Universita' degli studi di Trento

Linguaggi Formali e Compilatori

Gianni Zampedri

Contents

Chapter 1

Introduction

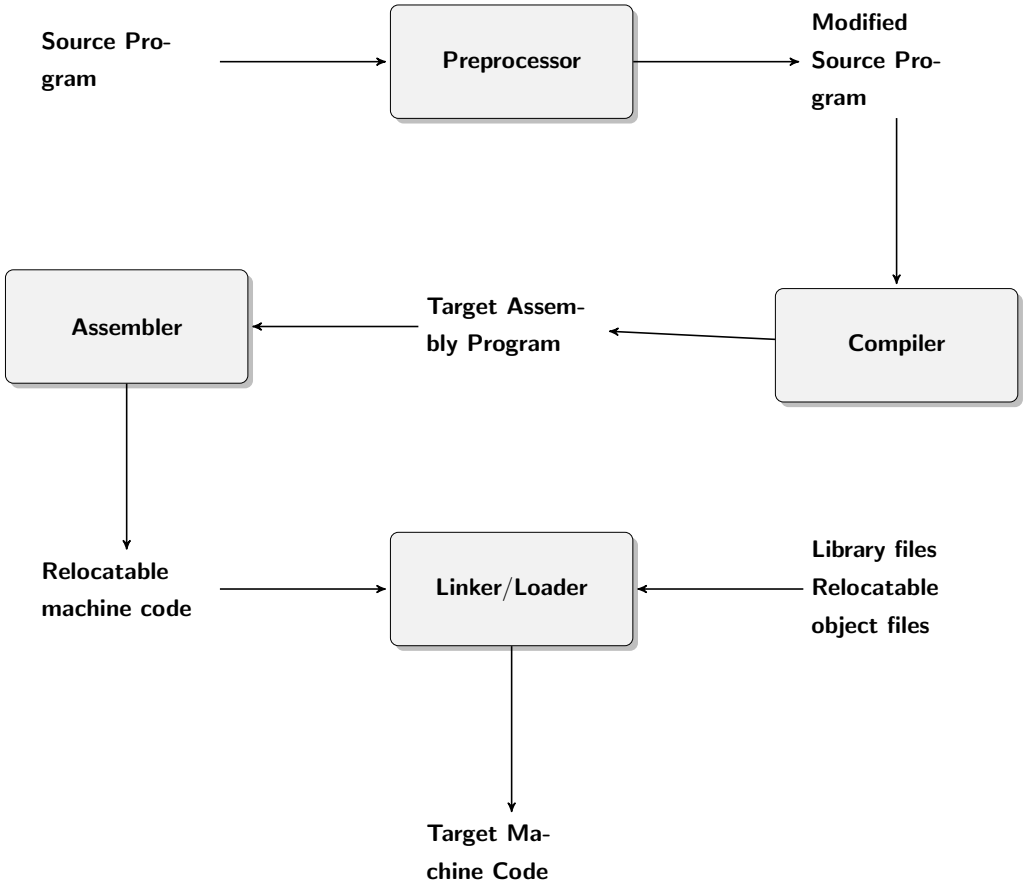
Programming languages are notations for describing computations to people and to machines.
But, before a program can be run, it first must be translated into a form in which it can be executed by a computer; the software systems that do this translation are called **compilers**.

1.1 Language Processor

A compiler is a program that can read a program in one language (the *source* language), and translate it into an equivalent program in another language (the *target* language). An important feature of the compiler is to report any errors that it detects during the translation process.
If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.
An **interpreter** is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user; an interpreter can usually give better error diagnostic than a compiler, because it executes the source program statement by statement.

NB: Java language processor combine compilation and interpretation; a Java source program may first be compiled into an intermediate form called *byte-codes*. The bytecodes are then interpreted by a virtual machine; a benefit of this arrangement is that bytecodes compiled on one machine can be interpreted to another machine.

A source program may be divided into *modules* stored in separate files. The task of a **preprocessor**, a separate program, is to collect the source program; it may also expand shorthands, called *macro*, into source language statements.
The modified source program is then fed to a compiler. The compiler may produce an assembly-language program as its output (easier to produce and to debug), which is then processed by a program called an **assembler** that produces relocatable machine code as its output.
Large programs are often compiled in pieces; the **linker** resolves external memory addresses, where the code in one file may refer to a location in another file. The **loader** then puts together all of the executable object files into memory for execution.



1.2 The Structure of a Compiler

A compiler can be seen as a single box that maps a source program into a semantically equivalent target program; there are two parts to this mapping:

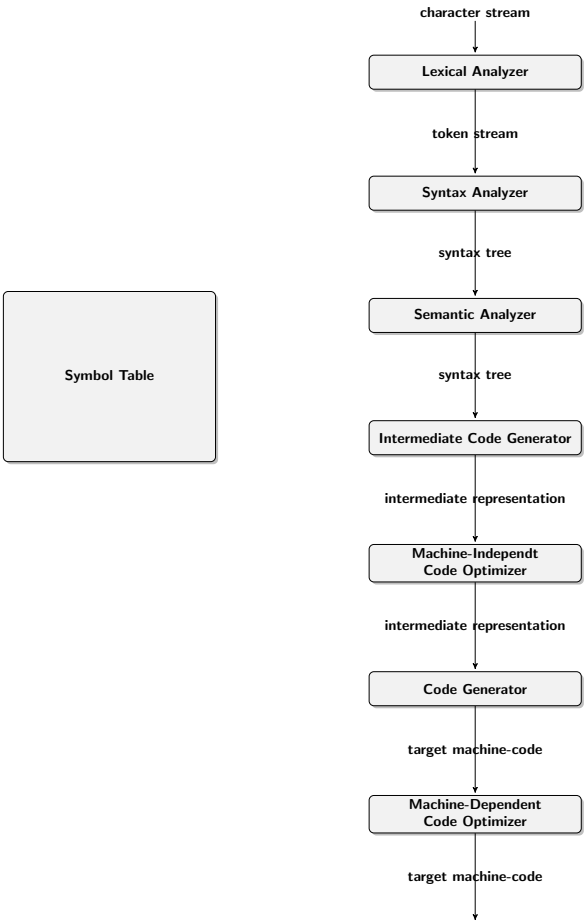
- analysis;
- synthesis.

The **analysis** part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. It the analysis part detects thath the source program is sintatically or semantically incorrect, then it must provide informative message. The analysis part also collect

information about the source program and stores it in a data structure called a *symbol table*.

The **synthesis** part constructs the desired target program from the intermediate representation adn the information in the symbol table.

If we examine the compilation process in more detail, we see that it operates as a sequence of phases, each of which transforms one representation of the source program to another.



In practice, several phases may be goruped together, and the intermediate representation between the grouped phases need not to be constructed explicitly. The symbol table, which stores information about the entire source program, is used by all phases of the compiler.

