCFG is shown in figure 11.2

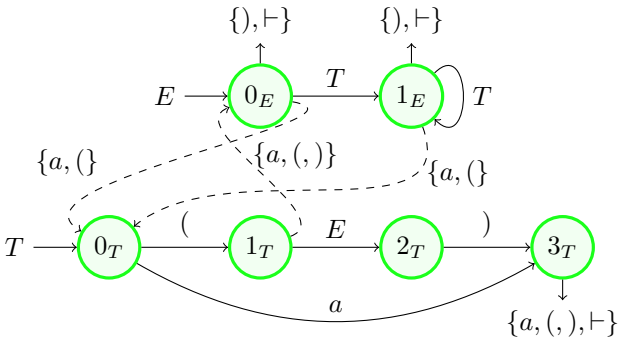


Figure 11.2: The PCFG of the example ELL grammar.

Stack ELL uses a push down automaton to parse the input string, thus we need a stack. In particular the stack of an ELL analyser is described as follows

- It contains the states of the automata in the net (i.e. a stack character for each state).
- It is initialised with the initial state of the net.

Guide set At each step we might be able to activate multiple transitions (i.e. if a is read go to state 1_S, if b is read go to state 2_S). In general this is not a problem, unless

- One transition is activated by a **non-terminal** character.
- One of the transitions is a an **end transition**.

In such cases we need a tie-breaking rule, in particular we can use a GuideSet. A GuideSet is the set of characters that are expected in the input soon after calling the machine. The GuideSet of a state is usually represented on the arc exiting from such state. For instance let us consider transition $0_E \to 0_T$. In this case machine E calls machine T only when a or (are read, in fact the only transition exiting from 0_T are labelled with a and (. It's also interesting to analyse the transition $1_T \to 0_E$, in fact in this case the GuideSet has to contain also) because 0_E is a final state, thus T could call E, end immediately and then read) (i.e. the only letter that can be read in 2_T).

Final states Final states are also critical, in fact when an automaton reaches an end state it might have to

- Return to its caller.
- End execution.

To handle final states we can use the ProspectSet that contains all the letters that have to be read to reduce (i.e. to return to the caller) . The prospect set is written on the exit arrows. For instance if we consider 0_E, if the next character in the input string is $\vdash$ then execution has to be returned to the caller. Execution has to be returned even if the next character is a), in fact T called E when it was in state 1_T and the next state of T (i.e. 2_T) can only read), thus if the next character in input is) we must go to 2_T, thus return to caller.

11.2.1 Moves

An ELL syntax analyser can execute 4 moves

- **Call** move.
- **Shift** move.
- **Reduce** move.
- **Recognition** move.

Call move When an automaton calls another automaton

1. A letter is popped from the stack.
2. The next state of the caller is pushed on the stack.
3. The first state of the called is pushed on the stack.

Normal move When a normal move is executed (i.e. a terminal letter is read from the input string)

1. A letter is popped from the stack.
2. The next state of the current automaton is pushed on the stack.

Reduce When an end letter is found (i.e. the automaton has reached a final state)

1. A letter is popped from the stack.

We only pop a letter from the stack because after popping it the state on top of the stack is the next state of the caller, i.e. the state from where execution has to restart.

Recognition move A ELL analyser recognises a string when

- The string is over.
- A final state is on top of the stack.

11.2.2 Computing the GuideSet

Consider the portion of a network in figure 11.3. The GuideSet of $q_A \to 0_{A1}$ can be formally computed as follows

$$Gui(q_A \to 0_{A1}) = \bigcup \begin{cases} Ini(L(0_{A1})) & \\ Ini(L(r_A)) & \text{if } Nullable(A1) \\ Gui(r_A) & \text{if } Nullable(A1) \wedge Nullable(L(r_A)) \end{cases}$$

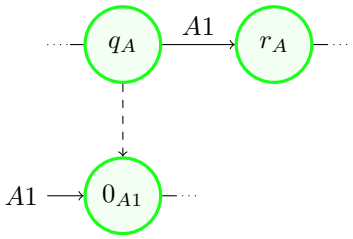


Figure 11.3: A portion of a network.

Chapter 12

Earley tabular syntax analyser

An Earley tabular syntax analyser allows us to recognise all types of grammars (even ambiguous one). This comes at a cost, in fact the driver isn't a simple push down automaton but it's a full computer (with a CPU and a RAM) and the time complexity is much higher than ELR and ELL analysers.

Usage Earley analysers aren't very much used in artificial languages because they are usually not ambiguous.

Parallel threads Earley analysers, like ELR ones, are able to handle multiple threads of execution in parallel in case there exists multiple path. In this case a thread isn't stopped when it has reached the reduce point.

12.1 Earley vector

Earley analysers use a vector E called Earley vector. The dimension of E is $n + 1$ where n is the number of characters in the input tape.

Item Each element of the vector is a list of **items**. Each item is a couple made of

- A **state** of the net.
- A **pointer** to an element of the vector (i.e. an index from 0 to n). An index also represents the epoch in which a computation thread has started.

Closure After adding items to an element we have to compute the closure of every item (iteratively, thus we also have to compute the closure of the elements obtained from the closure). The closure is computed as for ELR analysers. An element can therefore be divided in two sublists

- The **base** (i.e. the items added to the element).
- The **closure** (i.e. the items obtained with the closure of the base).

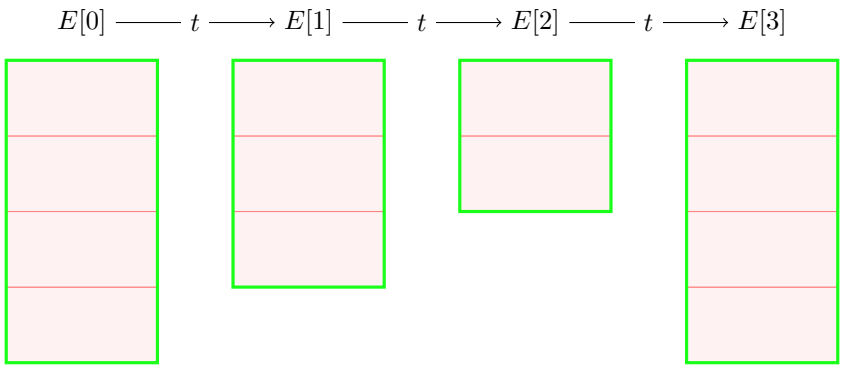


Figure 12.1: An example of Earley vector (elements in green, items in red).

