FLC defucked version

Luca Alessandrini, Elia Ravella, Juri Sacchetta

November 11, 2022

Contents

Chapter 1

Regular Expressions and Regular Languages

1.1 Introduction

Regual languages are the simplest family of formal language.

1.2 RL and RE

1.2.1 Introduction and Definitions

A Regular Language is a **language over an alphabet that can be expressed by applying** *a finite number of times* this three operations:

1. Union $\bigcup, \vee$
2. Concatenation $\cdot$
3. Star (Kleene Star)

Rule of precedence: star, concatenation, union. It is permitted to use $\varepsilon = \emptyset^*$

Definition 1 (Regular Language) *A language is regular if it is the meaning (hence if it is generated) by a regular expression.*

Definition 2 (Family of Regular Languages (REG)) *Collection of regular languages.*

Definition 3 (Family of Finite Languages (FIN)) *Collection of all languages of finite cardinality.*

Caution: REG and FIN are sets of sets of set of strings, not set of strings themeselves.

Notice that *every finite language is regular* because it is the (finite) union of finitely many strings, and each string is in turn the concatenation of finitely many letters or in general of alphabetic symbols

$$\left(x_1 \bigcup x_2 \bigcup \ldots \bigcup x_k\right) = \left(a_{11}a_{12}\ldots a_{1n} \bigcup \ldots \bigcup a_{k1}a_{k2}\ldots a_{kn}\right)$$

Remember that $FIN \subset REG$

1.2.2 Subexpression of a regexp

1. Consider a fully parenthesized regex

$$e = \left(a \bigcup (bb)\right)^* \left(c^+ \bigcup \left(a \bigcup (bb)\right)\right) \tag{1.1}$$

2. Number every alphabetic symbol that occurs in the regexp

$$e_N = \left(a_1 \bigcup (b_2 b_3)\right)^* \left(c_4^+ \bigcup \left(a_5 \bigcup (b_6 b_7)\right)\right) \tag{1.2}$$

3. isolate the subexpressions and put them into evidence

1.2.3 Other Operators

Also set operators can be used: Intersection and Complement are used when it's more concise to express a language to a set of languages (the former) or the inverse of a complex rule (the latter: a clear example is "a language with no 2 consecutive 'a' in it").

To define a Regular Language formally a *Regular Expression* is used, defined as a string of either elements of the alphabet or operators that symbolize the three operation for regular languages. The Derivation Relation binds the RE with its productions: in fact, an expression is said to be *derived* from another one if it's totally or partly a choice of another one. A choice is an alternative between two options that the RE gives you (a union operator poses a choice). A Language is defined as the set of all possible productions that can be derived from a RE.

1.3 Ambiguity

Easy enough: a RE is ambiguous if more than one identical productions exists. This obviously leads to problems in the parsing procedure, due to the difficulty to "trace back" the choices that has been made to get to that point.

1.4 Closure Property

Pretty straightforward, is the same "closure" concept from classic algebra; a family of languages is closed under an operator if applying n times that operator to a language of that family obtains only languages of that family. You can't "exit that family" with that operator.
The REG(ular) family of languages is closed under

- concatenation

- union
- star

Therefore is closed also under derived operators from these (cross).

Chapter 2

Context Free Grammars

Regular expressions as a tool to generate languages are not enough powerful. Super simple languages like "all words followed by their reverse" or "concatenation of two strings of the same length" cannot be generated.
Context Free grammars are defined by means of four entities

- V: *non-terminal alphabet*, is a set of *non-terminal symbols (or syntactic classes)*
- $\sum$: *terminal alphabet*, is the set of the *characteres* that costitute the phrases
- P: is a set of *syntactic rules* (also called *production rules* or simply *rules*)
- $S \in V$ is a particular non-terminal, called axiom

One particular nonterminal is choosen as starter (or axiom). A Rule is an ordered pair with a left side and a right side: left side can contain only nonterminals, right side both terminals and nonterminals.

In order to define a language, one can use RULES that, after repeated application, allow to generate all and only the phrases of the language The set of such rules constitutes a GENERATIVE GRAMMAR (or SYNTAX).

Definition 4 (Regular Language) *A set of rules generating all the possible strings belong to the language.*

2.1 Rule Classification

Lowercase or greek letters: terminals.
Uppercase letters: terminals.

- Recursive: A $\rightarrow \alpha$ A β
- Left Recursive: A $\rightarrow$ A β
- Right Recursive: A $\rightarrow \alpha$ A
- Copy: A $\rightarrow$ B

- Linear: at most one nonterminal on the right side
- Right Linear: linear + nonterminal is suffix
- Left Linear: linear + nonterminal is prefix
- Chomsky normal: right side is either two nonterminals or a terminal
- Greibach normal: right side starts with a terminal symbol
- Operator normal: A $\rightarrow$ B c D

2.2 Grammar Cleaning

Writing a grammar following non-grammar rules (so doing it for a purpose and not for its own sake) can lead to useless rules or unreachable terminals/nonterminals. This is prevented by the Grammar Cleaning algorithm:

1. Compute the well-defined nonterminals (the ones that are "reachable from the production")
2. From the well defined nonterminals build the production tree (such a rare technique in computer science) and check if there are nonterminals in the nonterminal alphabet that are not reachable from the axiom.
3. (Optional but useful) Check for circular derivation like A $\rightarrow$ A $\rightarrow$ x

2.3 Infinite Grammars

A grammar is said to be infinite
$\Leftrightarrow$
A grammar has a recursive production

The grammar is assumed to be clean and free from circular derivation.

2.4 Syntax Trees

The production procedure can be visualized by a tree (how strange) where every arch represents a symbol of the right part (or a choice for it). They're regular trees (so connected directed acyclic graphs) but the nodes on the same level (or at least sons of the same father) are ordered from left to right.
The degree or arity of a node is the number of its *siblings* (and not of his sons, that's more intuitive). The *frontier* of the tree is the sequence of all leaves. A syntax tree has the axiom as the root and a sentence as the frontier.
Remember: a syntax tree represents **a single production** and not all possible ones.

2.5 Ambiguity

Ambiguity in grammars is similar to the ambiguity for regular expressions. The definition is pretty much the same: a sentence is ambiguous if it admits several different syntax trees.
Syntactic ambiguity (the only of interest when studying formal languages) is the kind of ambiguity that derives from the *structure* of the production, rather than the effective meaning it carry.

The formal problem of discovering/computing the ambiguity of a grammar is an undecidable one. This is in practice not a great problem: heuristics to check if a given grammar is ambiguous exists, but most importantly the grammar designer should keep in mind (and design accordingly) the said grammar in an unambiguous way.

2.5.1 Ambiguous Forms and Remedies

Bilateral Recursion

A bilateral recursion (so a both left and right recursive rule) often leads to ambiguity because there's usually no fixed order of generation of nonterminals.

This can be usually avoided by splitting the two side recursive rule into more rules each one either right or left recursive.

Ambiguity from Union

Pretty straightforward: if two different languages can generate the same sentence, the union of these languages is ambiguous by construction.

To cope with such problem, often a full redesign of the grammars is needed.

Ambiguity from Concatenation

Concatenating two (or more) languages can be a problem if a suffix of a language is a prefix in a language that follows it.

To remove this kind of ambiguity, it's sufficent to add an extra separator character between languages.

Ambiguity in Conditional Phrase

The infamous "dangling else problem" is an ambiguity problem that falls in this category. Generally this kind of ambiguity arises when more variants are accepted for the same language structure, and no particular delimiter for parts is provided.

Usually removed by adding delimiters (endif, fi)

Inherent Ambiguity

An inherent ambiguous language is generated by a set of grammar in which *all* the grammars are ambiguous

2.6 Closure Property

The CF languages (so all languages generated by Context Free grammars) are closed under the operations of

- union
- concatenation
- star

Complement and Intersection

Let L and L2 be regular languages. Their intersection and the complement of each is still a regular language. (The formal proof utilizes the recognizers automata to prove this).

2.7 Equivalence

Two grammar are said to be equivalent if they define the same language. Two grammars are *strongly* equivalent if they define the same language **and** they have the same *condensed skeleton trees*; this means that every production from both grammar is generated "with the same choices".
Strong equivalence $\Rightarrow$ weak equivalence
Strong equivalence can be formally proved, while checking for weak equivalence is an undecidable problem.

Structural Adequacy

The difference between strong and weak equivalence is not only a formal one: if we consider (for example) the generation of arithmetic expression (that should take into account also the operator precedences) only the grammar with the syntax tree that respects the precedences, and all the *strongly equivalent* ones, will be structurally adequate.

2.8 Grammar Transformation and Normal Forms

Normal forms are standardized rules. They don't reduce grammar expressiveness of the grammar, and using them does not affect the language iself. They

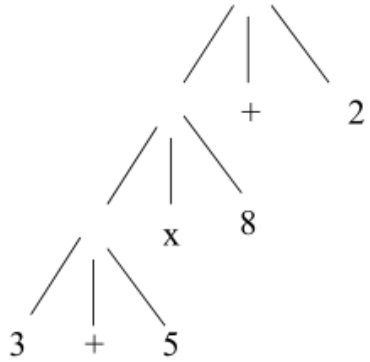


Figure 2.1: Structurally inadequate syntax tree: it can be seen how the operator precedence is not respected

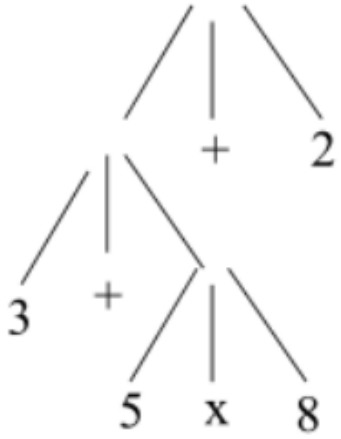


Figure 2.2: Structurally adequate tree

just guide the design process to obtain a grammar with certain properties.

Elimination of the axiom from right part

Straightforward. Use an additional rule from the axiom to a sub-axiom that is replaced in all other rules.

Nullable nonterminals and empty rules

A nonterminal is nullable if exists a derivation from said nonterminal to the empty string. To eliminate them, eliminate all the empty productions from the right part of the rules, substituting them with the non-empty sub productions.

Copy rules

Copy rules are rules in the A $\rightarrow$ B form, with both A and B nonterminals. An algorithm to eliminate copy rules functions this way:

1. A set *Copy(A)* is defined, that contains all the transitive copies of A and A itself

2. *Copy(A)* is filled with all the nonterminals that are copies (direct or indirect) of A, so that a straight derivation A $\rightarrow$ C exists

3. The ne set of rules is defined, eliminating all the productions from A to the nonterminals that are in *Copy(A)*.

The above algorithm should take into consideration that a production A $\rightarrow$ BC can be a copy production if either B or C are nullable. Often it's assumed a nonnullable grammar.