1.2.1 Lexical Analysis

The first phase of a compiler is called *lexical analysis* or *scanning*. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called *lexemes*. For each lexem, the lexical analyzer produces as output a *token* of the form

$$(token - name, attibute - value)$$

that it passes on the following phase, syntax analysis. In the token, the first component `token-name` is an abstract symbolt that is used during syntax analysis, and the second component `attribute-value` points to an entry in the symbol table for this token.

For example, suppose a source program contains the assignment statement

```
position = initial + rate * 60
```

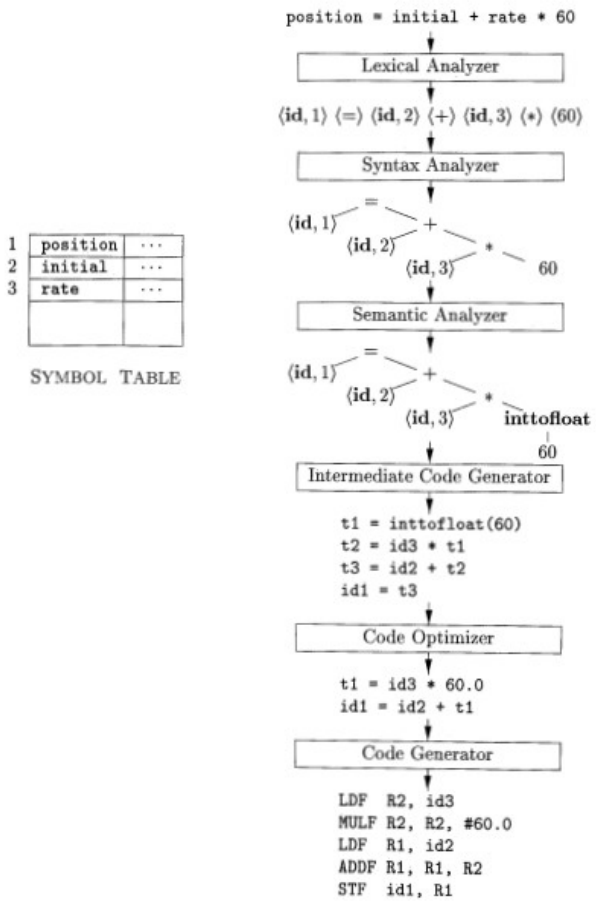
After lexical analysis, the representation would be a sequenc e of tokens as following:

```
<id, 1> <=> <id, 2> <+> <id, 3> <*> <60>
```

where, bringing for example the token `<id, 1>`, `id` is an abstract symbol standing for *identifier* and 1 points to the symbol-table entry for `position`; the token names =, + and * are abstract symbols for the assignment, addition and multiplication operators.

1.2.2 Syntax Analysis

The second phase of the compiler is *syntax analysis* or *parsing*. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation of the grammatical structure of the token stream. A typical representation is a *syntax tree* in which each interior node represents an operation and the children of the node represent the arguments of the operation.



This tree shows the order in which the operation in the assignment `position = initiali + rate * 60` are to be performed.

The following phases of the compiler use the grammatical structure to help analyze the source program and generate the target program.

1.2.3 Semantic Analysis

The *semantic analyzer* uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.
An importan part of the semantic analysis is *type checking*, where the compiler checks that each operator has matching operands.
The language specification may permit some type conversions called *coercions*. For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.

Chapter 2

Formal Grammar

A formal **grammar** is a set of formation rules for strings in a formal language. The rules describe how to form strings from the language's alphabet that are valid according to the language's syntax. A grammar does not describe the meaning of the strings or what can be done with them in whatever context, but only their form.
A formal grammar is a set of rules for rewriting strings, along with a "start symbol" from wich rewriting must start.

2.1 Introduction

A grammar mainly consists of a set of rules for transforming strings. To generate a string in the language, one begins with a string consisting of only a single *start symbol*. The *production rules* are then applied in any order, unitl a string that contains neither the start symbol nor designated *nonterminal symbols* is produced. A production rule is applied to a string by replacing one occurence of its left-hand side in the string by its right-hand side. The language formed by the grammar consists of all distinct string that can be generated in this manner.
Thus, a grammar is formally described as following

$$G = (V, T, S, P)$$

where:

- V represents the Vocabulary;
- T is a set of terminal symbols ($T \subset V$);
- S is the start symbol ($S \in V \setminus T$);
- P is the set of production.

A production rule is of the form $\alpha \to \beta$, where:

- $\alpha \in V^+$ (1 ore more elements);
- $\beta \in V^*$ (0 ore more elments).

NB: ε is a special symbol which represnts the empty string.

Example.
An example of grammar could be the following:

$$P = \{S \to aAb, aA \to aaAb, A \to \varepsilon\}$$

$$G = (\{S, A, a, b\}, \{a, b\}, S, P)$$

In this way, would be possible, for example, to generate the following word:

$$S \to aAb \to aaAbb \to aabb$$

A **string** is an empy or finite sequence of symbols (of a given alphabet); the *length* of a string is the number of symbols which occur in the string, the *concatenation* is a binary operation which merge two strings.

A string γ **directly derives** from a string μ in a given grammar $G = (V, T, S, P)$, if and only if $\mu = \sigma\alpha t$, $\alpha \to \beta \in P$ and $\gamma = \sigma\beta t$, where $\sigma, t \in V^*$.

A derivation of a string for a grammar is a sequence of grammar rule applications, that transforms the start symbol into the string. A derivation proves that the string belongs to the grammar's language.

A derivation is fully determined by giving, for each step:

- the rule applied in that step;
- the occurence of its left hand side to which it is applied.

A string γ **derives** from a string μ if and only if there exists a sequence of string $\omega_0, ..., \omega_n$, where $\mu = \omega_0$, $\gamma = \omega_n$, $\forall i : 0 \leq i \leq n-1, \omega_{i+1}$ directly derives from ω_i.

A **language** L generated by a grammar G is defined as following:
$L(G) = \{\omega | \omega \in T^*; \omega \text{ derives from } S\}$

Example.
Given the grammar $G = \{V, T, S, P\}$, where $P = \{S \to aAb, aA \to aaAb, A \to \varepsilon\}$, tha language generated by G is

$$L(G) = \{a^n b^n | n > 0\}$$

Exercise. *Given the grammar* $G = (\{S, A, B, a, b\}, \{a, b\}, S, P\}$, *where* $P = \{S \to AB, A \to aA, A \to a, B \to Bb, B \to b\}$, *where capital letters are non terminal, which is the language generated by G?*

I try to generate some strings:

$S \to AB \to aAB \to aaB \to aab$

$S \to AB \to ABb \to Abb \to abb$

Thus, I can understand that the language generated by this grammar is:

$$L(G) = \{a^n b^m | n > 0, m > 0\}$$

Exercise. *Given the Grammar* $G = (\{S, A, B, a, b\}, \{a, b\}, S, P)$, *where* P *is the set of the following production rules,* $S \to CD$, $C \to aCA|bCB$, $AD \to aD$, $BD \to bD$, $Aa \to aA$, $Ab \to bA$, $Ba \to aB$, $Bb \to bB$, $C \to \varepsilon$, $D \to \varepsilon$, *and where capital letters are non terminals, find which is tha language generated by this grammar.*