String recognition If the algorithm can build the Earley vector and the last element contains only items with final states with pointers to $E[0]$ than the input is completed.

Initialisation The Earley vector is initialised with the initial states of the machine. We also have to remember to compute the closure of the base.

12.1.1 Moves

An Earley syntax analyser can do 4 moves

- A **terminal shift**.
- A **closure** executed like in ELR analysers.
- A **non-terminal shift** or reduction.
- A **completion** (i.e. a sequence of closure and non-terminal shift).

Input read - terminal shift When a letter is read from input we have to

1. Add a new element to E.
2. Add items to the new element. In particular every item added contains the state reached by the pilot after reading the letter in input and a pointer to the element that holds the origin state of the transition. For instance if there exists a transition between 1_A and 2_A and 1_A is in $E[1]$, then we have to add $< 2_A, 1 >$ to $E[2]$. Once an item is created it can't be changed. The new item has the same pointer of the source because the thread is the same of the former pointer (thus the computation has started in the same epoch).
3. Compute the closure of the items added in the previous step.

For instance consider that we currently are in $E[1]$ and we read a from input. In such case we add an element to E and for each item I in $E[1]$ if it has an outgoing transition labelled a then we add the destination state to $E[2]$.

Reduction When the state of an item I is final, we have to perform a reduction. In particular during a reduction operation we have to

1. Follow the pointer of item $I =< m_Y >$ to the referenced element.
2. Consider the element reached at step 1. Find the item(s) J in which the state (say n_X) has an outgoing transition labelled with the same automaton of I's state (i.e. labelled Y). For instance if $I =< 3_A, 2 >$, then we have to go to $E[2]$ and search for items in which the state has a transition labelled A.
3. Add an item with the successive state of n_X (say n'_X) and the index of item J. In our example if $J =< 0_S, 0 >$ then we add item $< 1_S, 0 >$ to the element in which I is.

Non-terminal shift When we add a new item, if such item has an outgoing non-terminal transition then we have to reduce it. The reduction works as for ELRs, in particular we have to compute the states to which the computation can be handled.

When we perform a reduction, the pointer of the new item has the same index of the element in which it is (e.g. an item created via reduction in element $E[1]$ has index 1). Performing a reduction identifies the fact that a new computational thread has started (this is why we use the current epoch).

12.2 Example

Consider the following grammar

$$\begin{aligned} S &\to A|B \\ A &\to aAb|ab \\ B &\to aaBb|aab \end{aligned} \tag{12.1}$$

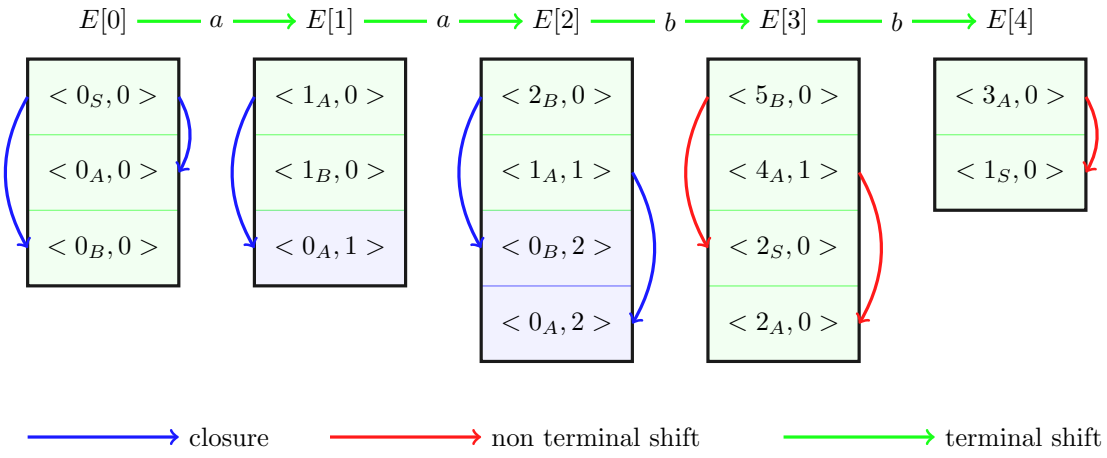


Figure 12.2: An example of Earley syntax analyser for grammar 12.1 and input *aabb*.

12.3 Complexity

The Earley analyser is much slower than ELL and ELR analysers, in fact it has a

- Cubic time complexity with respect to the input string length.
- Quadratic time complexity with respect to the input string length.

Notice that Earley method succeeds in keeping the complexity polynomial even if both complexities are far worst than the ones of previously studied analysers.

Space complexity Early has a quadratic space complexity. In particular the space complexity refers to the space occupied by the Earley vector. To compute the dimension of E we can notice that

- E has a length of $n + 1$.
- Every element of E has a length of kn where k is the number of states (i.e. a constant).

The space complexity can be written as

$$\mathcal{O}(n \cdot (kn)) = \mathcal{O}(kn^2) = \mathcal{O}(n^2))$$

Time complexity Early has a cubic time complexity with respect to the input length n

$$\mathcal{O}(n^3)$$

This complexity is given by the fact that

- The time complexity is always bigger than the space complexity, which for the Earley algorithm is n^2.
- In the worst case we reduce every item of every element of E, thus for every item we have to search an item in a vector (we have the index of the element but not of the item in the vector). This operation has a cost of kn (k being the constant that represents the number of states).

If we repeat a search of cost kn for kn^2 times we obtain a complexity of

$$\mathcal{O}(kn \cdot kn^2) = \mathcal{O}(n^3)$$

Part IV

Translation and semantics

Chapter 13

Introduction

The process that allows to transform a string from a language to another language is called translation. Translation is a fundamental part of a compiler, in fact it has to translate a program written in some programming language in a machine language executable code.

In general the compilation process can be divided in two stages

1. In the first stage the program is analysed, verified and translated to an intermediate language. Sometimes the syntax tree is generated.
2. In the second stage the intermediate language is translated in machine code applying
   - Verification.
   - Optimisation. In this phase the code is optimised considering the processor architecture.
   - Scheduling. Scheduling allows to reorder the instruction to exploit pipelining.

Intermediate language In the description of a compiler we will consider an intermediate language made of

- Scalar variables and constants.
- Assignments.
- Arithmetic and logic operations.

Chapter 14

Static flow analysis

Static flow analysis (SFA) allows us to verify if some piece of code (in general if a string) has some properties. In particular with SFA we can attach some true or false statements to a program. Static flow analysis doesn't rely on syntax analysis, in fact it's based on a regular language, thus it only needs a finite state automaton to be executed. Static flow analysis is executed on the intermediate language.