Recursion conversion

Another important class of grammars is the nonleft-recursive. Right recursive grammars are preferred because their laguage is simpler to parse. To transform a left recursive rule to a right ecursive an algorithm has been designed:

1. Identify all the recursive rules of form A $\rightarrow$ A γ
2. Add an auxiliary nonterminal X and add the set of rules X $\rightarrow \gamma$ X; the expansion of A becomes A $\rightarrow \alpha$ X, where α is a possible production obtained from A.

Now all the immediate l-recursive rules are taken care of.

Chomsky normal form

Only two possible types of rules: homogeneous binary (nonterminal to two nonterminals) or terminal with singleton right part (nonterminal to single terminal). If the empty string is in the language, it can only appear as an axiomatic production, and the axiom cannot appear in *any* right part.
Obtaining a Chomsky grammar:

1. all non binary rules are converted to binary: the first non terminal is singled out
2. and all the others are replaced by another additional nonterminal, which is added to the nonterminals alphabet and has as production only the one that generates all the leftover nonterminals from passage 1
3. passages 1 and 2 ainherently recursive
4. to cope with the rules that have both terminal and non terminals in the right part, the terminals are simple replaced with singleton nonterminals

Greibach and real-time forms

Real time grammars are grammars where each right part begins with a terminal symbol. If the additional costraint "the right part could contain exactly one terminal followed by zero or more nonterminals" we obtain a Greibach normal grammar.

2.9 Extended Context Free Grammars

They're simply context free grammars (also called Backus Normal Form -> BNF) extended with the regular expression's operation such as star, and union operation. The closure of the CF family under this two operations renders this exension as a merely aesthetic improvement, because the expressiveness remain unchanged.
The language of a ECF grammar is generated as the one of a CF grammar; the only effect the regular expansion has on the syntactic trees is an augmentation of the breadth and a decrease in the height. This is a further improvement to readability.

2.10 Grammar of regular languages

As previously stated, regular languages (generated by regular expressions) are a proper subset of context free languages (generated by a context free grammar). This means that there's a one-to-one correlation between regular expression and a grammar that expresses the same language.

2.10.1 Uni/Linear Grammars

A grammar is said to be linear if every rule of that grammar is in the A $\rightarrow$ uBv form, where A and B non terminals (B can also be ϵ) and u and v terminal symbols.
If only one of the two terminals is present the grammar is said to be right- or left-linear, wheter the nonterminal is right or left side of the terminal symbol.
A grammar where all rules are either right or left linear is said unilinear. A strict unilinear grammar has only one terminal per side of the rule.

Relation with regular expressions

regular languages are a subset of unilinear languages: REG $\supseteq$ UNI; moreover, from an uniliear grammar we can always derive a regular expression with the same expressiveness $\Rightarrow$ UNI $\supseteq$ REG. We obtain (IMPORTANT) UNI $\equiv$ REG.

Chapter 3

Finite Automaton as Regular Language Recognizers

3.1 Preliminar definitions

You should know what a FSA is. In particular, you should know well DFSAs
A Finite State Machine (or finite state automaton) is composed of a set of states, a transiction function (that determines the change of state at a certain input) a set of termination states, and an input alphabet.
A DFSA is expressive as an uniliear grammar (so like a regular expression).

3.2 Clean automaton

A state can be useful or useless. A state is useful if it is reachable from an initiale state and a final state can be reached from it (condition of accessibility and post-accessibility). Otherwise it's useless.
For every finite automaton there exists an equivalent clean automaton.
To clean an automaton it can be easily flooded starting from the initial state then starting from the final state(s); all states not flooded at the end are useless.

Property 1 (Minimal automaton - uniqueness property) *For every finite state language, the finite recognizer is minimal w.r.t. the number of states exists and is unique (apart from a renaming of states)*

This property enables to represent a whole class of equivalent automata with a single one, that moreover is *minimal*.
An algorithm to reduce automaton to their minimal equivalent is based on the equivalence relation between states called *distinguishablity*.

Definition 5 (Undistinguishable states) *Two state are undistinguishable if applying the same input to the both reach a final state or both do not.*

This relation looks like it does not provide a tool to effectively distinguish two state, but it's a RECURSIVE DEFINITION:

- The sink state (or error state) is distinguishable from all other state;

- If a state is final is distinguishable from every other non final state;
- Two states are distinguishable **if their next state is distinguishable**.

3.3 Nondeterministic Automata

As a unilinear grammar rule can contain multiple alternatives for the same nonterminal, a FSA can have multiple states associated to a input symbol. Moreover, an automaton could also have spontaneous moves, so arcs that are not associated with any kind of input, besides the empty string.

If one or both these two kinds of situation happen to be in a FSA it's called nondeterministic, due to the nondeterminism of the output (same input can lead to different output).
There are a few motivations to add nondeterminism to an automaton:

- Concision: a nondeterministic automaton is often smaller than his deterministic counterpart, rendering the former more readable
- Mapping to grammars: as said, multiple alternative rules are more easily mapped on a NFSA than on a FSA
- Language reversing: when reversing an automaton (so changing arrows direction and initial/final states) nondeterminism often pops out
- ND recognizers: NFSA can be used to draft out a recognizer automaton

3.3.1 Elimination of nondeterminism

The final implementation of a recognizer automaton must be deterministic (for now).

Theorem 1 *Every nondeterministic automaton can be transformed in a deterministic equivalent one.*

The theorem 1 lets to the definition of:

Lemma 1 *Every unilinear grammar*[1] *admits a equivalent nonambiguous one.*

We focus on removing spontaneous moves to do we introduce:

Backward propagation method

1. Compute the transitive closure of ϵ moves: if three states are connected sequentially by spontaneous moves, add all the ϵ moves that shows all the ϵ paths.
2. Backward propagation of the scanning moves over the epsilon moves: if a path contains a normal scan move AND a spontaneous move, then connect the first and last states of that path with a scan move with the element scanned.

[1] See section: 2.10.1

3. Add all the states that spontaneously arrive to the final state to the final states set.
4. Delete all ϵ moves left.

Definition 6 (Ambiguity) *As for the grammars, an automaton is ambiguous if it accepts a string with two different computations.*

3.4 Automaton to regexp - Brzozowski McCluskey method

The Brzozowski McCluskey method (BMC) method allows to generate a regular expression from an automaton that produces the same language.
Assumptions:

- unique initial state
- initial state has no incoming arcs
- unique final state
- final state has no outgoing arcs

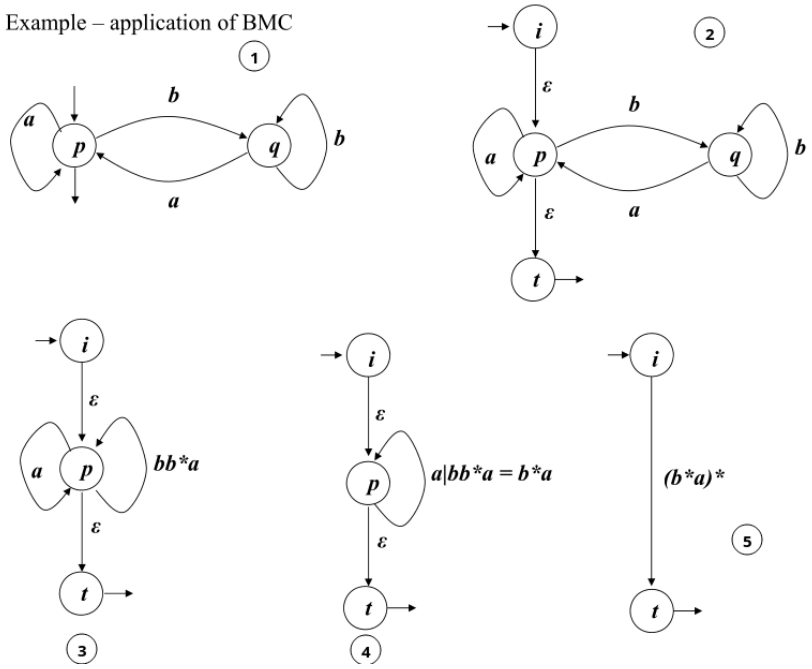
The algorithm is quite simple: at every step, an internal node is removed and substitued by a set of arcs labeled as regular expressions (the additional arcs must not modify the automaton behaviour, obiously. They should compensate with the regexp the introduce the lack of the removed state). The algorithm terminates when only initial state and final state are left, so one only arc with a single regexp, equivalent to the whole automaton.
Notes for nerdier people: this methods is similar (if not the same) to the one used to pass from an unilinear grammar to a regexp making use of "linear language equations".

3.5 Regexp to non deterministic recognizer - the Thompson Structural Method

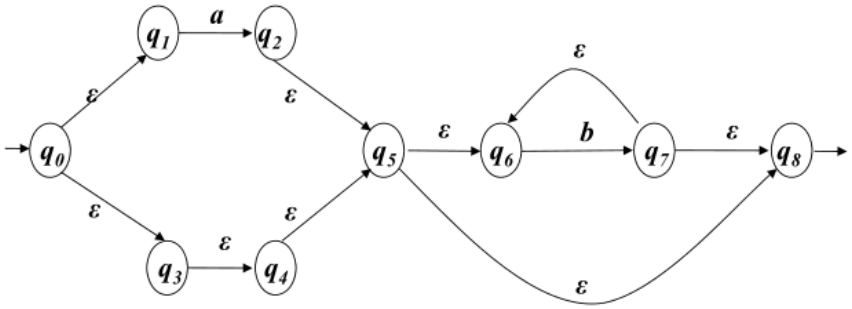
Thompson's method analyzes a regexp by simple parts and generates the components associated to them. It then interconnects the components to build the whole automaton. A parallel is drawn between the operators in a regexp and the finite state components they generate.

- An **atomic expression** is just translated as an arc between two states. The empty string is considered an atomic expression.
- **Concatenation** is expressed as a spontaneous move that connects two states, a single epsilon arc.
- On the other hand, **alternatives** are parallel blocks of states; all blocks connected to a single entry state and a single exit state through epsilon moves. The **star closure** instead is built as a recursive block of components, encapsulated in a initial and final state. The extreme steates are connected by a epsilon move.

Example:



$$(a \cup \varepsilon).b^*$$



3.6 Regexp to deterministic automaton - Berry Sethi Method

3.6.1 Theorical fundament: local languages and local automata

In order to fully understand the Berry Sethi method (ahaha) we must introduce the family of local languages. Local languages are a proper subset of regular languages (LOC $\subset$ REG) and are composed of

- a **set of initials** that is the subset of the alphabet that represents the terminals that start every sentence.
- a **set of finals** that is the subset of the alphabet that contains the terminals which end the sentences.
- a **set of digrams** that contains all the substrings of length 2 present in the sentences.

The complement of the digram set is also an important set to consider.
A local language is called so iff the non empty sentences of that language are exactly definable through combinations of the three Ini, Fin and Dig sets.
The double implications states that the local language must include *all and only* the local sentences, so generated by combining Ini, Dig e Fin.
This is a subtle definition: it's possible for a language to define a Dig set of digrams that (for example) can be generated only in a certain order, so that not all the possible combination of digrams are possible; that language is not local.
Another important property is that a word can be estabilished as member of a given local language by watching only two characters of it.