In this case, finding the generated language is not very easy; in fact, possible words belonging to this language are:

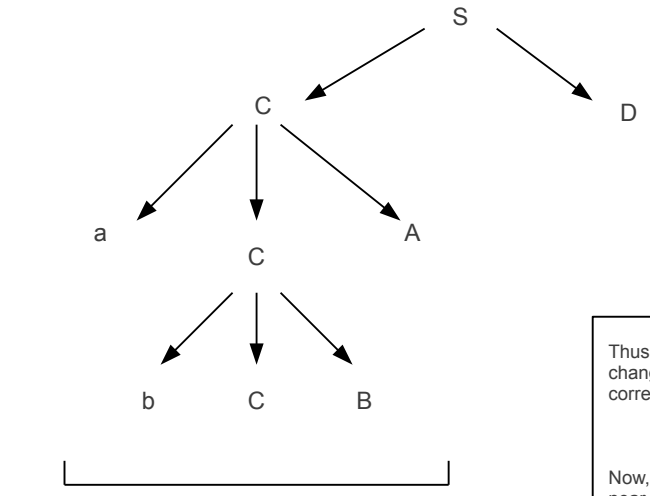
$S \to CD \to aCAD \to aCaD \to aa$

$S \to CD \to bCBD \to bCbD \to bb$

$S \to CD \to \varepsilon$

$S \to CD \to aCAD \to abCBAD \to abCBaD \to abCaBD \to abCabD \to abab$

In this case, coulde be useful to threat the language as a tree to have a better understanding:



2.2 Free grammars and free languages

In formal languages, a **context-free grammar** (**CFG**), is a formal grammar in whic every production rule is of the form

$$A \to \beta$$

where A is a single nonterminal symbol, and β is a string of terminals and/or nonterminals (β can be empty).

A grammar is **linear**, if it is context-free and all of its productions' right hand sides have at most one nonterminal; i.e, its production rules are of the shape:

- *Left linear*: nonterminal on the left ($A \to aB|a$)
- *Right linear*: nonterminal on the right ($A \to Ba|a$)

A language L is said to be a **context-free language** (**CFL**), if there exists a free grammar G such that

$$L(G) = L$$

NB: there are many grammars generating the same language.

Exercise. *Provide a grammar G such that* $L(G) = \{a^n b^n | n > 0\}$.

Possible solutions are:

- $S \to aCb$
  $C \to aCb|\varepsilon$
- $S \to aSb|ab$

Exercise. *Provide a grammar G such that* $L(G) = \{a^k b^n c^{2k} | n, k > 0$.

A possibile solution could be:

$$\begin{cases} S \to aScc|aAcc \\ A \to bA|b \end{cases}$$

NB: in general, there is no algorithm to understand which language is generated by a grammar.

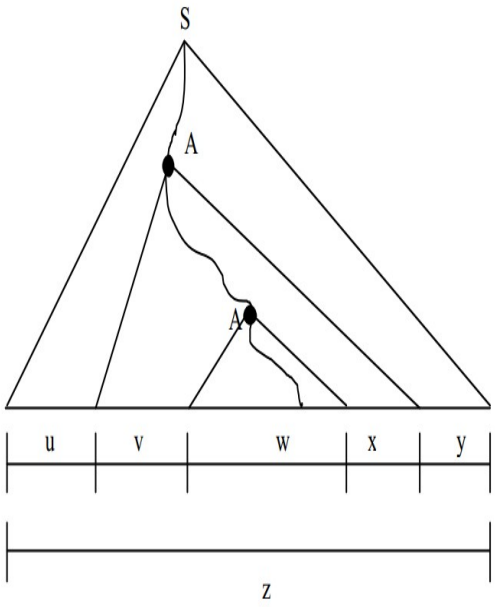
2.2.1 Pumping lemma for free languages

Pumping lemma for free languages. *Let L be a free language. The $\exists p \in N^*$ depending only on L such that $\forall z \in L.|z| > p$:*

- $z = uvwxy$
- $|vwx| \leq p$
- $vx \neq \varepsilon$
- $uv^iwx^iy \in L \forall i \in N$

NB: $vx \neq \varepsilon$ means that v and x can't be simultaneously empty.

Proof. We know that there exists a grammar G such that $L(G) = L$, where G has not empty (ε) productions.
Let n be the number of non-terminals of the grammar G. Let $p = 2^{n+1}$, and let z be a word of L with $|z| > p$. As $z \in L$, there exists a derivation-tree of z with p leaves, and, thus, with height of at least $log_2 p$ (In practice, we consider all the words that can be derived from the start symbol of G in at most $n + 1$ steps).
Then, consider the longest path C from the root to a leaf, having accordingly length of at least $n + 1 = log_2 p$. Now, consder a path of C composed by the last $n+1$ consecutive nodes, which precede the leaf; as G has n non-terminals, in the path there will be a non-terminal (called A) repeated in two different nodes.
We call w the longest word obtained form the leaves od the subtree which has as root the second ripetition of A; similarly, we call vwx the word obtained from the leaves of the subtree which has as root the firs ripetition of A. Thus, it results that $z = uvwxy$ reguarding to appropriate words u e y.



Other results are:

1. $|vx| \geq 1$. At least one word between x and v is different from ε; otherwise, the tow (distinct) nodes labeled with A would be coincident.

2. $|vwx| \leq p$. The height of the three which has as root the first ripetition of A is at most $n+1$, and thus it owns at most $p = 2^{n+1}$ leaves.

3. $\forall k \geq 0$, we have that $uv^k wx^k y \in L$. From the analysis of the derivation tree, we can understand that: $S \to G^* uAy, A \to G^* vAx, A \to G^* w$. So, $\forall k \geq 0$, we have that $S \to G^* uAY \to G^* uvAxy \to G^* uv^2 Ax^2 y \to$

$$G^* ... \rightarrow G^* uv^k Ax^k y \rightarrow G^* uv^k wx^k y.$$

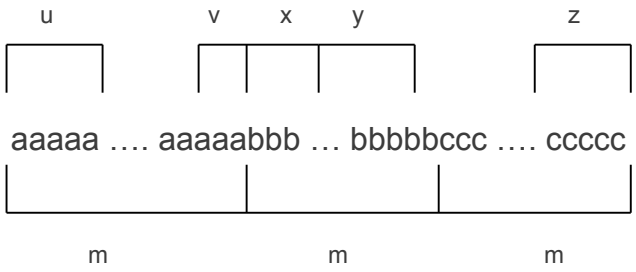
□

The pumping lemma for context-free languages describes a property that all context-free languages are guaranteed to have.
The property is a property of all strings in the language that are of length at least p, where p is a constant that varies between context-free languages. Say s is a string of length at least p that is in the language. The pumping lemma states that s can be split into five substrings, wher vY is non-empty and the length of vxy is at most p, such that repeating v and y any (and the same number) of times in s produces a string that is still in the language.