14.1 Control flow graph

As we said at the beginning, static flow analysis is executed by a finite state automaton, thus we have to build it. The automaton used for static analysis is called Control Flow Graph (CFG). The control flow graph is generated starting from the intermediate code generated by stage 1 of compilation and

- Each instruction is represented on a node.
- If instruction a is executed immediately after instruction b, then a directed arrow is drawn from b to a.

The Control Flow Graph can be further simplified considering only if a variable is used or assigned. The Control Flow Graph of the code in intermediate language in Listing 14.1 is shown in Figure 14.1, while the simplified CFG is shown in Figure 14.2.

```
1       a := 1
2  e_1: b := a + 2
3       c := b + c
4       a := b * 3
5       if a < m goto e_1
6       return c
```

Listing 14.1: An example of code in intermediate language.

Finite graph The Control Flow Graph is finite because a program is made of a finite number of statements, thus it's possible to define a regular language in which every node of the graph is represented with a different symbol (i.e. a CFG with n nodes generates a language with n symbols) of the input alphabet.

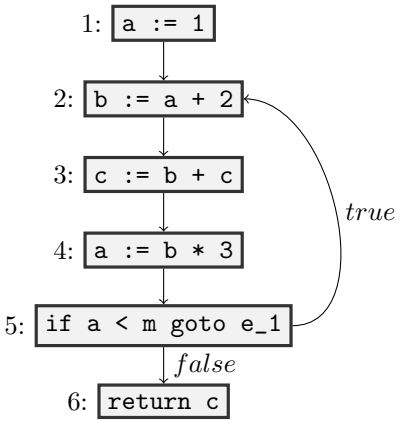


Figure 14.1: The Control Flow Graph of code 14.1 (the numbers on the left side represent the symbols of the alphabet).

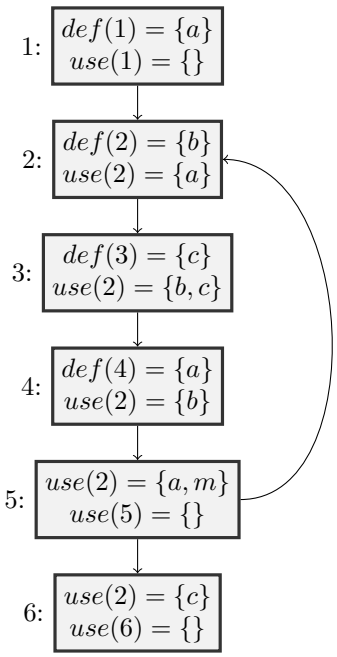


Figure 14.2: The simplified Control Flow Graph of code 14.1.

Conservative approximation Some instructions of the program might be unreachable, but we should insert them in the CFG anyways. This approach is called conservative approximation.

14.1.1 Definition and use sets

For each node p of the CFG (i.e. for each statement of the program) we can define

- $def(p)$. The set of variables defined in p.
- $use(p)$. The set of variables used in p.

14.2 Liveliness of a variable

One important property that can be verified on a CFG is the liveliness of a variable at a certain point of the program. A variable a is alive at point (i.e. at statement, or at node) p if the CFG has a path from p to another point q and the following conditions hold

- The path $P : p \to q$ doesn't pass through any instruction r ($r \neq q$) that defines a.

$$\nexists r \neq q : r \in P \wedge a \in def(r)$$

- Instruction q uses variable a.

$$a \in use(q)$$

In other words the variable a is alive at point p if the assignment at p influences what happens at q.

14.2.1 Liveliness intervals

In the context of alive variables we can define if an interval is alive. Say

- I is a set of instructions.
- $D(a)$ is the set of instructions in which a is defined.
- $U(a)$ is the set of instructions in which a is used.

Variable a is alive at the output of instruction p if in the language L of the CFG exists a valid phrase $x = upvqw$ such that

$$u, w \in I^* \wedge p \in I \wedge v \in I \setminus D(a)^* \wedge q \in U(a)^*$$

The set of strings x that satisfy the condition aforementioned is a regular language $L_p \subseteq L$.

We can use L_p to determine if a is alive at point p, in particular if L_p is non empty then a is alive at point p.

$$L_p \neq \emptyset \Rightarrow a \text{ alive at } p$$

Flow equation method

Checking if L_p is non empty can be expensive, thus we can use the flow equation method to compute if variable a is alive at point p. The flow equation method uses a backward propagation algorithm to determine the interval in which a variable is alive. In particular the algorithm computes, for every instruction p

- The set $\text{live}_{in}(p)$ of variables alive at the input of p.
- The set $\text{live}_{out}(p)$ of variables alive at the output of p.

Given a node p

- If p is final (i.e. has no outgoing arcs) then the set of variables at its output is empty.

$$\text{live}_{out}(p) = \emptyset \quad \forall p \text{ final}$$

- The set of active variables at the output of p is the union of the sets of active variables of the successors of p

$$\text{live}_{out}(p) = \bigcup_{q \in Suc(p)} \text{live}_{in}(q)$$

Furthermore we need to define how to transform the output set of a node in the input set of the same node. In other words we have to consider the effect on instruction p. The set of variables alive at the input of p is the union of

- The variables used in p.
- The variables alive at the output of p that haven't been defined in p. The variables defined in p have to be removed because the definition of a variable breaks the liveliness of a variable.

$$\text{live}_{in}(p) = \text{use}(p) \cup \big(\text{live}_{out}(p) \setminus \text{def}(p)\big)$$

Usages

Liveliness intervals can be used to optimise memory allocation, in fact registers are a scarce resource, thus they have to be handled efficiently. In particular if two variables aren't alive in the same interval then they can be assigned to the same register.

14.2.2 Reaching definition

The definition of a variable a in instruction q reaches the input of an instruction p (even with $p = q$) if there is a path from q to p so that its nodes don't define a again. In other words a_q reaches the input of p if $L \subseteq I^*$ is a phrase $x = uqvpw$ so that

$$u, w \in I^* \wedge q \in D(a) \wedge v \in \big(I \setminus D(a)\big)^* \wedge p \in I$$

Flow equation method

The reaching definitions can be computed using the flow equation method. In this case we have to apply a forward algorithm.

Before describing the algorithm we have to define the set $sup(p)$ of the definitions suppressed by an instruction p as the set of instructions a_q that have been defined in q and in p.

$$sup(p) = \left\{a_q | q \in I \wedge q \neq p \wedge a \in def(q) \wedge a \in def(p)\right\} \text{ if } def(p) \neq \emptyset$$

The algorithm computes, for each instruction p

- The set $in(p)$ of variables defined at the input of p.
- The set $out(p)$ of variables defined at the output of p.