Local language recognizer

Building a recognizer for a local language is extremely simple: due to the "window of fixed length" to recognize a word, a 4 state automaton is enough.

1. Verify that the initial char is in Ini,
2. Verify that all the couples are in Dig,
3. accept if last char is in Fin.

This algorithm can be fully carried out by a finite automaton. This automaton should be built this way:

- a unique initial state q_0 that's also final if ϵ is in the language;
- a set of final states F = Fin;
- a transiction function:

$$\delta = \begin{cases} \forall a \in \text{Ini } \delta(q_0, a) = a \\ \forall xy \in \text{Dig } \delta(xy) = y \end{cases} \tag{3.1}$$

An automaton is said *local* if it satisfies the condition

$$\forall a \in \text{Alphabet},\ q \in \text{States}\ |\{\delta(a,q)\}| <= 1 \tag{3.2}$$

That translates in "two identically labeled arcs cannot enter distinc states".
A *normal local automaton* satisfies also the two following properties:

- The state set is exactly the values in the alphabet plus the initial state;
- All and only the arcs labeled a enters state a. This implies no arcs enter the inital state.

Equivalence between local automaton and local language

For any language L the three conditions

- L is local
- L is recognized by a local automaton
- L is recognized by a normal local automaton

3.6.2 The Berry Sethi Algorithm

The Berry Sethi algorithm exploits the properties of local languages to obtain a deterministic recognizer for any regular expression.
To do so, we must introduce a new set of terminals (to the Ini, Dig and Fin) that is the Fol(lower). It is defined as the set of characters that can appear right next to one, given. It carries the same information of the Dig set, and it's defined from that:

$$\text{Fol}(a) = \{b | ab \in \text{Dig}\} \tag{3.3}$$

If no followers exists for a certain symbol, the $\dashv$ symbol signals it. So, id *reg* is our regular expression, then "*reg*′ $\dashv$" is the numbered version of *reg* followed by the terminator. "Numbered" just means that every character in the regexp has a unique numeric identifier.
The algorithm is composed of three main parts:

1. Initial state definition: the initial state is marked as the Ini set of the regexp analyzed (Ini(*reg*′ $\dashv$)).
2. Every distinct element in the Ini state will label an outgoing arc from the initial state. REMEMBER: even though the regexp is numbered, arcs and states uses the nonnumbered version of the characters.
3. Every state is named (labeled) as the set of followers of the read character (the one on the followed arc).
4. The steps 2 and 3 are repeated until no more states can be generated. The final states are the ones that have in the followers set $\dashv$.

Berry Sethi algorithm as automata determinizing process

The BS algorithm can be also used to convert a nondeterministic machine into a deterministic one. It's very easy to do so: it is enough to number all the characters on the non-spontaneous moves of the starting automaton, compute the obtained Ini, Fol and Fin sets from the language obtained, then applying BS algorithm to the language obtained. The result is a deterministica automaton that recognizes the same language as the starter one.

3.7 Recognizers for Complement and Intersection

As regular languages can be complemented and intersected, so can their recognizer automata be transformed to match such operations.
Algorithm to complement a DFSA regular recognizer:

1. Add the sink state
2. Connect all the states to the sink state, wheter a state cannot read a character (for example, on the alphabet a, b, if state q cannot cannot read "a" an arc q $\rightarrow$ sink state is added, labeled "a")
3. Reverse the final with the initial state

Due to the De Morgan identity L $\cap$ R = !(!L $\cup$!R), where "!" is the complement, it's sufficient to know an algorithm for complementing the recognizer to also build one for the interception.

Chapter 4

Pushdown automata and parsing

Pushdown automaton: a finite state machine enriched with a LIFO queue (or *stack*) as memorization device, that supports the classical push-pop operations. It's useful to portay this device with a terminator on the bottom, so that it signals the emptiness of the stack. Even a single-state nondeterministic pushdown automaton can (with only the stack) be used as a recognizer; this is called *daisy automaton*.
To define a pushdown automaton we just expands the FSA deefinition with the stack alphabet and a special character from this alphabet, the stack bottom / terminator / initial stack symbol. REMEMBER: the transiction function doeas not depends solely on the given input anymore, but also on the stack's top symbol; the domain is changed, wrt the classic FSA.
Domain and range of the transiction function for a FSA: (set of states) $\times$ (input alphabet) $\rightarrow$ (set of states)
Domain and range of the transiction function for a PDA: (set of states) $\times$ (input alphabet) $\times$ (stack alphabet) $\rightarrow$ (set of states) $\times$ (stack alphabet)
In the graph for a pushdown automaton, also the stack operation must be specified. **A pushdown automaton is as expressive as a context free grammar**

4.1 Deterministic stuff

Only deterministic context free languages (so, accepted by deterministic PDA) are considered in the field of compilers, due to efficiency and complexity reasons. Non determinism can be found in PDA in 2 forms mainly:

- the transiction function is multi valued (it has more than one possible output for certain input)
- spontaneous moves *interfere* with the recognizing procedure, in particular
  - a spontaneous move and a reading move are defined and they overlap
  - a spontaneous move is multi-valued

4.1.1 Deterministic languages family

The DET family (the context free languages that can be recognized by a deterministic PDA) is a proper subset of the CF languages family.
This family is closed under these operators:

Operation	Property	(Property already known)
Reflection	$D^R \notin DET$	$D^R \in CF$
Star	$D^* \notin DET$	$D^* \in CF$
Complement	$\neg D \in DET$	$\neg L \notin CF$
Union	$D_1 \bigcup D_2 \notin DET, D \bigcup R \in DET$	$D_1 \bigcup D_2 \in CF$
Concatenation	$D_1.D_2 \notin DET, D.R \in DET$	$D_1.D_2 \in CF$
Intersection	$D \bigcap R \in DET$	$D_1 \bigcap D_2 \notin CF$

Figure 4.1: Closures properties of deterministic languages

4.2 Parsing

Parsing (syntax analysis) is the procedure to estabilish if a string belongs to a certain language. A parser is an automaton that puts in place the analysis. It compute the syntax trees or gives an error if it finds one.

Types of parsing

Parsers can be classified in top-down parsers and bottom-up parsers (another distinction that's totally new to CS students, uh?) to distinguish between the approach to the syntax trees.

- *top down* analyzers grow the tree from the root downwards towards the leaves; this means that every step of the algorithm corresponds to a *derivation in the grammar* so a *left* $\rightarrow$ *right* passage in a rule
- *bottom up* parsers starts from the leaves of the syntax tree and reach the root; this means that every step of the parsing procedure corresposnds to a *reduction in the grammar*, so a *right* $\rightarrow$ *left* passage in a rule

4.2.1 Grammars as networks of finite automata

To build a recognizer for a grammar, we can see it as a set of indipendent productions for each nonterminal. Every nonterminal is associated with a dedicated machine (a FSA, for example) that when has to read another nonterminal just "invokes" another machine to do so.
This means that, at the end of the transformation, we have a machine for every nonterminal, that can interpret all the productions of said nonterminal, and

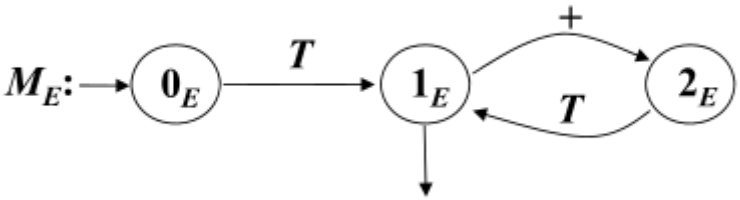
on the arcs has all and only the right-side elements of the rule taken into account.

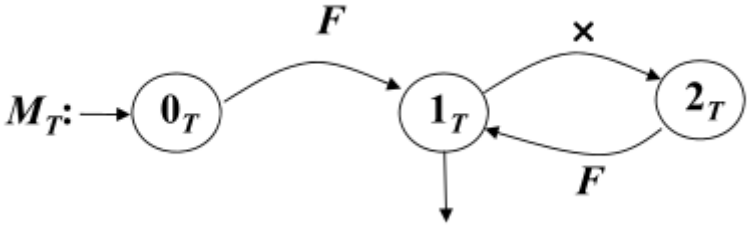
Good example

We consider a grammar that generates arithmetic expressions. We isolate the

$$E \rightarrow T(+T)^*$$
$$T \rightarrow F(\times F)^*$$
$$F \rightarrow a \mid \text{`('} E \text{`)'}$$

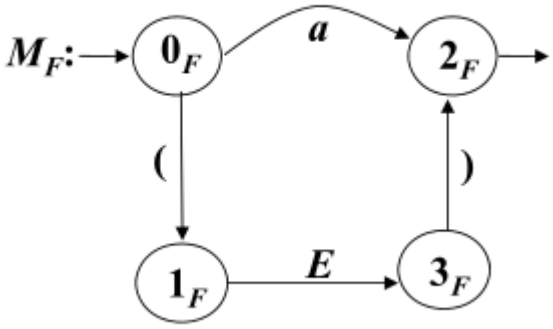
three nonterminals E, T and F and build a machine for each one of them





4.2.2 Lookahead sets

All parsers are implemented as DPDA. They rely on a second component, the lexical analyzer, that reads the input stream and recognizes the patterns in it in order to separate all the lexeme (or tokens) to give to the parser itself.
To ensure the automaton is actually deterministic, parsers *preprocess* the grammar's rules, and select the sets of tokens that can be read after each move; these are the *followers* or *lookahead* set. These sets of token are useful when another



machine in the network must be called: it's necessary to know which character we *will* read and compare it with the first character read in the machines. To be noted: each *nonterminal, for each configuration,* has a set of lookahead characters.

Lookahead and closures

Each pair {state, token} is said to be a candidate if the token is a legal lookahed character for the current activation of the machine in state "state". That does not mean so much, at the end of the day.
To compute each set of candidates we use a *closure function.* The closure function is defined this way:
Let C be a set of candidates:

$$\text{closure}(C) = \begin{cases} C \\ \langle 0_b, b\rangle \in \text{closure}(C) \text{ if } \begin{cases} \exists \text{ candidate } \langle q, a\rangle \in C \textbf{ and} \\ \exists \text{ arc in the machine net so that } q \xrightarrow{B} r \textbf{ and} \\ b \in \text{Ini}(\text{L(r)} \cdot a) \end{cases} \end{cases} \tag{4.1}$$

EXPLANATION OF THE FORMULA: the formula returns a *set of candidates*, so a set of pairs ⟨ state, character ⟩. Its input is *another set of candidates*, so the output will be the closure not of a unique state, but of a set of them. The formula states that a candidate can be in the closure if:

- The candidate *is* the closure;
- The candidate is composed of an initial state (in the formula written as 0_b) of a machine in the network and a character ("b") which is in the set of initials of a language that is *reached through machine B.*

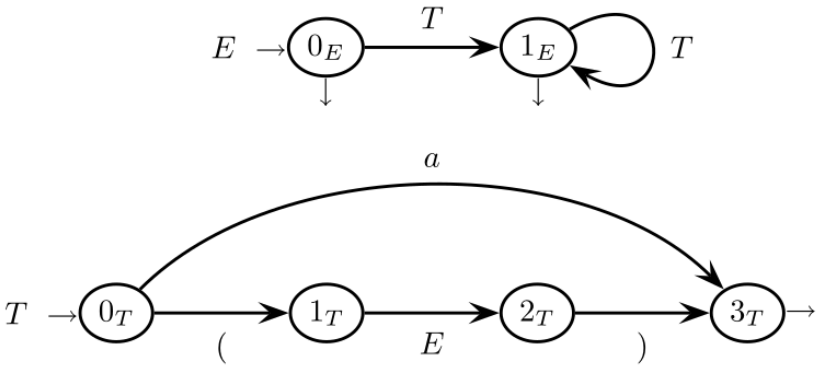
Notice how subtle the recursivity is: in order to compute a language of a state (L(r) in this case) we should build the closure of the candidates starting from state r. The fixed point is reached when state r is final, so L(r) is ϵ thus the following character ("a" in the formula) is added to the candidate.