The pumping lemma is often used to prove that a given language is non-context-free by showing that for each p, we can find some string s of length at least p in the language that does not have the properties outlined aboved.

Exercise. *Show that* $L = \{a^n b^n c^n : n \geq 0\}$ *is not context-free.*

Assume that L is context-free. Then the Pumping Lemma applies. Let m be the "magic number"; consider the string $a^m b^m c^m$, which is clearly in L and is longer than m. By the pumping lemma:



- $a^m b^m c^m$ can be written as $uvxyz$ with $|vxy| \leq m$ and $|vy| > 0$ such that $uv^i xy^i z$ is in L for all $i = 0, 1, 2, 3...$
- There are only a limited number of possibilities for v and y.
  - v crosses the border between the a's and the b's (or y crosses the border between the b's and the c's).
  - v is all a's and y is all b's.
  - v is all b's and y is all c's.
  - ...
- Take each case separetely. Clearly if v or y "crosses a boundary", uv^2xy^2z will not be of the form a's followed by b's followed by c's.
- But if v is all a's then y can not contain c's. So pumping up will leave too few c's.

- If v is all b's then pumping up will leave too few a's.
- Therefore the language is not context-free.

2.2.2 General properties for context-free grammars

Exercise. *Given two grammars $G_i = (V_i, T_i, S_i, P_i)$, with $i = 1, 2$, there exists a grammar G such that $L(G) = L(G_1) \cup L(G_2)$?*

For example, we could have the following distinct productions belonging to two different grammars:

$$S_1 \to aS_1b|ab$$

$$S_2 \to cS_2d|cd$$

We should try to find a grammar which is $\{a^n b^n\} \cup \{c^n d^n\}$.
To do this, we have to choose an S new and write the following productions

$$S \to S_1|S_2$$

$$S_1 \to aS_1b|ab$$

$$S_2 \to cS_2d|cd$$

In this way, we have generated a language which is the union of the two languages generated by the two different grammars.

Lemma. *The class of context-free languages is closed wrt (with regard/respect to) union.*

Algorithm

`Algorithm`

- $G_i = (V_i, T_i, S_i, P_i)$ for $i = 1, 2$ - refresh (V_i, T_i) if $(V_1 \backslash T_1) \cap (V_2 \backslash T_2) \neq \emptyset$ - so that there are no clashes between the refreshed non terminals. - Choose fresh S, - then define $G = (V_1^{'} \cup V_2^{'} U \{S\}, T_1 \cup T_2, S, P_1^{'} \cup P_2^{'} U \{S \rightarrow S_1^{'} | S_2^{'}\})$

Lemma. *The class of context-free languages is closed wrt concatenation.*

Algorithm

- $G_i = (V_i, T_i, S_i, P_i)$ `for` $i = 1, 2$
- `refreshing phase, new` S
- $G = (V_1^{'} \cup V_2^{'} \cup S, T_1 \cup T_2, S, P_1^{'} \cup P_2^{'} \cup \{S- > S_1^{'} S_2^{'}\})$

Lemma. *The class of context-free languages is NOT closed wrt intersection.*

Exercise. *Show that the class of context-free languages in not closed wrt intersection.*

Choose L_1 and L_2 such that they are context-free languages and such that $L_1 \cap L_2$ is provably not context free language.
We know that $\{a^n b^n c^n\}$ is not context-free.
I can choose the following languages:

- $\{a^k b^n c^n\}$
  - $S \rightarrow AB$
  - $A \rightarrow aA|a$
  - $B \rightarrow bBc|bc$

- $\{a^n b^n c^k\}$
  - $S \to AC$
  - $A \to aAb|ab$
  - $C \to c|cC$

The intersection is:
$L_1 \cup L_2 = \{a^n b^n c^n, n > 0\}$

Negation of the pumping lemma

We know that the pumping lemma states that

Pumping lemma for free languages. *Let L be a free language. The $\exists p \in N^*$ depending only on L such that $\forall z \in L.|z| > p$:*

- $z = uvwxy$
- $|vwx| \leq p$
- $vx \neq \varepsilon$
- $uv^i wx^i y \in L \forall i \in N$

NB: **Reminder about negation**
$\overline{\exists p.P(p)} = \forall p.\neg P(p)$
$\overline{\forall p.P(p)} = \exists p.\neg P(p)$
$A \to B = \neg A \vee B = A \wedge \neg B$

So, we can negate the thesis of the pumping lemma in the follwing way:

$$\neg(\exists p.\forall z.(\underbrace{(z \in L \& |z| > p)}_{Q1} \to \underbrace{\exists uvwxy.(P1 \& P2 \& P3 \& P4)}_{Q2}))$$

where $P1, P2, P3$ and $P4$ are the properties of the pumping lemma.
Now, the following steps are:

$$\forall p.\neg(\forall z.Q1 \to Q2)$$

$$\forall p.\exists z.\neg(Q1 \rightarrow Q2)$$

$$\forall p.\exists z.(Q1\&\neg Q2)$$

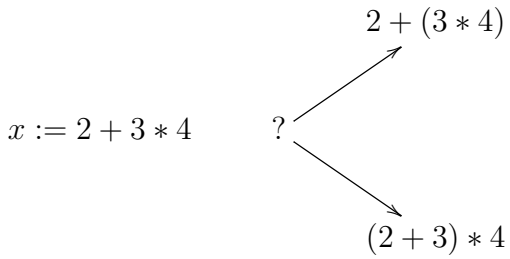
$$\forall p.\exists z.(Q1\&\forall uvwxyz.\neg((P1\&P2\&P3)\&P4)$$

$$\forall p.\exists z.(Q1\&\forall uvwxyz.\neg(P1\&P2\&P3) \wedge \neg P4)$$

$$\forall p.\exists z.(Q1\&\forall uvwxyz.(P1\&P2\&P3) \rightarrow \neg P4)$$

2.2.3 Ambiguous grammars

A parser checks if what we have written is related to a grammar.



Leftmost canonical derivation : if at each step of the derivation the leftmost non-terminal is the non-terminal transformed by the derivation.

Rightmost canonical derivation : if at each step of the derivation the rightmost non-terminal is the non-terminal transformed by the derivation.

Example.

Given the grammar

$$E \to E + E | E * E | id$$

the following sequence of productions is not canonical:

$$E \to E + E \to E + id \to E + E + id \to id + E + id$$

A grammar G is **ambiguous** if $\exists \omega \in L(G)$ that can be derived either by two distinct leftmost canonical derivations or by two distinct rightmost canonical derivations.