Given an instruction p

- If p is the initial node then the set of variables defined at the input of p is empty (if we assume the program has no parameter unlike `argc, argv` in C).

$$in(p) = \emptyset$$

- The set of variables defined at the input of p is the union of the sets of variables defined at the output of the immediate predecessors of p.

$$in(p) = \bigcup_{q \in Pred(p)} out(q)$$

- The set of defined variables at the output of p is the union of the set of variables defined in p and the set of variables not suppressed in p.

$$out(p) = def(p) \cup \big(in(p) \setminus sup(p)\big)$$

Chapter 15

Syntax directed translation

15.1 Introduction

Syntax directed translation (also called transduction) allows to easily transform a string of a language L_1 in a string of another language L_2 and, given two strings of L_1 and L_2 to check if $y \in L_2$ is a translation of a string $x \in L_1$. Syntax directed translation is based on grammars, thus it simple and easy to define and use, but it can recognise and apply only some kind of translations.

15.1.1 Relation

Formally, given two alphabets Σ and Δ a translation is a correspondence of strings from Σ^* (called **source strings**) to Δ^* (called **destination strings**). We can also define a transduction as a binary relation ρ between strings $x \in \Sigma^*$ and $y \in Delta^*$

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

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

In this context we can define L_1 and L_2 as the projections of the translation ρ over the first and second component of the relation

$$L_1 = \{x \in \Sigma^* | \text{ for some string } y \text{ such that } (x, y) \in \rho\} \tag{15.1}$$

$$L_2 = \{y \in \Delta^* | \text{ for some string } x \text{ such that } (x, y) \in \rho\} \tag{15.2}$$

15.1.2 Function

A translation can be seen as a function τ

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

where $\wp(\Delta^*)$ is the power set of Δ^* (i.e. all the possible subsets of Δ^*). Namely τ can be written as

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

The function τ can also be used to represent the language L_2 using L_1

$$L_2 = \tau(L_1)$$

Properties

As for any function, we can define the usual properties of τ as follows

- τ is **injective** if any two different source strings are mapped on different destination strings. In other words any destination string can be the image of at most one source string.
- τ is **suriective** if every destination is the image of at least one source string.
- τ is **bijective** if it is both injective and surjective. In other words τ is bijective if there exists a one-on-one correspondence between source and destination strings (i.e. every source string is mapped to one and only one destination string).

Furthermore the function τ is

- **Total** if every source string has an image of one or more destination strings.
- **Partial** if it's not total, that is if some source strings don't have an image (i.e. can't be translated).
- **One-valued** if every source string has an image of at most one destination string.
- **Multi-valued** if some source strings may have an image of two or more destination strings.

15.2 Regular transduction

Regular transduction uses generalised regular expressions. In particular regex have to be extended in order to work on two languages in parallel.

A generalised regular expression works on two alphabets Σ (say the source alphabet) and Δ (say the destination alphabet). To better understand how regex are generated let us consider two alphabets

$$\Sigma = \{1, /\}$$
$$\Delta = \{1, :, \div\}$$

And the following regex

$$1^*(/(1)^*)^*$$

that can be translated in one of the following regular expressions in Δ

$$1^*(: (1)^*)^* \qquad\qquad 1^*(\div(1)^*)^*$$

This means that a string $x = 3/5/7$ of Σ^* can be translated with $y_1 = 3 : 5 : 7$ or $y_2 = 3 \div 5 \div 7$ in Δ^*.

A generalised regular expression maps a source symbol $s_\Sigma \in \Sigma$ in one or more output symbols $s_\Delta \in \Delta^*$. The mapping is expressed putting s_Σ over s_Δ separated by a horizontal bar. This means that a symbol σ of the extended regular expression is made putting a symbol of Σ over a string of Δ^*.

$$\sigma = \frac{s_\Sigma}{s_\Delta}$$

For instance if we want to translate the symbol / in : we should write $\frac{/}{:}$. The symbol σ can also be written as follows

$$\sigma = (s_\Sigma, s_\Delta)$$

The translation from Σ^* to Δ^* can be written composing the symbols σ as in normal regular expressions (i.e. using $*$, $+$, union, intersection and so on). In our example a string in Σ^* can be translated in a string of Δ^* using the following extended regular expression

$$\left(\frac{1}{1}\right)^*\left(\left(\frac{/}{:}\right)\left(\frac{1}{1}\right)^*\right)^* \cup \left(\frac{1}{1}\right)^*\left(\left(\frac{/}{\div}\right)\left(\frac{1}{1}\right)^*\right)^*$$

or equivalently using

$$(1,1)^*\Big((/,:)(1,1)^*\Big)^* \cup (1,1)^*\Big((/,\div)(1,1)^*\Big)^*$$

Notice that we can interpret the translation regex as two separated regex (the upper and lower one) that must have the same structure (and of course can have different symbols).

15.2.1 Recognition

Because regular transaction is based on regexs, we can use a finite state automaton (FSA) for recognition and translation. Let's start with recognition. To allow a FSA to check if a destination string is the correct translation of a source string we need to slightly modify a FSA. In particular the new FSA has two input tapes, one for the source string and one for the destination string, preloaded with the source and destination strings (e.g. $3/5/7$ and $3:5:7$). The automaton can read from both tapes and the transitions are labelled using the symbols $\sigma = \frac{s_\Sigma}{s_\Delta} = (s_\Sigma, s_\Delta)$. A part of the automaton is shown in Figure 15.1.

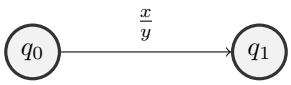


Figure 15.1: A part of the automaton used to recognise a translation.

More formally the FSA can be defined as a tuple

$$\langle Q, \Sigma, \Delta, \delta, F\rangle$$

where

- Q is the set of states of the automaton.
- Σ is the source alphabet.
- Δ is the destination alphabet.
- $\delta : Q \times \Sigma \times \Delta^* \to Q$ is the transition function that given a state q, a letter $s \in \Sigma$ and a string $d \in \Delta^*$ returns the next state. Notice that the transition function can read multiple symbols from the destination tape because a source symbol can be translated with multiple output symbols.
- $F \subseteq Q$ is the set of final states of the automaton.

When the automaton is in state q_0 and reads x in tape 1, y in tape 2 then it goes to state q_1. If the state reached by the automaton is final and both tapes are over then the destination string is a valid translation of the source string.