Meaningful example

We consider grammar G

$$G = \begin{cases} E \to T^* \\ T \to \text{'('}\ E\ \text{')'} \mid \text{a} \end{cases} \tag{4.2}$$

that generates expressions and terms. We consider the machine network



We compute the closure for candidate $\langle 0_e, \dashv \rangle$

- $\langle 0_e, \dashv \rangle$
- $\langle 0_t,\ (\ \rangle$
- $\langle 0_t,\ \text{a}\ \rangle$
- $\langle 0_t, \dashv \rangle$

4.2.3 ELR parsers

ELR = Extended Left to right Rightmost
When reducing a string in input, a bottom up parser has two possible moves available: *shift* to the next character (that simply means that a new state is pushed on the stack and no pattern has been recognized yet) or *reduce*: this means that a pattern (a right part of a rule) has been fully recognized and a state can be popped from the stack. These two operations are an adapted version of the pop-push model for the memory. They also represent the moving forward and backward in the rules of the grammar.
ELR is the name for the parsers that recognize a EBNF grammar.
To build the recognizer starting from the grammar a three-phase method has been outlined:

1. (Prerequisites: we have a network of machines that describes the grammar's rules) We build a special automaton called *pilot automaton* that is a generalization of the machines network. It covers all the possible combination of states and input.

2. The pilot is inspected to check if it's suitable for deterministic parsing, and three types of failure are searched for (we'll discuss them in a moment).
3. If the test at point 2 is passed, then the DPDA is built from the pilot.

Construction of the pilot automaton

The pilot automaton is (as already said) a method to represent *all* the possible computational path of another one. Every state (called macro state or m-state) is labeled as all the states it represents, and all the arcs keep in consideration all the possible reading for all the possible states encapsulated in a single m-state. Every state is divided in 4 parts:

- the two (canonically) upper parts are the *base* candidates: the states and lookahead characters of **non initial** state;
- the two (canonically) lower parts are reserved to the *kernel* candidates: the states and associated lookahead of the **initial** state.

Every final state in every m-state is usually highlighted (to help spot reductions). An algorithm that creates a pilot automaton can be described in this way:

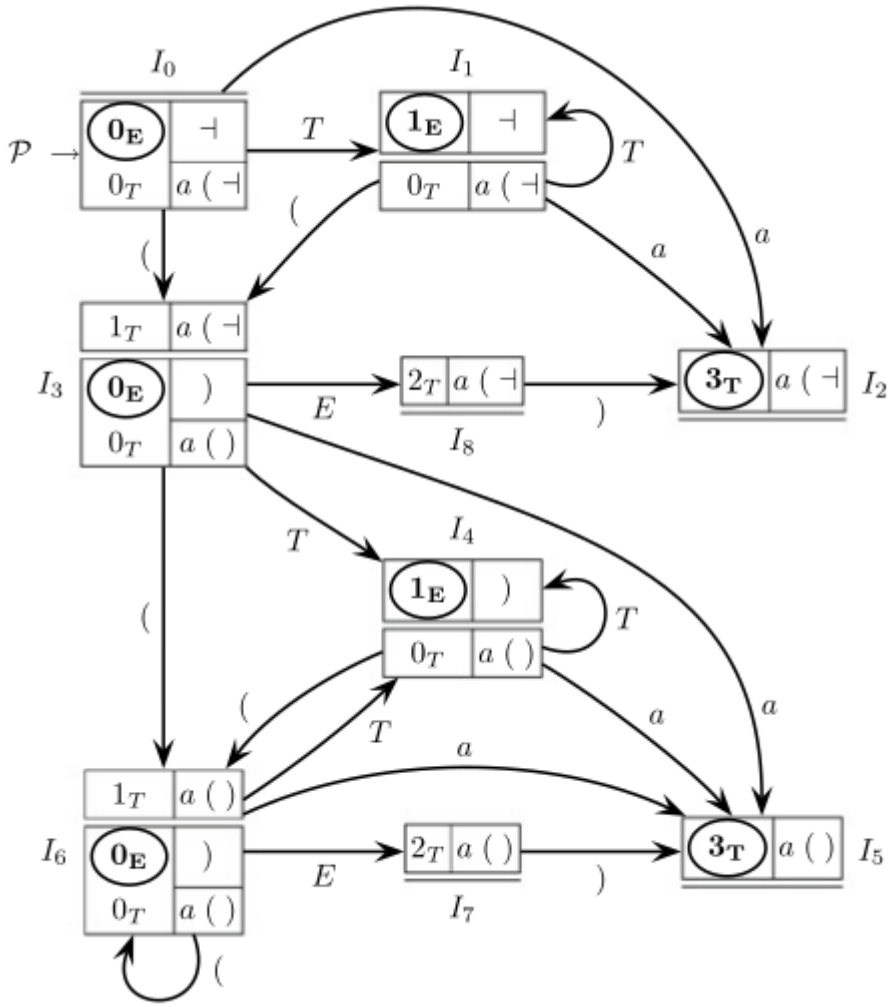
1. The initial state groups the first state of the first accessed machine AND all the states of the machine that can be reached with a ϵ-move. (So? They're all the first states of the nonterminals that *label* the outgoing arcs from the initial state. Being all initial states *for construction*, first m-state does not have the base part)
2. All the other states are built in a similar way: following an arc, I add all the state that I can found to the next m-state; I select the initial ones and put them in the closure, then I check for states that are reachable through a ϵ-move, and also add them to the closure. For the closure part, I shall recalculate the lookaheads, things that *generally* I do not do with the pure base part.

Is a pilot automaton still a PDA? Yes: the extra labeling of the states and the interpretation of the strings can be seen as pushing and poppoing the entire candidates onto the stack and popped back when needed.

Meaningful example

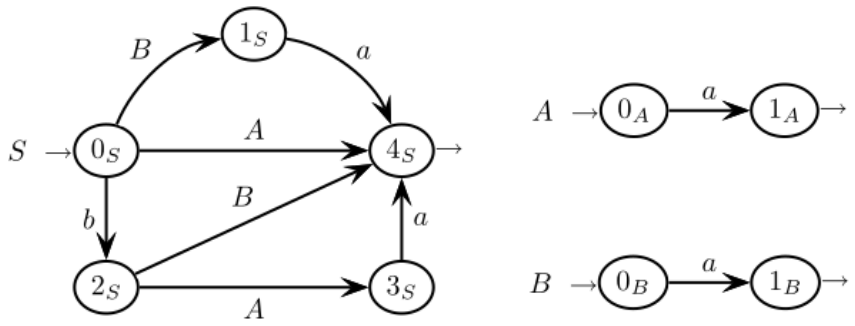
We build the pilot automaton for the grammar in (5); at first glance, it can seem overly complicated, so I'll add a list of things to be noted.

- All m-states have a number of outgoing arcs that's equal to the number of outgoing arcs of *all his states combined*
- The first state has no base part
- Following a path on the machine chart translates in "inheriting" the lookahead set
- Watching the graph, there are two visible subsets that represents the two possible paths towards the accepting states: one in the "upper level" (which has as a lookahead $\dashv$) and one in the "nested level". The first is composed of the states $\{I_0, I_3, I_8, I_2\}$ and the second one of the states $\{I_3, I_6, I_7, I_5\}$

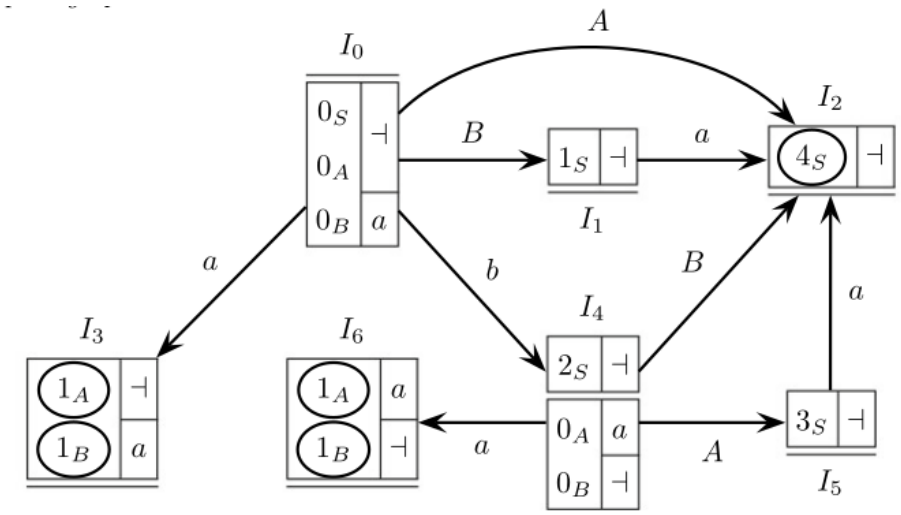


Another meaningful example

Without forgetting we are at the *first* step of our DPDA building, I add another good example.
We start from this machine sets:



We derive the pilot automaton with the algorithm described above:



Again, I add a short list of things to be noted

- the first transition labeled "a" is the perfect example of *following a paht makes you inherit lookaheads*;
- m-states I_3 and I_6 represents the same states, but with different lookaheads;

Deterministic check (also called ELR condition)

The three possible failures in deterministic check:

- shift reduce conflicts: both a shift and a reduce are a possible in a certain configuration. This means that in a pilot automaton for all the candidates $\langle q, \pi \rangle \in I$ so that q is final and for all the arcs $I \xrightarrow{a} I'$ that go out from I with terminal label a it must hold that $a \notin \pi$;
- reduce reduce conflicts: two possible (or more) reduce paths are possible in a given configuration. In a pilot automaton, it's formulated as: for all

the candidate $\langle q, \pi \rangle, \langle r, \rho \rangle \in I$ that q and r are both final it must hold that $\pi \cap \rho = \emptyset$;

- convergence conflicts: two different computations share a lookahead and a future state (this could cause reducing conflicts). This conflict is the most tricky to discover, as it can only be done analyzing the graph manually: the condition to be met is something like "no different states in a single m-state share a lookahead and a state".