Exercise. *Given the following grammar*

$$E \to E + E | E * E | id$$

say if it is ambiguous.

$$\begin{aligned} \mathrm{E} &\to E * E \\ &\to E * E * E \\ &\to \underbrace{E + E * E * E}_{Leftmost} \end{aligned} \qquad \begin{aligned} \mathrm{E} &\to \mathrm{E} + \mathrm{E} \\ &\to \mathrm{id} + \mathrm{E} \\ &\to \mathrm{id} + \mathrm{E} * \mathrm{E} \\ &\to \underbrace{id + E * E * E}_{Rightmost} \end{aligned}$$

Thus, the grammar is ambiguous.

2.3 Exercises

NB: The pumping lemma can't absolutely never be used to demonstrate that a language is free.

- To demonstrate that a language is free, we have to provide a grammar G which is able to generate that language.
- To demonstrate that a language is not free, we have to use the pumping lemma by contradiction (Supposing that the language is free, and the contradicting it).

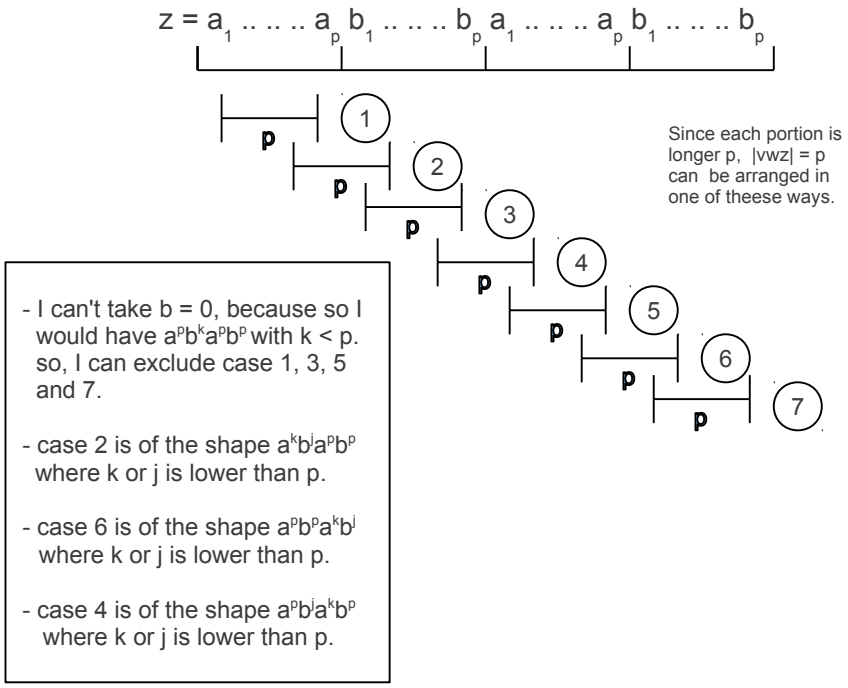
Exercise. *Say whether or not*

$$L = \{\omega\omega | \omega \text{ is a word on the alphabet } \{a, b\}\}$$

is a free language.

To demonstrate that it is not a free language, we must use the pumping lemma by contradiction.
Applying the pumping lemma, we know that:

- $\omega = a^p b^p$
- $z = a^p b^p a^p b^p$
- $z = uvwxy$
- $|vwz| \leq p$
- $|vx| \neq 0$



Thus, the language is not free.

Exercise. *Given the language*

$$L = \{a^i b^j c^k | i, j, k > 0 \ and \ (i = j \ or \ j = k)\},$$

define a grammar G such that $L(G) = L$; then, say whether is ambiguous or not.

A possible grammar is:

$$\begin{array}{ll} S \to AB|CD & \\ A \to a|aA & C \to aCb|ab \\ B \to bBc|bc & D \to c|cD \\ \hline a^i b^j c^j & a^i b^i c^j \end{array}$$

The grammar is ambiguous because there are two possible leftmost derivations which generate the same word:

$$S \to \underline{A}B \to a\underline{B} \to abc$$

$$S \to \underline{C}D \to ab\underline{D} \to abc$$

NB: beacuse of the condition ($i = j$ or $j = k$), every generated grammar would be ambiguous.

Exercise. *Say whether the language*

$$L = \{a^m b^n a^{m-n} | m, n > 0 \text{ and } m > n\}$$

is a context-free language.

The language L can be written as

$$L = \{a^{m-n} a^n b^n a m - n\}$$

and so

$$L = \{a^j a^n b^n a^j | n > 0, j > 0\}$$

Thus, a possible grammar is:

$$S \to aSa | aBa$$

$$B \to aBb | AB$$

Chapter 3

Finite Automata

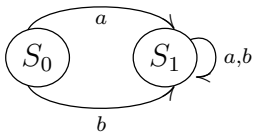
Finite automata are essentially graphs, like transition diagrams, with a few differences:

1. Finite automata are *recognizers*: they simply say "yes" or "no" about each possible input string.

2. Finite automata come in two flavors:

   (a) *Nondeterministic finite automata* (NFA) have no restrictions on the labels of their edges. A symbol can label several edges out of the same state, and ε, the empty string, is a possible label.

   (b) *Deterministic finite automata* (DFA) have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.



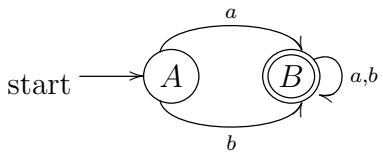
3.1 Deterministic Finite Automata

A deterministic finite automaton (**DFA**) is a finite state machine that accepts/rejects finite strings of symbols and only produces a unique computation (or run) of the automaton for each input string.
It is the simplest form of an automaton.

A DFA is a 5-tuple (S, a, δ, S_o, F) where:

- S is the set of states;
- a is the alphabet;
- δ is a *tranistion function* ($\delta : S \times a \to S$);
- S_0 is the initial state;
- F is the set of finite states.

For example, in the following DFA:



- A is the initial state;
- the set of finite states is the singleton B ($F = \{B\}$);
- $\delta : (A, a) \to B$ and $\delta : (A, b) \to B$;

In particular, the *state transition table* is:

	a	**b**
A	B	B
B	B	B

If DFAs recognize the languages that are obtained by applying an operation on the DFA recognizable lanaguages, then DFAs are said to be *closed under the operation*. The DFAa are closed under the following operations: Union, Intersection, Concetenation, Negation.

A run can be seen as a sequence of compositions of transition function with itself. Given an input symbol a belonging to the alphabet, one may write the transition function as $\delta : S \times a \to S$. This way, the transition function can be seen in simpler terms: it's just something that "acts" on a state in S, yielding another state.
One may then consider the result of function composition repeatedly applied to the various functions δ, and so on; thus, one may recursively define a new function $\hat{\delta}$ in the following way:

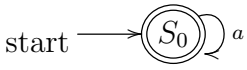
$$\hat{\delta}(s, \omega) = \begin{cases} s & \omega = \varepsilon \\ \delta(\hat{\delta}(s, \omega_1), a) & \omega = \omega_1 a \end{cases}$$

A DFA representing a regular language can be used either in an accepting mode to validate that an input string is part of the language, or in a generating mode to generate a list of all the strings in the language.
In the accept mode an input string is provided which the automaton can read in left to right, one symbol at time. The computation begins at the start state and proceeds by reading the first symbol from the input string and following the state transition corresponding to that symbo. The system continues reading symbols and following transitions until there are no more symbols in the input, which marks the end of computation.