Determinism

Determinism for a translation recogniser is defined in the same way as for normal FSA, thus it's always possible to transform a non-deterministic automaton in a deterministic one.

Furthermore we can notice that an automaton can be transformed in a grammar of right unilinear type in which

- The non terminal symbols are the states of the automaton.
- The terminal symbols are the translation symbols $\frac{s_\Delta}{s_\Sigma}$ used in the automaton.

For instance the automaton in Figure 15.2 can be translated as

$$S \to \frac{a}{\varepsilon} Q_1$$
$$Q_1 \to \frac{a}{a} Q_2$$
$$Q_2 \to \frac{a}{\varepsilon} Q_1 | \frac{\varepsilon}{\varepsilon}$$

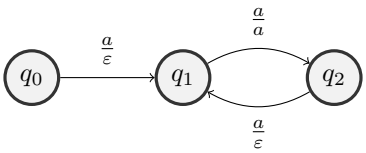


Figure 15.2: The FSA that can be transformed in a grammar.

15.2.2 Translation

The same automaton used for translation recognition can also be used to translate the source, in this case the first tape is loaded with the string to translate and the second tape is empty. Given the fact that the second tape is empty, the automaton works a little differently, in particular, if we consider the automaton in Figure 15.1, when it is in state q_0 and reads x in input, it goes to state q_1 and writes y in output.

Determinism

Determinism in translator finite state automata is critical, in fact the automaton might read the same letter from source and translate it in two different ways, thus it can't decide which transition to fire. It's important to underline that a translator might be non-deterministic even if the same automaton used as recogniser is deterministic. This happens because the recogniser can read a letter from both the source and the destination tape. To better understand this concept let us consider the automaton in Figure 15.3. The automaton, when working as a translator, is non deterministic in fact if it reads a from the input tape, it can go in two directions (i.e. to q_1 and to q_2). On the other hand, the same automaton, used as recogniser, is deterministic because it reads both the source and destination tape and depending on the letter read on the latter, only on transition is fired. This example shows that it's not always possible to obtain a deterministic translator regular translator.

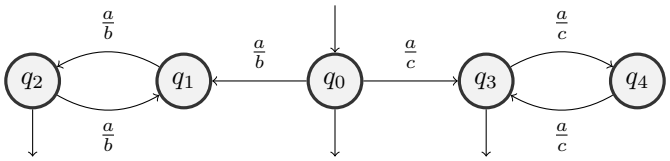


Figure 15.3: A non-deterministic translator.

15.3 Context free transduction

15.3.1 Recogniser

Regular translators can't translate all languages, because they are based on regular languages. For instance a translator can't invert a string ($x \to x^R$), thus we need more powerful translators. Context free translators are based on grammars thus are more powerful than regular translators. In particular context free translators are based on an extended grammar that with two sets of terminal symbols Σ and Δ (source and destination). Such sets aren't used directly in the grammar, in fact we use symbols γ in the set $C = \Sigma \times \Delta$. Each symbol $\gamma \in C$ is represented as for regular grammars

$$\gamma = \frac{s}{d} \qquad\qquad \gamma = (s, d)$$

where $s \in \Sigma^*$ and $d \in \Delta^*$.

The symbols in C can be used and combined with non-terminal symbols as in normal grammars. For instance the following grammar inverts a string of as and bs.

$$S \to \frac{a}{\varepsilon} S \frac{\varepsilon}{a} \;\Big|\; \frac{b}{\varepsilon} S \frac{\varepsilon}{b} \;\Big|\; \frac{\varepsilon}{\varepsilon} \tag{15.3}$$

Notice that we can distinguish two different grammars (one above and the other below) that must have the same structure but can have different terminal symbols. This fact can be seen also in the syntax tree that must have the same inner nodes (i.e. same non-terminal symbols). The upper and lower grammars of our examples are

$$S \to aS\varepsilon | bS\varepsilon | \varepsilon \qquad\qquad S \to \varepsilon Sa | \varepsilon bS | \varepsilon$$

As we can see from this example the two rules have the same structure but different terminal symbols (in particular the terminal are inverted).

Given the grammar 15.3 we can obtain a derivation as for normal grammars. When the string is made only of terminal symbols γ, the source string is represented above the line and the destination string is represent below. An example of derivation is

$$S \to \frac{a}{\varepsilon} S \frac{\varepsilon}{a} \to \frac{a}{\varepsilon}\frac{a}{\varepsilon} S \frac{\varepsilon}{a}\frac{\varepsilon}{a} \to \frac{a}{\varepsilon}\frac{a}{\varepsilon}\frac{b}{\varepsilon} S \frac{\varepsilon}{b}\frac{\varepsilon}{a}\frac{\varepsilon}{a} \to \frac{a}{\varepsilon}\frac{a}{\varepsilon}\frac{b}{\varepsilon}\frac{a}{\varepsilon} S \frac{\varepsilon}{a}\frac{\varepsilon}{b}\frac{\varepsilon}{a}\frac{\varepsilon}{a} \to \frac{a}{\varepsilon}\frac{a}{\varepsilon}\frac{b}{\varepsilon}\frac{a}{\varepsilon}\frac{\varepsilon}{\varepsilon}\frac{\varepsilon}{a}\frac{\varepsilon}{b}\frac{\varepsilon}{a}\frac{\varepsilon}{a} \to \frac{aaba}{\varepsilon\varepsilon\varepsilon\varepsilon} \frac{\varepsilon\varepsilon\varepsilon\varepsilon}{abaa}$$

Ambiguity

In practice the relations between the source and the destination should be one-to-one, but in general it can also be one-to-many. In other words in practical application is desirable to translate a string with a single string but it is also allowed to have multiple translation for the same string.

If the source grammar is ambiguous then the source tree might have different frontiers (i.e. leaf nodes), thus multiple translations.

Another important thing to notice is that the translation from an alphabet Δ to another alphabet Σ might be ambiguous, even if the translation from Σ to Δ is non-ambiguous.

15.3.2 Translator

The translator for context-free translators can be obtained as for regular translators. In particular the destination tape must be empty and the automaton, when a translation is fired, must write on the destination tape the lower part of the label's symbol.

Chapter 16

Attribute grammars

Attribute grammars offer a general model to implement complete transition in a general way. Attribute grammars also allow to analyse and translate features that cannot be expressed with a grammar (e.g. the length of a string). Attribute grammars use the syntax tree obtained by syntax analysis, thus even if a grammar is ambiguous we can still use attribute grammars because we only need one tree, no matter how it has been found. Furthermore if a grammar is ambiguous, all syntax tree produce the same translation.