Parser algorithm

We focus on the implementation that makes use of a vector stack, so a LIFO memory with has also and index-like access (like vectors. You should guess what a vector stack is). This enables the parser to save the exact position to return to when a reduce move is performed, accessing directly the cell pointed with the saved index.
The accepting procedure is quite obvious: the pilot automaton is traversed following the arcs when a new term is read, and reducing when the next term is equal to the lookahead char in the m-state. Accepting is when we reach a final state and we only have the axiom on the stack. This procedure is an incremental process: every time a m-state is leaved, on the stack are pushed all the possible candidates, and the next m-state will save the pointer (index) to the previous one and so on, so that at reduce time the parser can pop off all the states from the stacks just following the indexes.
(The official and formal way to describe the algorithm is leaved to the curious reader: page 271+ of the "Formal languages and Compilation" book from Reghizzi, Breveglieri, Morzenti)

4.2.4 ELL parsers

ELL = Extended Left to right Leftmost
ELL parsers are among the most basic top down parsers; the idea is to mimic the traversing of the machines network. The string is *progressively generated* using the grammar rules character by character until $\dashv$ is reached. ELL parsers relies on a stronger assumption than ELRs: so that the input string can identify the rule that produces it *starting from the first character*.
The most used technique is to enrich the arcs of the pilot automaton with "guide sets" that communicate the next characters that can be encountered. Relying on a stronger hypothesis means being effective in fewer cases: ELL parsers can't be built if

- Multiple transitions between states are defined. This simply means that ELL parsing isn't applicable to *all* the pilot automata where two or more states in the same m-state originate an arc with the same label;
- There are left-recursive derivation;
- The pilot automaton does not meet ELR condition (ELL represents a smaller set of languages).

These three conditon form the **ELL condition** for automata.

Parser Control Flow Graph

A PCFG is a representation of the automata that compose the machine network that *links* them all together, highlighting subsequent calls and recursive dependencies. From the PCFG it's easy to derive directly the parser code.
Starting from the machine net graph, building a PCFG means:

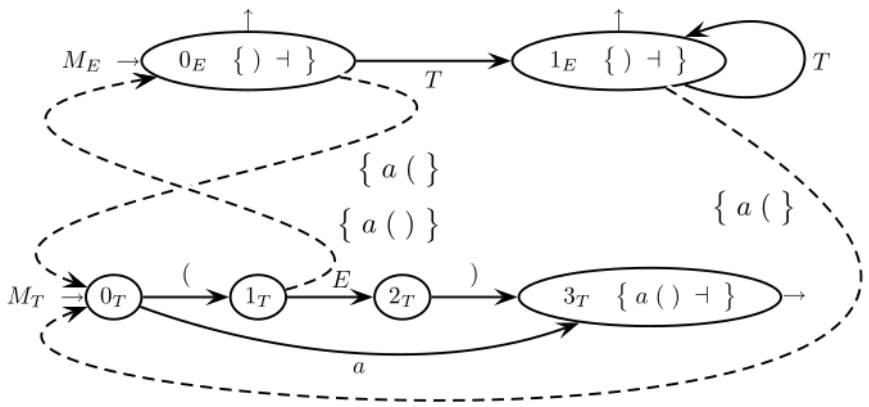
1. adding call arcs: these arcs connect the states that have an outgoing arc labeled as nonterminal with the first state of the machine that recognizes that nonterminal;

2. labeling call arcs: every call arc is enriched with a set of nonterminals that are the "guide set" for that call. This set of nonterminals is composed of all the followers of that call:

   - token that are the initials of the language generated by the submachine
   - the empty string if the language of the subm is nullable
   - the nonterminals that labels the subsubcalls to other submachines

3. enriching final states with so-called "prospect sets": prospect sets are the union of *all* lookahead sets of *all* the occurrence of that state in the pilt automaton. This union of sets must be found watching the pilot automaton.

Example We build the PCFG for the grammar in (5), so to have a continuity.



ELL test

Given the PCFG, we can easily check if an automata is ELL by exploiting a corollary of the ELL condition itself, which states that a machine network satisfies ELL iff for each state, all the outgoing arcs' guide sets are disjoint. The double implication means that building the PCFG and checking the disjointness of the guide sets proves if a machine network satisfies ELL or not.
This means we can avoid building the pilot automaton to check for ELL property

Possible implementation techniques

This short paragraph explains how to implement a predictive parser (but not why, tho) by means of two of the most common techniques.

- Recursive procedures: every submachine is encoded as a single procedure that scans the input tape char by char and that analyzes the input token as the automaton would: guide sets and arc labels are used to build the patterns to match the current character against. Termination of a subprocedure: acceptance; else an error is thrown. To recognize a nonterminal the associated procedure is called. (A lot of talking for an implementation technique that's really simple in practice: just modularize the code according to the machines in the network).
- Explicit stack: this method does not take advantage of the recursiveness to memorize the state of the application, instead it saves it on the stack directly. The parser can perform 4 different moves (here with "replace q with r" is intended pop q push r):
  1. scan: if the shift arc q $\rightarrow$ r exist, then q is replaced with r.
  2. call: move performed when a submachine is invoked. Pop of the current state, push of the first state after call (so if $q \xrightarrow{B} r$ replace q with r and push 0_B), push of the first state of the called machine.
  3. return: the next character is in the prospect set of the current state, which is final: pop.
  4. recognition: if we are in the final state of the axiom machine and the current character is $\dashv$, recgnize and halt. Otherwise, error and halt.

Lookahead lenghtening

Often it's sufficient to increase the lenght of the lookahed (to more than one token) to render a top down parser deterministic fom a non-ELL grammar. In this course, the only method is a "eagle eye" one where you spot the perfect lenght just looking at the PCFG and choosing which minimum lookahed set grants determinism, basing yourself on pure parser instinct.

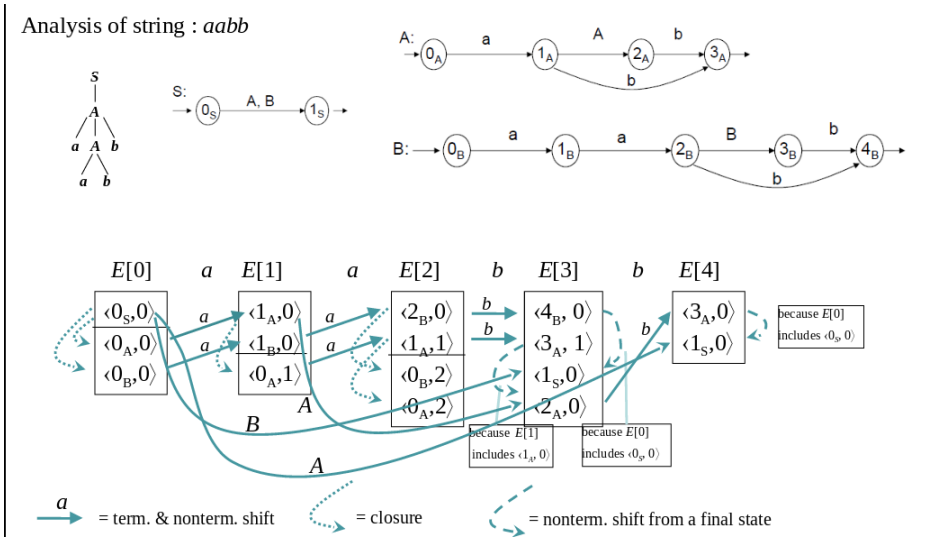
4.2.5 Earley algorithm

The Earley algorithm is a method to recognize strings from a grammar even if this is ambiguous. The pseudocode approach both the book and the slides take is very counterintuitive, in my opinion. So, I will try just to explain *how practically* a Earley recognizers is built and used. Keep in mind that this procedure should build *a software*.

First, initialize the vector of candidates to E = { $\langle 0_s, 0 \rangle$ } that represent the initial state and the first value
Then, with n taken as the lenght of the string to recognize, we initialize all the elements of E[1..n] to the empty set.
We complete the first element E[0]. Completing an element (applying the *completion* operation to it) means computing the closure of the state (and adding to the E[i] set of candidates the closure's states) and calculating also all the states

that are reachable through a nonterminal shift (watch out: with nonterminal shift we are referring to the operation of "arriving on E[i] with a nonterminal shift", not leaving this state), adding them also to the state.
We then compute, for every token in the string, the *terminalshift* for every element in the vector (terminalshift means nonterminalshift with terminal symbols. For every candidate in previous state that can terminalshift to a candidate in the current state add that transiction [represented as the candidate $state, index - 1$] to this vector element) to build the shifting operation, then we complete it.

This algorithm is similar to the method to build a pilot automaton, but instead of lookahed sets just the position in the vector of the caller state is memorized. The string is accepted if the last element of the vector is of type $\langle finalstate, 0\rangle$: this means that the condition is accepting (finalstate) and the whole string is accepted (0).
These example and notation should clear all remaining doubts:



Chapter 5

Translation and Static Analysis

What is translation (mathematically speaking)? Let Σ and Δ be two alphabets (*source* and *target* respectively); a translation is a correspondence between source and target strings, that is formalized by binary realtion between the source and target universal languages (that is Σ^* and Δ^*). Formally, a translation is a subset of $\Sigma^* \times \Delta^*$. So, it can be also seen as a function, and inherits all the function's peculiar properties (like surjectivity, injectivity and all the terminology that concerns dominion, image and inverses).

5.1 Purely Syntactic Translation

5.1.1 Translation Grammars

When the source language is generated by a grammar, one of the most natural translation is to map each subtree to a target subtree. This is done by building a *translation grammar* that have the same number of rules of the source language grammar and differs only in the terminal symbols in the right part of the rules. The nonterminals symbols are *the same and in the same order*.
A typical example of translation grammars come from bynary functions notation, when translatin the classic infix notation ($A \times B$) into a pre/postfix polish notation, like $\times AB$ that's way more concise and understandable, being less prone to error caused by operator's precedence.
We have so a grammar for the infix notation expression, and one that describes the target language, a prefix polish notation:

$$\begin{cases} E \to E + E \\ E \to E * E \\ E \to (E) \\ E \to i \end{cases} \quad \begin{cases} E \to \text{add } E\ E \\ E \to \text{mult } E\ E \\ E \to E \\ E \to i \end{cases} \tag{5.1}$$

So we build our translation grammar this way:

$$\begin{cases} E \to \frac{\epsilon}{add} E \frac{+}{\epsilon} E \\ E \to \frac{\epsilon}{mult} E \frac{*}{\epsilon} E \\ E \to \frac{(}{\epsilon} E \frac{)}{\epsilon} \\ E \to \frac{i}{i} \end{cases} \tag{5.2}$$

Ambiguity

If the source grammar is ambiguous then the translation scheme is ambiguous: if a sentence can be generated bby more than one syntax trees, obviously the translator should take care of this multi-valued rules. Ambiguity in the translation grammar can also derive by duplicated rules, that can be mapped to different values: in this case, both the source and target grammar are unambiguous, but multiple translation rules correspond tothe same source rule, rendering the translation grammar multi-valued ⇒ ambiguous.
Summing up these two concepts, a translation grammar $G_\theta = \{G_1, G_2\}$ is unambiguous (single-valued) if

1. G_1 is unambiguous;
2. No two rules of target grammar G_2 map on the same G_1 rule.

5.1.2 Pushdown Transducer

We have defined a translation between grammars, now we have to define a machine that performs it. Such machine is a *pushdown transducer*, so a pushdown automaton with output capabilities. It has the same definition of a PDA, enriched with the output (target) alphabet, and with a different transiction function: in fact, this should now take into account also the output of the machine, not only the state transitions.
A translation relation is defined by a translation grammar iff it is computed by a nondeterministic pushdown transducer. This property ensure the equivalence of the two methods, and also grants the possibility of defining a machine starting from the grammar as in traditional parsing.