A string ω is **accepted** (*recognized*) by the DFA $M = (S, a, \delta, S_0, F)$ if $\omega \in L(M)$ where $L(M) = \{\omega \in a^* : \hat{\delta}(S_0, \omega) \in F\}$.

Example.

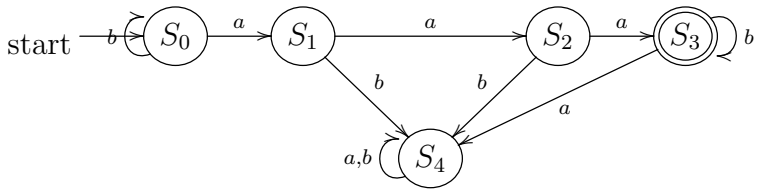
$a = \{\text{a}\}$



The language is $\{\varepsilon, a, aa, aaa, ...a^n\}$, thus $\{a^n | n \geq 0\}$.

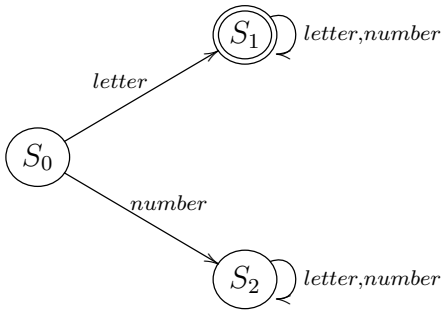
Exercise. *Suppose that our alphabet is $a = \{\ a,\ b\ \}$; "build" the DFA who consists of exactly 3 consecutive a.*

The alphabet is of the shape: $b^k aaab^j$. A possible DFA is the following:



NB: in this case, S_4 is a sort of "failure state".

Exercise. *Write an automaton for a language which starts with a letter followed by any number of letters or numbers.*



DFAs were introduced to model real world finite state machines in contrast to the conept of a Turing machine, which was too general to study properties of real world machines.

3.2 Non Deterministic Automata

A nondeterministic finite auomaton (**NFA**) consists of a 5-tuple $(S, a, \delta S_0, F)$, where:

1. S: a finite set of states;
2. a: a set of input symbols, the input alphabet; we assume that the empty string ε is never a member of a.
3. δ: a transition function that gives, for each state, and for each symbol in $a \cup \{\varepsilon\}$ a set of next states;
4. S_0: the start state or the (*initial state*),
5. F: a set of *final states* (accepting states).

There are two main differences with a DFA:

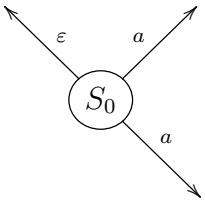
- The same symbol can label edges from one state to several different states;
- An edge may be labeled by ε, the empty string, instead of, or in addition to symbols from the input alphabet.

NB:

$$\delta S \times (a \cup \{\varepsilon\}) \to 2^S = P^S \ (\textit{set of the parts})$$

$$\delta(s, a) = \emptyset$$

The following is a simple example of NFA:



As for the DFA, we can also represent an NFA by a **transition table**, whose rows correspond to states, and whose columns correspond to the input symbol and ε. The entry for a given state and input is the value of the transition function applied to those arguments. If the transition function has no information about the state-input pair, we put $\emptyset$ in the table for the pair.

ε-closure

The ε-closure of a state s is the set of states that are reachabile from s by means of zero or more ε transitions.

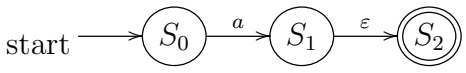
NB: $\forall s.s \in \varepsilon - closure(s)$

As for the DFA, also for the NFA we can define a $\hat{\delta}$ function as following:

$$\hat{\delta}(s,\omega) = \begin{cases} \varepsilon - closure(s) & \omega = \varepsilon \\ \bigcup_{s' \in \hat{\delta}(s,\omega_1)} \varepsilon - closure(\delta(s', a)) & \omega = \omega_1 a \end{cases}$$

Example.

Givent the following NFA:



We can calculate:

$$\delta(S_0, a) = \{S_1\}$$

$$\hat{\delta}(S_0, a) = \{S_1, S_2\}$$

An NFA *accepts* an input string x if and only if there is some path in the transition graph from the start state to one of the accepting states, such that the symbols along the path spell out x. Note that ε labels along the path are

effectively ignored, since the empty string does not contribute to the string constructed along the path.

A string ω is **accepted** (**recognized**) by the NFA $M = (S, a, \delta, s_0, F)$ if and only if $\hat{\delta}(s_0, \omega) \cap F \neq \emptyset$, $L(M) = \{\omega \in a^* : \hat{\delta}(s_0, \omega) \cap F \neq \emptyset$.

Chapter 4

Regular Expressions

In lexical analysis, the most important operations on languages are union, concatenation and closure.
Union is the familiar operation on sets.
The concatenation of languages is all strings formed by taking a string from the first language and a string from the second language, in all possible ways, and concatenating them.
The (*Kleene*) *closure* of a language L, denoted L^*, is the set of strings you get by concatenating L zero or more times.
Finally, the *positive closure*, denoted L^+, is the same as the Kleen closure, but wihtout the term L^0 (the concatenation of L zero times); that is, ε will not be in L^+ unless it is in L itself.

Operation	**Definition and Notation**
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Suppose we wanted to describe the set of valid C identifiers (a letter followed by letter, number or underscore); we are able to describte identifiers by giving names to sets of letters and digits using the language operators union, concatenation, closure.

This process is so useful that a notation called **regulare expressions** has come into common use for describing all the languages that can be built from these operators applied to the symbols of some alphabet.
In this notation, we could describe the language of C identifiers by:

$$letter\_\ (letter\_|\ digit)^*$$

The regulare expressions are built recursively out of smaller regular expressions, using the rules described below. Each regular expression r denotes a language $L(r)$, which is also defined recursively from the languages denoted by r's subexpressions.
Here are the rules that define the regular expressions over some alphabet a and the languages that those expressions denote.

Base : there are tow rules that form the basis:

1. $r = \varepsilon$ is a regular expression, and $L(r) = \{\varepsilon\}$, that is, tha language whose sole member is the empty string.
2. If $r =$ a, and a is a symbol in a, then $L(r) =$a, that is, the language with one string, of length one, with a in its one position.