Example To better understand how attribute grammars work let us consider the translation from binary rational numbers to decimal rational numbers. In other words we want to translate a string from

$$L_2 = 0,1^* \ '.' \ 0,1^*$$

to

$$L_{10} = [0-9]^* \ '.' \ [0-9]^*$$

where the . separates the decimal part from the integer part. For instance the string 10.01_2 is translated into 2.25_{10}.

16.1 General structure

Attribute grammars use

- A BNF **support grammar**.
- **Syntax functions** associated to the rules of the support grammar. In particular we can assign to each rule one or more functions (or even no function) that expresses how to translate the grammar rule.

16.1.1 Syntax functions

A syntax function is a function that assigns the value of an expression to an attribute (i.e. a variable). In principle the function can be whatever, in fact it can also be a piece of pseudocode.

$$var := expression$$

Attributes

Attributes grammars start from a syntax tree. In particular the process starts from the leaves and propagates through the root. When we consider a node (say not a leaf node), it has some children that have already been translated (i.e. whose syntax function have already been executed). Attribute grammars use variables (also called **attributes**) that get assigned at each node of the syntax tree using the variables assigned at the child node. When we reach the root node, the value of the attributes is used as translation.

Usually we use a subscript to indicate if we reference a variable from the node or from the children. In particular if we consider attribute a, we can use a_0 to reference the node variable and a_n with $n > 0$ to reference the values of the children.

Attributes can be divided in

- **Left attributes** (also node attributes or synthesised attributes). Left attributes are associated to the node D_0. A function can assign only assign values to the synthesised attribute of the node itself. In other words, if s is a synthesised attribute, we can write $s_0 := expression$ but we can't write $s_1 := expression$ or use s_0 in an expression.
- **Right attributes** (also child attributes or inherited attributes). Right attributes are associated to the child nodes D_n ($n \geq 1$). A function can only assign values to the inherited attributes of the child nodes. In other words, if i is an inherited attribute, we can write $i_2 := expression$ but we can't write $i_0 := expression$. It's also possible to use i_0 in the expression of any function.

Example

To better understand how attributes are used in attribute grammars we can write the support grammar and the relative function for the binary to decimal translation. In particular the attribute grammar is written in table 16.1. As we can see we have only used synthesised attributes.

rule	function 1	function 2
$N \to D.D$	$v_0 := v_1 + v_2 \cdot 2^{l_2}$	
$D \to DB$	$v_0 := 2 \cdot v_1 + v_2$	$l_0 := l_1 + 1$
$D \to B$	$v_0 := v_1$	$l_0 := 1$
$B \to 0$	$v_0 := 0$	
$B \to 1$	$v_0 := 1$	

Table 16.1: Attribute grammar for the binary to decimal translation.

Decorated tree To visualise how a string is translated we can decorate the syntax tree using the attributes v and l. The decorated tree is shown in Figure 16.1.

16.1.2 Formal definition

An attribute grammar can be formally (or algorithmically) defined as follows

1. Let $G = (V, \Sigma, P, S)$ be a syntax where
   - V is the set of nonterminals.

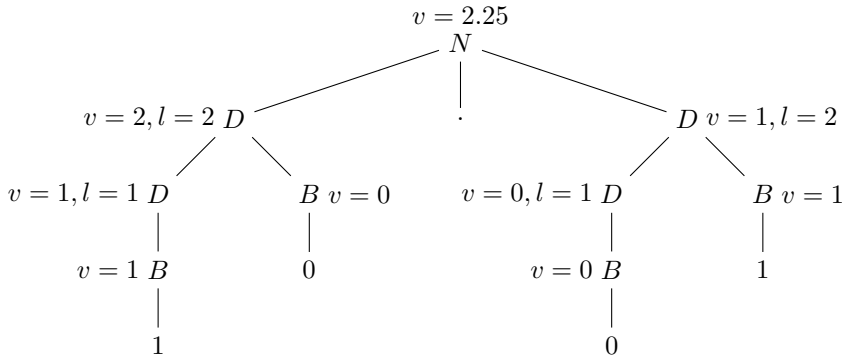


Figure 16.1: The decorated tree for string 10.01_2

- Σ is the set of terminals.
- P is the rule set.
- S is the axiom.

Suppose that the axiom is not referenced anywhere in the right parts of the rules and that the axiomatic production is unique.

2. Define a set of semantic attributes, associated with nonterminal and terminal symbols. The attributes associated with a nonterminal symbol D are denoted by $\alpha, \beta, \ldots$ and are grouped in the subset $attr(D) = \alpha, \beta, \ldots$. The set of all the attributes is divided into two disjoint subsets: left (or synthesised) attributes, e.g. σ, and right (or inherited) attributes, e.g. δ or η.

3. For every attribute (be it left or right) specify a domain, i.e. a finite or infinite set of possible attribute values. An attribute can be associated with one, two or more (non)terminal symbols. Suppose attribute $\alpha \in attr(D_i)$ is associated with symbol D_i, then write α_1 with a pedex i. If there is not any danger of confusion, freely write "α of D", "α_D" or in a similar way).

4. Define a set of semantic functions (or semantic rules). Each semantic function is associated with a support syntax rule p

$$p : D_0 \to D_1 D_2 ... D_r \quad r \geq 1$$

Two or more semantic functions may share the same syntactic support rule. The set of all the functions associated with a given support rule p, is denoted as $fun(p)$ (may be empty).

5. A generic semantic function, as follows

$$\alpha_k := f(attr(\{D0, D1, ..., Dr\} \setminus \{\alpha_k\}))$$

where $0 \leq k \leq r$, assigns attribute $\alpha_k(\alpha of D_k)$ a value by means of an expression f, the operands of which are the attributes associated with the same support syntax rule p (not another one), excluding the expression value itself (α_k). Semantic functions must be total.

6. Semantic functions are denoted by means of a suited semantic metalanguage like

- A common use programming language (C or Pascal).
- Pseudocode.

- A standardized software specification language (e.g. XML).