Single state nondeterministic transducer

Theoretically speaking, the simplest extension from a parser to a transducer is built directly associating to the grammar rules some move, this time adding the write operation. We obtain a nondeterministic single state transducer, with not the minimal practical value.

ELL grammars translation

ELL parsers can be implemented by recursive function calls (classic top down approach), as explained in (21.4.3). Extending the recursive calls parsers with write functions just after the submodule call is enough to render the parser a transducer. Simple as that. Remember that ELL parsers rely on a stronger assumption then ELR, so they're less complex and easier to extend; they're also applicable to a reduced number of cases.

ELR grammars translation

ELR parsers are not easily extended as their ELL brothers. In fact, just adding the output action after a shift operation could lead to contradictory writes, becaouse of the inherent nondeterminism of the bottom up approach to parsing. Instead, the reduce operation grants the recognition of a single grammar rule, and (due to the total mapping of the source to the target rules) one target rule can be identified and selected for output.
This is easier said than done: a reduction could take place before another that may need to write a characte *before* the current one; to avoid this kind of phenomena, the translation grammar must be in **postfix normal form**: a translation grammar is in postfix normal form when all its rules are of the kind

$$A \to \Sigma^* \delta^* \tag{5.3}$$

so they produce the output strings *only* at the end of the rules as a suffix of the production, and not in the middle of it.
To convert a grammar in the postfix normal for it's enough to replace all the terminals in the translation that violates the postfix form with new nonterminals that simply produces the substitued character (this change should be reflected in the target grammar also). A translation grammar in the postfix normal form ensures that, when implemented via an automaton, the output operations take place only at reduction time. This trnsformation makes sure that the construction of a pilot automaton for a *transducer* is identical to the one for a *parser* with just the write operation added to the m-states that have a final state in them.
In some particular cases (that are indeed very rare, according to the literature) normalizing a grammar can introduce shift reduce conflicts due to the introduction of initial - final states in the automaton.

Regular Translation

Regular Languages Translation As regular expression are a proper subset of CF grammars, they have a special family of transducers. Consider this translation grammar:

$$G_\tau = \begin{cases} A_0 \to \frac{a}{c} A_1 \mid \frac{a}{c} \mid \frac{a}{b} A_3 \mid \epsilon \\ A_1 \to \frac{a}{c} A_2 \mid \epsilon \\ A_2 \to \frac{a}{c} A_1 \\ A_3 \to \frac{a}{b} A_4 \\ A_4 \to \frac{a}{b} A_3 \mid \epsilon \end{cases} \tag{5.4}$$

Things to be noted:

1. the grammar translates string of "a" in strings of "b" when the number of "a" is even
2. the grammar translates string of "a" in strings of "c" when the number of "a" is odd
3. to "count" the number of letters, the grammar constructs two loops ($A_0 \to A_1 \to A_2 \to A_1$ and $A_0 \to A_3 \to A_4 \to A_3$) to either create an odd number

of letters or an even one. This can be noted observing which rules contains the empty string in the loop

4. the grammar is right linear (so? it's equivalent to a regular expression)

5. the grammar is in prefix normal form (does this have some relevance now? no)

As well as a right linear grammar can easily be converted in a regular expression, we can derive a *regular transducer* that translates the language of this translation grammar. It's pretty straightforward:

$$e_\tau = \left(\frac{a^2}{b^2}\right)^* \cup \frac{a}{c}\left(\frac{a^2}{c^2}\right)^* \tag{5.5}$$

Two Input Automaton Starting from the defined "regular translator" in the previous paragraph, we can see that the combination of terminals of the source and target expressions can be seen as a terminal alphabet itself (it will be a terminal alphabet built on the cartesian product of two alphabets, but the idea is there). So, with the regular expression $\leftrightarrow$ FSA equivalence in mind, we can define a FSA that recognize (validates) the translations of the source expressions. This machine will ha ve *two* input tapes, one for the source and one for the target terminals, and will recognize the input when it's a valis translation. The only thing that's odd about 2I-FSA is the way empty strings are managed: if the transiction function accepts a empty string as input character *on one of the two tapes*, the machine could still be blocked by the reading on the other one.
As an example we produce the 2I-FSA that recognizes the translation of the above regular translation.

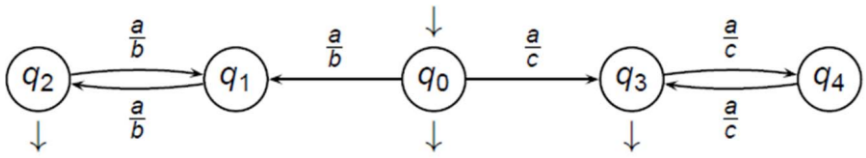
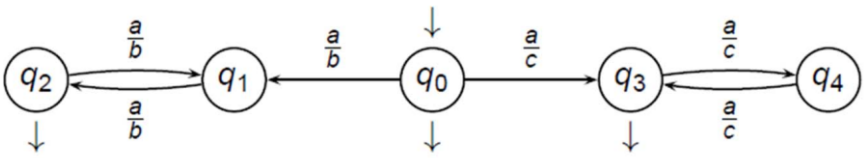


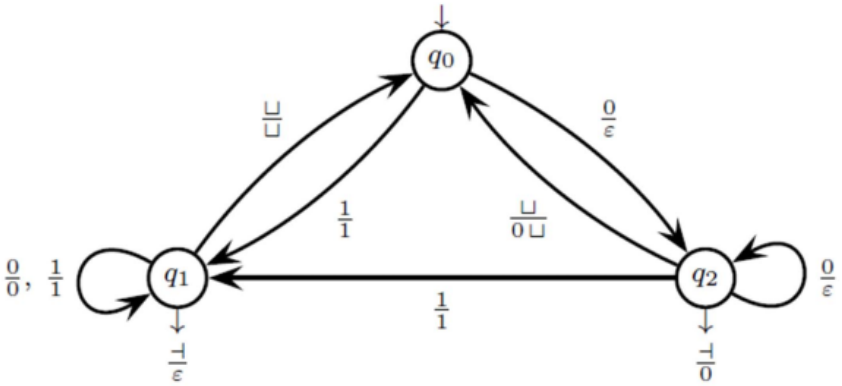
Figure 5.1: To be noted: this automaton is deterministic, because the two moves exiting from q_0 have different labels

Finite Transducer We want an automaton that formalizes the translation (REMINDER: A 2I-FSA DOES NOT TRANSLATE, BUT VALIDATES A TRANSLATION) so that starting from a string on an input tape prints the translated string on an output one. The schema is very simple:



Is it familiar? It's the *same* formalization as the 2I-FSA that validates the translation, but this time the labels have a **totally different meaning**: the bottom character represents the one printed on the output tape. So, this time the automaton is nondeterministic, because of the two arcs exiting from q_0 have the same input character; only one of the two will make the automaton finish in an acceptance state; this also means that *no output can be produced until a final state is reached.*

Sequential Transducer Ok, now we want a machine that translates input *real time*, so while scanning the input tape (to overcome the limitation of finite transducers that must reach a final state to print). The sequential transducer model differs from the aforementioned ones in the theoretical representation (it makes use of three functions: a transition one, an output one, and a *final* one) but not in the graphical rep. The core idea is very similar to the finite transducer but the output characters are *actually* printed as output. This model needs an additional function that computes the "string terminator": the so called final function, in fact, just appends to the translated string a marker that signals the end of the translation, and the final state reached by the computation.
This may seem overly complicated, let's use an example to clarify: a sequential transducer that eliminates all the leading zeroes from a string. The input string are built on a alphabet composed of three characters, $\{\ 0, 1, \sqcup\ \}$ where $\sqcup$ represents the separator between values. If a string is composed solely by zeroes, a single zero is outputted. I'll skip the definition of the regular translation expression for this automaton, to present directly the machine:



The above machine can be explained section by section:

1. state q_2 carries out the meaningless zeroes. She outputs nothing, never, exception made for when the terminator is read, when a single zero is written.

2. state q_1 takes care of the meaningful part of the string. It's reachable through a scan of the character "1", that signals the start of the meaningful part of the string; its only task is to traduce the string as is.

3. q_0 interpret blank spaces.

5.2 Syntax Directed Translation

The compilation process cannot be performed only through syntactic methods. A *meaning* must be assigned to sentences, and (while this can be accomplished in a formal way) the only "structured reading and translation" of a sentence is not enough to do so. Syntax directed translators starts from the syntax trees of a grammar and enrich them with semantic attributes.

5.2.1 Attribute Grammars

Attribute grammars have the same computing expressiveness of Turing machines (they are very powerful). They are the third step of computations during a compilation procedure: they are preceded only by lexical and syntactic analysis. The purpose of attribute grammars is to help design the *decorated syntax tree* that enriches the normal s.t. with a semantic value.
Let's run an example:

Meaningful example We'll build an attribute grammar for the language $L = \{0,1\} \bullet \{0,1\}$ that represents decimal numbers in a fixed point fashion. The grammar that generates it is

$$G_L = \begin{cases} N \to D \bullet D \\ D \to D\,B \mid B \\ B \to 0 \mid 1 \end{cases} \tag{5.6}$$

Where N is the axiom, D is the string, B is the single bit. The goal of our attribute grammar is to compute the decimal value of each part of the number, assigning the real weight to the single bits. To do so, we need to keep track of two fundamentals attributes:

1. The value of the numeric parts of the string: the N, D and B nonterminals have all this attribute, because they are all *semantically* numbers;

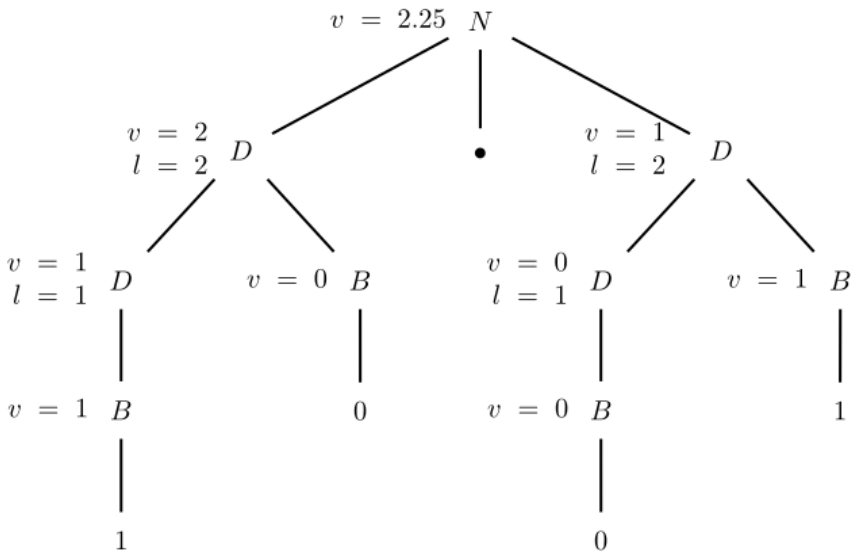
2. Their position in the string; an easier way to compute so is to keep track of the string lenght. Only nonterminal D needs this attribute. N does not need it because it's just a concatenation of two values.