Step : there are four parts to the induction whereby larger regular expressions are built from smaller ones. Suppose r and s are regulare expressions denoting languages $L(r)$ and $L(s)$, respectively:

1. $(r)|(s)$ is a regular expression denoting the language $L(r) \cup L(s)$;
2. $(r)(s)$ is a regular expression denoting the language $L(r)L(s)$;
3. $(r)^*$ is a regular expression denoting the language $(L(r))^*$;
4. (r) is a regular expression denoting $L(r)$. This rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.

Alternation	$L(r) \cup L(s)$
Concatenation	$L(rs) = \{\omega \mid \omega = \omega_1\omega_2$ and $\omega_1 \in L(r)$ and $\omega_2 \in L(s)\}$
Repetition	$(L(r))^* = \{\omega^k \mid k \geq 0$ and $\omega \in L(r)\}$

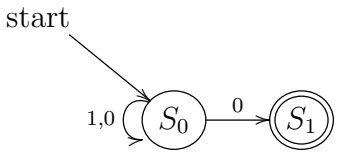
Exercise. *Write a regular expression which "generates"*

- *binary numbers ending with zero;*
- *strings over* $\{a, b\}$ *with no consecutive occurences of* a.

---

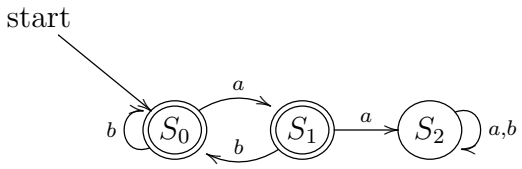
1. Binary numbers ending with zero

$$(0|1)^*0$$



2. Strings over $\{a, b\}$ with no consecutive occurences of a

$$(\varepsilon|a|b)(ba|b)^*$$



NB:
It is important to note that, speaking about regular expressions

$$b^+ = bb^*$$

4.1 Construction of an NFA from a Regular Expression

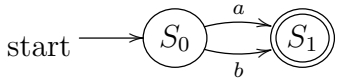
We now give an algorithm for converting any regular expression to an NFA that defines the same language.

Thompson's Algorithm: we will use the rules which defined a regular expression as a basis for the construction:

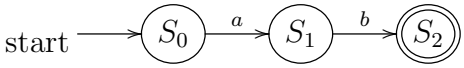
1. The NFA representing the empty string is:

2. If the regular expression is just a character, e.g. a, then the corresponding NFA is:

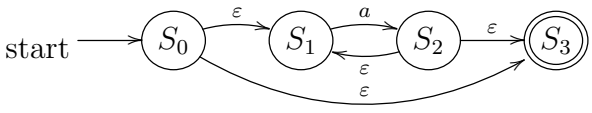
3. The union operator is represented by a choice of transitions from a node; thus, $a|b$ can be represented as:



4. Concetenation simply involves connecting one NFA to the other, e.g. ab is:



5. The Kleen closure must allow for taking zero or more instances of the letter from the input; thus a^* looks like:



The Mc-Naughton-Yamada-Thompson algorithm is used to convert a regular expression to an NFA.

- **Input**: a regular expression r over alphabet a;
- **Output**: an NFA N accepting $L(r)$.

Method: this algorithms begins by parsing r into its constituent subexpressions. The rules for contructing an NFA consist of the previous rules, which can be divided in:

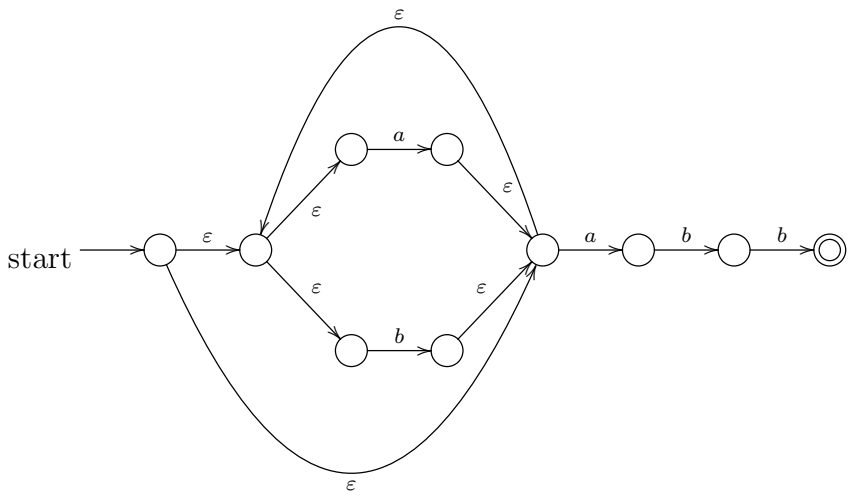
- *basis rules*: the first and the second one; they are used for handling subexpressions with no operators.
- *inductive rules*: used for constructing larger NFA's from the NFA's for the immediate subexpressions of a given expression (from the third to the fifth one).

It is useful to list several properties of the constructed NFA's, in addition to the all-important fact the $N(r)$ accepts language $L(r)$.

1. $N(r)$ has at most twice as many states as there are operators and operands in r. This bound follows from the fact that each step of the algorithm creates at most two new states.
2. $N(r)$ has one start state (without incoming transitions) and one accepting state (without outgoing transitions).
3. Each state of $N(r)$ other than the accepting state has either one outgoing transition on a symbol in the alphabet a or two outgoing transitions, both on ε.

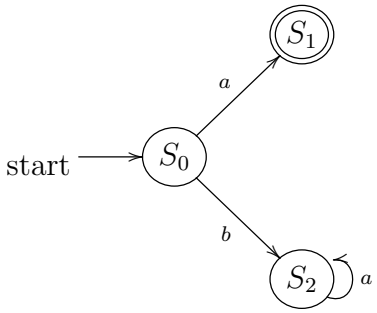
Example.

The NFA, built with the Thompson's algorithm for the regular expression $r = (a|b)^*abb$ is:

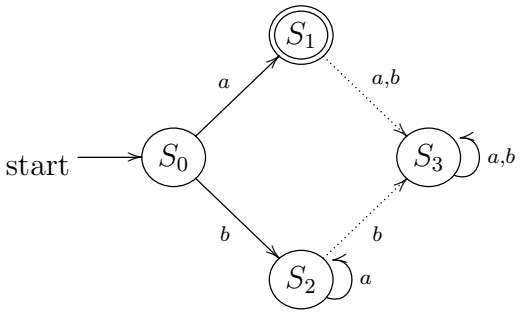


NB:

The following automaton

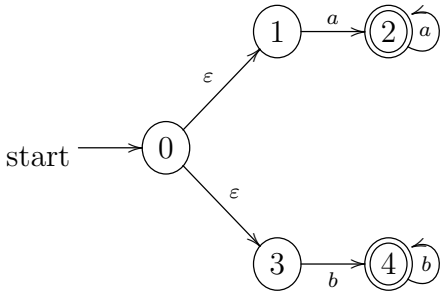


it is not a DFA, because the delta function (δ) has to be total. If I create a new state where to drive the "missing" edges, I can have a DFA as the following:



4.2 Conversion of an NFA to a DFA

Suppose that we have the following initial automaton:



We can convert an NFA to a DFA using **subset construction**. To perform this operation, let us define the ε-closure of a set of states.