7. Models of semantic functions for computing left and right attributes of rule p, respectively:
   - $\sigma_0 := f(\dots)$ defines a left attribute, associated with the parent node D_0.
   - $\delta_i := f(...)$ so that $1 \leq i \leq r$ defines a right attribute, associated with a child node D_i.
8. Attribute associated with a terminal symbol
   - Is always of the right (inherited) type.
   - Is often directly assigned a constant value during lexical analysis (before semantic analysis), a semantic function is seldom used.
   - and commonly is directly assigned the terminal symbol itself it is associated with.
9. The elements of the set $fun(p)$ of the semantic functions that share the same support rule p, must satisfy the following conditions:
   - For every left attribute σ_i it holds:
     - if $i = 0$ there exists one, and only one, defining semantic function so that
     $$\exists!(\sigma_0 := f(...)) \in fun(p)$$
     - if $1 \leq i \leq r$ there does not exist any defining semantic function:
     $$\nexists(\sigma_i := f(...)) \in fun(p)$$
   - for every right attribute δ_i it holds:
     - if $1 \leq i \geq r$ there exists one, and only one, defining semantic function so that
     $$\exists!(\delta_i := f(...)) \in fun(p)$$
     - if i = 0 there does not exist any defining semantic function so that
     $$\nexists(\delta_0 := f(...)) \in fun(p)$$
10. Relationships of attributes and support rules:
    - The left and right attributes σ_0 and $\delta_i (i \neq 0)$, associated with (non)terminal symbols occurring inside of the syntax support syntax rule p, are said to be internal to p.
    - The right and left attributes δ_0 and σ_i $(i \neq 0)$, associated with (non)terminal symbols occurring inside of another syntax support syntax rule q different from rule p so that $q \neq p$, are said to be external to p.
11. It is permitted to initialise some attributes with constant values or with values computed initially by means of external functions. This is indeed the case for the attributes (always of the right type) that are associated with the terminal symbols of the grammar.

16.2 Execution order

When using attribute grammars we are exploring the syntax tree from the leaves to the root. In particular for each node we already should have computed the functions of the child nodes. In other words when we want to compute a function of a node, all the attributes of the child nodes used in that function should be already computed.

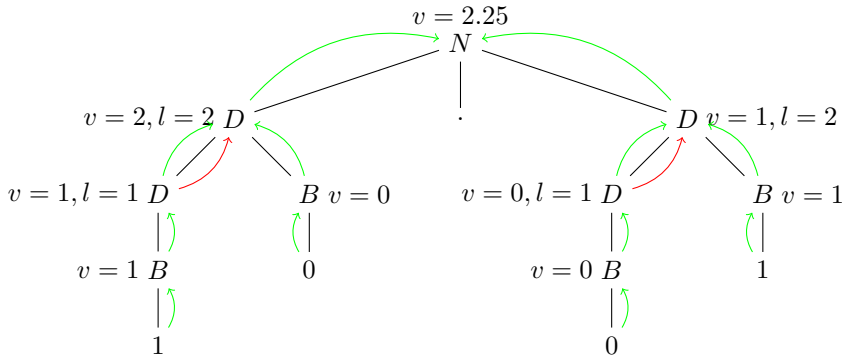


Figure 16.2: The dependency graph (as decorated tree) for string 10.01_2.

16.2.1 Dependency graph

To find an order in which all nodes should be visited can start by creating a dependency graph. The dependency graph can be obtained decorating the syntax tree. In particular we write an arrow from node i to node j if a function of node j needs an attribute of node i. Since a node can have multiple functions we can use colours or different types of arrows to differentiate the dependencies. For instance a green arrow from i to j means that an attribute of i is used in function 1 of j and a red arrow from k to j means that a parameter in k is used in j's function 2.

The dependency graph for our example is shown in Figure 16.2 (green arrows are used for function 1 and red arrows for function 2).

Visiting order Given the dependency graph we would like to know if there exists a visiting order of nodes so that when a node i is visited, all the nodes with attributes used in i have already been visited. If the dependency graph is acyclic then it's possible to find an unique visiting order for all the nodes.

16.2.2 Linearisation

Knowing that a visiting order exist, we would like to find one. In particular we would like to linearise the nodes so that all dependencies are respected. A dependency is respected if, when there is an arrow going from i to j, then j appears before i in the linear order. To linearise the tree so that all the dependencies are respected we can sort the nodes in topological order. The sorting algorithm works as follows

1. Consider a node with only incoming arrows.
2. Add the node to the end of the list of topological sorted nodes.
3. Remove the node from the graph.
4. Repeat 1 to 4 until no node that satisfies the condition at point 1 is satisfied.

Notice that the nodes should be visited in reverse topological order.

Part V

Laboratory on Compilers

Chapter 17

Introduction

A compiler is a software that translates the source code written in a programming language into an executable code (assembly).

A compiler is a pipeline made of different parts in which the input of a section is the output of the previous one. The input of the pipeline is the source code, the output is the executable code.

17.1 Front-end and back-end

Usually compilers can compile from multiple languages to assembly for different architectures, for instance `gcc` can compile `C`, `C++`, `Java` and `Objective-C` (but not only) to assembly for different architectures. For this reason a compiler is usually divided in two main sections

- The **front-end** that translates the source code in a middle representation that is independent from the assembly language. The `gcc` compiler uses a mid-level language called `GIMPE`.
- The **back-end** translates the middle representation in the specific assembly language.

17.1.1 Front-end process

The translation from a programming language to the mid-level language is divided in three parts

1. Initially the compiler divides the input in a series of tokens. The tokens can be keywords (like `for` in `C`), names or values.
2. The compiler than tries to fit the tokens to a grammatical rule.
3. Finally the compiler extracts the semantics.

Chapter 18

Lexical analyser

A lexical analyser is a tool that receives as input a string and extracts all the tokens in such string. In a programming language a token is for example an identifier or a constants.

Lexical analysis is performed by a scanner. Writing a scanner is a tedious task and error-prone so it is better to use a scanner generator to automatically generate a scanner that satisfies a set of rules.

18.1 Flex

Flex is an open source scanner generator. The process to create a program

1. The scanner is created starting from a description file (.ll extension) that contains the rules to recognise the tokens.
2. The description file is passed to `flex` that generates the C source code of the scanner.
3. The C source code of the scanner can be compiled (with `gcc` for instance) to generate the executable scanner.
4. The scanner executable takes a string as input and parses it to find some tokens.

18.1.1 Flex input file

The description file is divided in three sections

- **Definitions**.
- **Rules**.
- **User code**.

Each section is separated by a double percentage symbol %%.

Definitions

The definitions section is used to define regular expressions. Every regex is associated with a human readable identifier that is used in the other sections. The identifier works just like a C macro, every time flex finds the identifier the corresponding regular expression (defined in the definitions section) is replaced.

```
1 LOWERCASE [a-z]
2 UPPERCASE [A-Z]
3 LETTER    [a-zA-Z]
4 DIGIT     [0-9]
```

Listing 18.1: The definitions section of a description file.