So, now we have a grammar and a set of attributes with a semantic value that must be linked to certain nonterminals. To do so, we can define functions

(associated to every production) that computes the value of the nonterminal of the reduction. In this case:

$$G_{Lext} = \begin{cases} N \to D \bullet D \text{ sum of the right with the left part: } v_0 = v_1 + v_2 \times 2^{-l_2} \\ D \to D\,B \mid B \text{ also consider the length increase: } \begin{cases} v_0 = 2 \times v_1 + v_2 \\ l_0 = l_1 + 1 \end{cases} \begin{cases} v_0 = v_1 \\ l_0 = 1 \end{cases} \\ B \to 0 \mid 1 \text{ just assign the final value to the nonterminal: } \begin{cases} v_0 = 0 \\ v_0 = 1 \end{cases} \end{cases} \tag{5.7}$$

A detailed explanation of what l_0, l_1, v_0, v_1 and v_2 are is needed: the letter is the attribute we're referencing ($l =$ lenght and $v =$ numeric value) and the pedix symbolizes the *cardinal position of the nonterminal we are considering*: this means that all the attributes that have the zero pedix will be the results of the computation, because they are the attributes of the left part of the grammar rule. Vice versa, v_2 represents the numeric value of the *third* nonterminal in the rule.
The computation evolves around the tree in this way:



Left and Right Attributes

The definition of left and right attributes is as tricky as fundamental for the comprehension of attribute grammars. Let's try to figure it out:

Left attributes We want to generate a value from a production, what did we do in the previous example? We computed the value for the right part recursively till a fixed point (the production that has associated a function that produces a costant, such as $l_0 = 0$). In this case, l_0 is a left attribute,

because it's associated with a left part of the rule. Left attributes are also called "synthetized" because they usually represents the result of the computation.

Right attributes We say that an attribute is "right" or "inherited" if it's associated to a right part of a rule, so the functions that calculate it has as output not a value that goes in the left part of the rule, but stays in the right part.

Still nebolous definitions, and no practical value. Let's make an example, *read it carefully*:
we have as input a string of words, separated by blank spaces (we will represent them with $\dashv$). We want to arrange the text into rows of fixed length W and compute for each word the column of its last character. We assume no words have length that's greater than W.

$$G = \begin{cases} S \to T \\ T \to T \dashv T \\ T \to V \\ V \to cV \mid c \end{cases} \tag{5.8}$$

The "c" terminal respresents any character. Which attributes are needed?

- *last* stores the column of the last letter
- *length* stores the length of the word
- *prec* is the column of the last character of the previous word

These three attributes are all we need to describe the set of words, element by element. The idea is to compute the last character of each word as: $last = prec + 1 + lenght$. It's pretty straightforward spotting the only right attribute: *prec*. In fact, it's extracted from the left part of the rule. Let's define all the functions associated to the rules.

Rule	Left attributes	Right attributes
$S \to T$		$prec_1 = -1$
$T \to T \dashv T$	$last_0 = last_2$	$prec_1 = prec_0$ $prec_2 = last_1$
$T \to V$	$last_0 = (prec_0 + lenght_1 + 1 > W)?lenght_1 : (prec_0 + lenght_1 + 1)$	
$V \to cV$	$lenght_0 = 1 + lenght_1$	
$V \to c$	$lenght_0 = 1$	

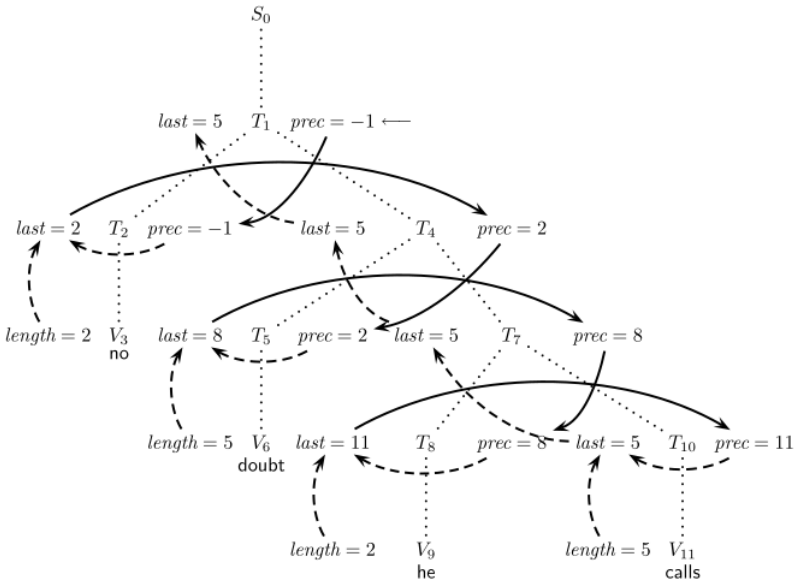
Let's describe what's each function do:

- Left attributes:

  - $last_0 = last_2$ stands for "the last element of two words is the last element of the last word".
  - $last_0 = (prec_0 + lenght_1 + 1 > W)\,?\,lenght_1 : (prec_0 + lenght_1 + 1)$ this function (expressed as a inline if instruction) verifies if we've reached the end of the line, composed of W coloumns. If we are (the condition is true) we've reached the first word of the next line, so the lasta character of this word will be on the last coloumn. Otherwise, it will be taken into account also the lenght of the preceding word.
  - $lenght_0 = 1 + lenght_1$ and $lenght_0 = 1$ are classical recursive calls to form the words lenght.

- Right attributes:
  - $prec_1 = -1$ initializes the first word's predecessor.
  - $prec_1 = prec_0$ and $prec_2 = last_1$ sets the value of the predecessor for each word. Note how the information, in this case, flows downward to the leaves: to be set are the values of attributes that only later in the execution will be utilized. In this case: we're setting the predecessor of each word: predecessor of the first = last one's predecessor and second one's predecessor = last character of the first.

The final decorated tree is:



Let's formalize all this reasoning:

Formal Definition of an Attribute Grammar

An attribute grammar is composed of:

1. A context-free grammar $G = \langle V, \epsilon, P, S \rangle$ with her set of terminals, non-terminals and the axiom. It is convenient to NOT have the axiom in the right part of the rules

2. A set of semantic attributes, associated with terminals and nonterminals. This set is separated in left attributes and right attributes: *the two sets are disjoint. No right attribute can be left attribute in another production and vice versa.* The set of attributes of nonterminal D is also referred to as $attr(D)$

3. A set of semantic functions, each associated with a grammar rule. The grammar rule associated with each function is called the support of that function. In general, several functions can have the same support, and the set of functions supported by production p is $fun(p)$. Semantic functions are formally defined as:

$$\begin{cases} \text{the support: } p = D_0 \to D_1 D_2 ... D_r \\ \text{the function: } \sigma_k = f(attr(\{D_0, D_1, ..., D_r\}) \backslash \{\sigma_k\}) \\ r \geq 0 \\ 0 \leq k \leq r \end{cases} \tag{5.9}$$

4. All the semantic functions must respect:

   (a) For each left attribute of a left nonterminal of a production there is *exactly one* function that defines it

   (b) For each production, no functions defines *right attributes for the left side nonterminals*

   (c) For each production, no functions defines *left attribute for the right side nonterminals*

   (d) For each right attribute of a right nonterminal of a production there is *exactly one* function that defines it

   These four rules better regulates the "attributes placements and initialization": in fact, they states that all the left-hand side nonterminals can have only left attributes and also that such attributes must be uniquely defined; also they states that for all right-hand side nonterminals only left attributes are possibly defined and *also* they are uniquely defined. The other attributes are called *external attributes*, because they're not comupted in the function.

Dependence Graph The graph showed in 23.1.1 is a complete dependence graph obtained by "summing" all the graphs of its subtrees. The single dependence graph can be referred to semantic functions, or to productions (so *to all* the semantic functions that have that rule as support). The decorated syntax tree, ultimately, is the complete dependence graph of all the productions.
How to build a decorated tree, even though it's far from Christmas: just link the arguments of the function to the attribute that function initializes. So, if the function is $attr_0 = f(\{attr_1, attr_2\})$ in the decorated graph we will have something like $attr_1 \to attr_0$ and $attr_2 \to attr_0$.

Grammar Validation

Given an attribute grammar defined as before, if the dependence graph of the tree is acyclic, there's a set of attribute values consistent with the dependences. A grammar is loop free id *every dep graph of every tree* is acyclic. How to test if a grammar is acyclic? Given the infite number of syntax trees, computing all of them to check for cycles is not viable. Instead, it's enough to ensure certain properties, that are sufficient to guarantee the desired condition of acyclicity.

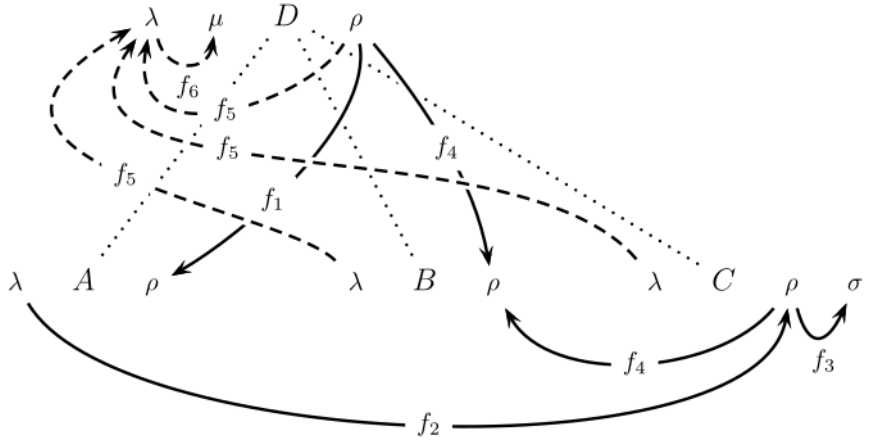
One Sweep Grammar "One sweep" is a property that ensures that a grammar can be evaluated with a depth first visit. What does this mean? It means that *it's possible to perform attribute evaluation during a depth first visit.* Why should this be checked? Because the depth first visit operates this way:

1. before entering a subtree compute all right attributes of the root of that subtree
2. when "emerging" from a visit of a subtree, compute all the left attributes of the root of that subtree

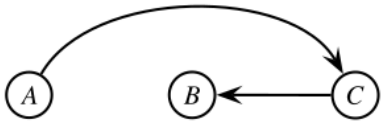
Both these two operations can get stuck in the evaluation of certain attributes, because of the possible functional dependences among attributes. A one sweep grammar is ensured to be free of such problems.
So how to check for "onesweepness"? We shall recall the definition of the *dep* set, and introduce the *sibling graph*:

1. The nodes are the right symbols of the productions $\{D_1 \dots D_r\}$.
2. Arcs $D_i \rightarrow D_j$ are drawn between nodes that have an attribute dependency like $\sigma_{D_i} \rightarrow \theta_{D_j}$. The nodes in the siblings graph **are not** the ones of the dependence graph of the production.