Rules

This section contains the rules to bind a regular expression to an action. Every rule is made of

- A regular expression. When writing the regular expression every identifier has to be written in curly brackets. For instance if we want to use the `LETTER` identifier we have to write `{LETTER}`. In a regular expression the symbol `"` can be used to escape meta-characters (e.g. `"if"` means that we are looking for the exact string `if`).
- An action, i.e. a piece of code in curly brackets.

Every time the scanner find a regex defined in this section it executes the related action.

When we define the action to execute after a match we can use

- The `char* yytext` variable that holds a reference to the first character of the input string that matches the rule.
- The `int yyleng` variable that contains the length of the matches string.

```
1 {LETTER}({LETTER}|{LETTER})*    { return ID; }
2 {DIGIT} { printf("%c", yytext[0]); }
```

Listing 18.2: An example of rules in the rules section.

If more rules match one string then the scanner applies the following rules

1. **Longest matching rule**. The rule that generates the longest match is selected.
2. **First rule**. If the string have the same length, the rule listed first in this section is selected.
3. **Default action**. If no rules are found the next character in input is considered matched implicitly and copied to the output stream as is.

Multiple scanners To support multiple scanners rules can be marked with a prefix written between diamond brackets before the rule. The prefix is called **start condition** and is used to identify a rule as active. When a rule is active the scanner only tries to match rules with such start condition. For instance in the code 18.3 if the active start condition is `NUMBER` the scanner tries to match only the third rule, otherwise it tries to match the first or second rule.

```
<CHAR>{UPPERCASE} { ECHO; }
<CHAR>{LOWERCASE} { ECHO; }
<NUMBER>{DIGIT} { ECHO; }
```

Listing 18.3: An example of usage of start conditions

The initial start condition can be referenced with the `INITIAL` prefix while the `*` prefix identifies a starting condition that matches every start condition. The current start condition is stored in the `YY_START` variable.

Start conditions are handled by the source code as integers.

Actions Flex defines some special actions that can be used when defining a rule

- The `BEGIN(SC)` action places the scanner on the start condition SP.
- The `ECHO` action copies *yytext* to output.

User code

This section contains arbitrary code that flex can add in the generated code directly as we write it. This section also contains the main function that the scanner executable has to run. The main function should always execute the `int`yylex()— function (i.e. the function that flex generates and that parses the input string).

```
int main(int argc, char** argv) {
    return yylex();
}
```

Listing 18.4: An example of main function in the code section.

Code can be also written in other parts of the document. If code is inserted outside the user code section it has to be included between curly brackets preceded by a percentage symbol

```
%{
    #include <stdio.h>

    #define MAX 10
%}
```

Listing 18.5: The syntax of a piece of code outside the code section.

We usually put a piece of code outside the user code section to place such code in a specific position in the C code. For instance the include statements are usually written at the beginning of the document so that they are placed at the beginning of the source C code.

Chapter 19

Syntactical analyser

The study of the rules whereby words or other elements of sentence structure are combined to form grammatical sentences.

A syntactical analyser has to

- Identify grammar structures.
- Verify syntactic correctness.
- Build a possibly non ambiguous derivation tree.

When a sequence of tokens is recognised as the right hand side of a grammar's rule, it is replaced with the left hand side of the rule.

The semantic analysis aims to verify the correctness at semantic level: it verifies if a syntactically correct sentence is meaningful. This verification can be described as a decoration of the AST (Abstract Syntax Tree) performing

- Type checking (check coherency of types in expressions) .
- Symbol definition before use.
- Generated code.
- Value of expressions.

Sometimes a sequence of tokens might be syntactically correct but it might be semantically correct. For instance in the code

```
int a;
a = 0.5;
```

the second instruction is syntactically correct but semantically wrong (we should cast the float).

Sometimes the syntactical analyser can replace some expressions with constants if possible. For instance the instruction

```
a = 1 + 1;
```

isn't treated as a sum and an assignment but only as an assignment because the compiler can replace the sum $1 + 1$ with the constant 2.

19.1 Bison

GNU Bison is the standard tool of the GNU software suite to generate a parser.

It's meant to be coupled with Flex, in fact Flex generates code in a syntax that Bison expect.

We feed Bison with a description of the syntax analysis we want to perform and builds a C program that can perform such analysis.

19.1.1 Workflow

1. Take description of parser.
2. Generate C source of parser.
3. Compile the executable. The executable can parse the input stream to generate the output.

3 sections divided by % symbol. The sections represents a part of the C source

- Prologue. Whatever we want at the top of the C source code.
- Definitions. Define tokens, precedence of operators to disambiguate ambiguous grammars, types of non terminals. Bison creates the token that are used by Flex. IN particular it creates an header file that can be imported by Flex.
- Grammar rules.
- User code. This section contains the main function.

```
%{
Prologue
%}
Definitions
%%
Rules
%%
User code
```

Token A token is defined using the keyword

```
%token
```

followed by the name of the token.

```
%token IF WHILE DO FOR GOTO
```

Precedence and associativity

Precedence is fundamental to disambiguate ambiguous grammars. Precedence alone can't solve the problem, in fact we also need to specify the associativity of an operation.

Associativity Associativity can be specified using the following tokens

- `%precedence`. We are not specifying one.
- `%left`. The operator is left associative, we fold from left to right.
- `%right`. The operator is right associative, we fold from right to left.
- `%nonassoc`. The operator is not associative and when an associative syntax is found, Bison stops.

Precedence Precedence is defined by the order in which the associativity tokens are declared, this means that a token defined later has the precedence over a token defined earlier.

19.1.2 Union

An union is a composite type (like a structure) in which we can assign only one of the elements a time. The token `%union` allow us to specify the entire collection of possible data types for semantic values. Each type defined is named `YYSTYPE`. Semantic value fields can be associated to terminal (using the `%token` declaration) and non-terminal symbols alike (using the `%type` declaration).

19.1.3 Grammar rules

Grammar rules are specified in BNF notation. If not specified, the left hand side of the first rule is the axiom.

```
LHS : NONTERM term | NONTERM | 'specific_char' | /* epsilon */;
```

Semantic analysis We can put C code between curly braces at the end or in the middle of a rule. Such code is executed whenever the rule is used. Such code is called semantic action.

The semantic action can access all the tokens in the stack of the automaton that is doing syntax analysis. The variable `$1` represents the token on top of the stack, `$2` the element below.

Definitions