Let's generate a sibling graph for a dummy rule, $D \rightarrow A\,B\,C$ that has the dependency graph as below:



The resulting sibling graph is:



For a grammar to have the one sweep property it must hold, for all the productions:

1. dep_p contains no circuits;
2. dep_p contains no path that goes from a left attribute to a right attribute of the same node;
3. dep_p contains no arc from the left attribute of a father to a right attribute of a children;
4. $sibl_p$ contains no circuits.

Validation of a Grammar through a One Sweep Evaluator

The algorithm we described before to visit a tree depth first can be formalized to generate a one sweep evaluator, a machine that carries out all the computation of an attribute grammar. REMEMBER: the evaluator built this way *relies* on the assumption that the grammar is one sweep.
A semantic procedure is written for each *nonterminal*: this procedure has as arguments the subtree generated by that nonterminal and his right attributes. The procedure must, for every support rule $p : D_0 \rightarrow D_1 \ldots D_n$ with $n \geq 0$

1. Choose the TOS (Topological Order for Siblings) for $\{D_1 \ldots D_n\}$ wrt the sibling graph for that nonterminal
2. Choose the TOR (Topological Order for Right attributes) for all the right attributes of all symbols $\{D_1 \ldots D_n\}$ wrt the dependence graph of that symbol
3. Choose the TOL (you guess. Hint: opposite of right...)

These three ~~demigods~~ orders prescribe *how to arrange the instructions* in the procedure body. To explain that, we'll make as example the procedure that is generated by the rule in the previous example, $D \rightarrow A\,B\,C$, with the dep graph as shown before. This grammar is easily proven one sweep (the sibling graph has no cycles and no cycles or cross dependecies are seen in the dep graph).
The procedure starts of choosing a TOS: due to the sibling graph's arcs, the order choosen is $\{A, C, B\}$.
Then she chosses the TOR, taking in consideration the dependencies shown in *dep*: $\rho \rightarrow \sigma$ is the one.
Last but not least TOL: again, watching the dependencies, we have $\lambda \rightarrow \mu$.
Once these three orders are identified, the procedure can be written: I'll try to

give an abstract explanation of *how* the three orders are used (because I hate pseudocode) then I'll add a pseudocode example.
Subtrees are computed in TOS order, and for each subtree the "computational pattern" is

1. compute right attributes in TOR order using the functions highlighted in *dep*
2. call the subprocedure of that symbol passing as argument also the attribute to popolate
3. compute left attributes with the assigned functions in TOL order

So in this case, the procedure becomes:
def D (in: root, ρ_D, out: λ_D, μ_D):
$\rho_A = f_1(\rho_D)$
$A(root_A, \rho_A; \lambda_A)$

$\rho_C = f_2(\rho_A)$
$\sigma_C = f_3(\rho_C)$
$C(root_C, \rho_C, \sigma_C; \lambda_C)$

$\rho_B = f_4(\rho_C, \rho_D)$
$B(root_B, \rho_B; \lambda_B)$

$\lambda_D = f_5(\lambda_B, \lambda_C, \rho_D)$
$\mu_D = f_6(\lambda_D)$

5.2.2 Combined Syntax and Semantic Analysis

In some cases syntax analysis and attribute evaluation (semantic analysis) can be performed at the same time. For each class of languages we've seen, a different approach can be developed.

Regular Laguages - Lexical Analysis with Attribute Evaluation

If the language is regular, then the *scanner* or *lexxer* 's task is to identify and isolate *lexemes*, the minimal tokens of a language that can be given a semantic weight. The scanner can also give attributes to these tokens, and compute them.

LL Languages - Attribute Recursive Descent Translator

If the syntax is suitable for top down parsing (is LL) and the grammar is one-sweep, it's necessary to add just another hypothesys to make sure that the parsing and the semantic analysis can procede step by step together. The condition to be satisfied is the L condition, that states:
a grammar satisfies the L(eft to right) condition if

- It's one-sweep
- The sibling graph (of every support) contains **no arc** so that $D_i \to D_j$ with $j > i \geq 1$ (so no *backward dependency*)

If the syntax is LL and the L condition is satisfied (what is wrong with the letter L I would like to know) then a *attribute recursive descent translator* can be built [1].

5.3 Static Analysis

5.3.1 Compilation of a Program

Compilers are composed of two part: a front end compiler (that translates the actual source code into a "lower level code". This is made possible by a scanner + parser pipeline) and a back end compiler (that takes the lower level code and generates the executable). After the front end compiler has generated the intermediate representation of the program, this goes through three additional phases:

1. **Verification** correctness check
2. **Optimization** target specific improvements to enhance performance
3. **Scheduling** modification of the produced code to fully use the pipelines/parallel capabilities of the machine

Control Flow Graph

To analyze the program in these three further phases, a schematic representation of the program itself is needed. This representation is usually a control flow graph, that stands for an automaton. Needless to say, this automaton is the same *formal structure* of the ones used to interpret the source code, but has a *totally diffferent meaning*: this automaton in fact describes the program itself and the use it makes of the variables.

How to Build a CFG CFGs are simple def-use graphs. They do *not* represents the logic unfolding of the program, but just the usage of the memory. The following abstractions are utilized:

- All the conditional statements are not represented
- "Instruction B follows instruction A" is drawed out as the classic arc
- All operations are reduced to two sets: *def* and *use*. These two sets represents the group of variable to which a value is assigned and the group of variables which value is read

So far, we've defined an automaton with some sets. But which strings does this automaton recognizes? A string recognized by an automaton described by means of a CFG is a possible *execution trace* of the program, or the execution path that traverse a valid sequence of instructions. In these pages, we will consider onyl *intraprocedural* static analysis, that is performed *in isolation* on every procedure.

[1] I know, it sounds like a General Zod weapon, doesn't it?

Conservative Approximation The choice to ignore the unreachable path enables the analyzer to reach unreachable segments of code. This could lead to pessimistic outcomes (like a program is deemed invalid or with errors, but these errors are all located in an unreachable segment). This condition is called conservative approximation because "pessimistic conclusions" are far preferrable rather than "optimistic conclusions": the latter can lead to assign a smaller-than-needed amount of resources, while the former never misses an error.

Liveness Intervals Analysis

Variable liveness analysis is used to discover definition errors or clashes or unreferenced accesses. A variable a is said to be "live" on the *exit* of a node p if there exists a path between p and another node q so that a does not appear in *any* def sets of the traversed nodes, but it appears in the use set of q.
This definition is usually referred to as "live-out", because it takes in consideration the exiting arcs from a node. A specular formulation ("live-in") can be given. Example of usage: if a variable is defined in a certain node, but it's not live-out of that node, the assignement can be deleted without altering the program behaviour.

Computing Liveness Intervals - Data Flow Equations The scary name of "data flow equations" is given to a rather simple approach to the problem of calculating all the $live_in$ and $live_out$ sets for each node of the control flow graph. The procedure is inherently recursive and starts from the final nodes. It unfolds this way:

$$\forall p \in CFG \,|\, p \in Finals \Rightarrow live_{out}(p) = \emptyset \tag{5.10}$$

So (quite logically) if a node is final his live out variable set is empty. Next equation (that represents the "recursive call") makes use of the *succ* set, representing all the *immediately reachable nodes* from a given one.

$$\forall p \in CFG \,|\, live_{out}(p) = \bigcup_{\forall q \in succ(p)} live_{in}(q) \tag{5.11}$$

$$\forall p \in CFG \,|\, live_{in}(p) = use(p) \cup (live_{out}(p) \backslash def(p)) \tag{5.12}$$

The sketched procedure is quite simple: starting from the last nodes, the CFG is traversed backwards adding time to time to each node his $live_{in}$ and $live_{out}$ sets. For every node that's not final, the $live_{out}$ set si composed of all the variables in the $live_{in}$ sets of his successors. The $live_{in}$ set instead is composed of all the used variables in that node plus the $live_{out}$ set purged from the redefined variables (doubts? check the "liveness" condition definition for a variable).

Applications Liveness analysis is performed during

- Memory allocation: if two variables *interfere* with each other (so they're both present in the live-in set of a node on the CFG) they must be both present in memory when such point is reached. If this does not happens, a cell of memory can be used to store both the variables, at different times. This is a useful performance enhancement.

- Useless definitions: as exemplified before, a definistion is useless if the defined variable is not in the live-out set of the instruction defining it. A condition like this emerges from the liveness analysis, and can lead to heavy code-cleaning.

Reaching Definition

A similar analysis to the liveness intervals one is the "definitions reach" one. This one also is performed on variables, and aims to search the point reached from a given variable definition. The formal definition of "reacheness" is very similar to the "liveness" one too: a definition in instruction p is said to *reach* an instruction q if there's no instructions on the $p \rightarrow q$ path that defines the same variable.

Computing Reaching Definitions - Data Flow Equations Again, these "equations" shape the recursive calls and fixed point of the ideal procedure that calculates all the reaching span of a definition. We need an additional set to be able to define the reach of a definition in an agile way: it's the *suppressed* set. This set is composed of all the variables in the def set that are also in some other instructions def set. This set represents all the "overwrites" that an instruction does, invalidating the value written in that variable at that point.
As in the liveness analysis, reach sets are defined wrt the direction of the data flow: there's an *in* and an *out* reach set for each node. Instead of the successors set, we ned the predecessors set this time. (It's the same algorithm as the liveness one, but it takes the reverse approach.)

$$i = Initial \,|\, in(i) = \emptyset \tag{5.13}$$

$$\forall p \in CFG \,|\, in(p) = \bigcup_{\forall q \in pred(p)} out(q) \tag{5.14}$$

$$out(p) = def(p) \cup (in(p) \backslash sup(p)) \tag{5.15}$$

See? IT'S THE SAME AS LIVENESS ANALYSIS!

Applications

- Constants propagation: when a variable (or a constant) is defined, the compiler can watch for the reaching of its definition. If the definition is sufficently long, then the constant value can be substituted into the variable accesses, transforming this way many instructions that require memory access to an immediate one (faster). Also, costant propagation can render a function full-immediate: this means that all his parameters are immediate (constant) values, and the returned value can be calculated at compile time (more time saved). This is valid also for conditional expressions! Entire unreachable branches can be detected and eliminated.

- Availability check: the reachness of a definition can be used to detect accesses to variables that are not defined yet.