The ε-**closure** funtction takes a state and returns the set of states reachable from it based on (one or more) ε transitions. Note that this will always include the state itself. We should be able to get from a state to any state in its ε-closure without consuming any input.

Algorithm for the definition of the $\varepsilon-$closure(T) with T set of states

---

```
1: ε−closure(T) = T              ▷ initially, it contains all the states of the set
2: push all the states in T on the stack
3: while stack not empty do
4:     pop T                                  ▷ take out the top of the stack
5:     for each u such that t →^ε u do
6:         if u ∉ ε − closure(T) then
7:             add u to ε−closure(T)
8:             push u
9:         end if
10:    end for
11: end while
```

The general idea behind the subset construction is that each state of the constructed DFA corresponds to a set of NFA states. After reading input $a_1, a_2, ..., a_n$, the DFA is in that state which corresponds to the set of states that the NFA can reach, from its start state, following paths labeled $a_1, a_2, ..., a_n$.

Our algorithm constructs a transition table *Dtran* for the DFA D. Each states of D is a set of NFA states, and we construct *Dtran* so D will simulate "in parallel" all possible moves N can make on a given input string.

- **base**: before reading the first input symbol, N can be in any of the states of $\varepsilon-$closure(s_0), where s_0 is its start state.

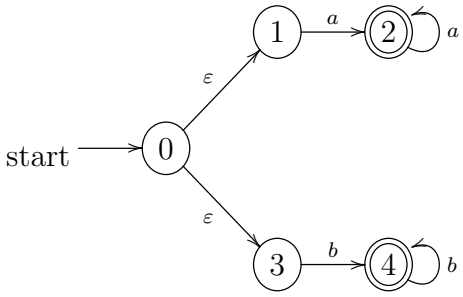
- **induction**: suppose that N can be in set of states T after reading input string x. If it next reads input a, the N can immediately go to any of the states reachable from a state in T with a transition a. However, after reading a, it may also make severale $\varepsilon-$ transitions; following this idea, the construction of the set of D's states, *Dstates*, and its transition function *Dtran*, is quite simple. The start state of D is $\varepsilon-$closure(s_0), and the final states of D are alla those sets of N's states that include at least one final state of N.

Algorithm NFA $\rightarrow$ DFA

1: initially $\varepsilon-$closure(s_0) is the only state in *Dstates*, and it is unmarked;
2: $\triangleright$ *Dstates* = $\{\varepsilon-\text{closure}(Ns_0)\}$
3: **while** *Dstates* contains at least one set T not yet marked **do**
4: **for each** input symbol a $\in a$ **do**
5: $U = \varepsilon-$closure(δ(T,a));
6: **if** $U \notin$ *Dstates* **then**
7: add U to *Dstates*;
8: set U not marked;
9: *Dstates*(T, a) := U;
10: **end if**
11: **end for**
12: **end while**

Example.

Suppose we have the following automaton:



and we want construct the corresponding DFA.

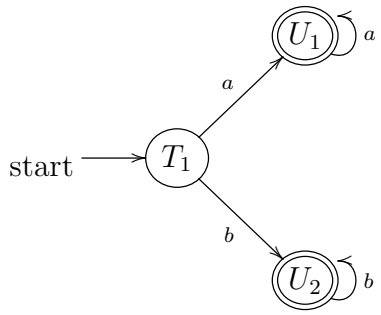
$\varepsilon-$closure$(0) = \{0, 1, 3\}$. We can call this set T_1.
$T_1 = \{0, 1, 3\}$.
T_1 is not marked, so we have to mark it and consider which set we reach with the possible inputs.

- $T_1 \rightarrow^a \{2\}$
- $T_1 \rightarrow^b \{4\}$

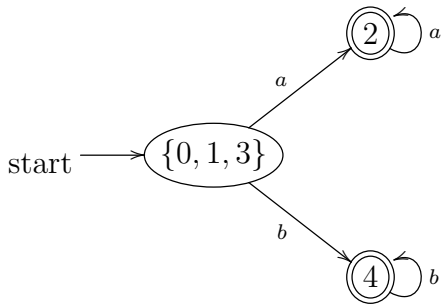
We have to calculate the $\varepsilon-$closure of the new sets of states reached:
$\varepsilon-$closure$(2) = \{2\}$ and $\varepsilon-$closure$(4) = \{4\}$.
So, with an a transition we reach the set $\{2\} \cup \varepsilon - closure(2) = \{2\}$ and with a b transition we reach the set $\{4\} \cup \varepsilon - closure(4) = \{4\}$.
So, we have the sets:
$U_1 = \{2\}$
$U_2 = \{4\}$
Now, we have to consider first U_1 and then U_2, mark them and check which set of states we reach with the possible input strings, obtaining the following table:

	a	**b**
T_1	U_1	U_2
$\underline{U_1}$	U_1	$\emptyset$
$\underline{U_2}$	$\emptyset$	U_2

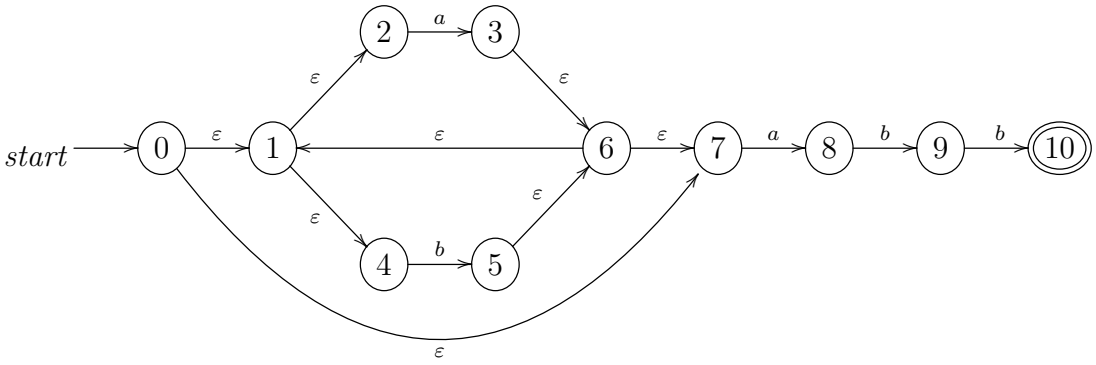
NB: the underlined set of states, are the final states of the DFA because they contain at least one final state of the NFA. So, the constructed DFA is:



which can be finally written as:



Exercise. *Convert the following NFA into a DFA:*



$\varepsilon-$closure$(\{0\}) = \{1, 2, 4, 7\} = T_1$.

We mark T_1 and check which sets of states we can reach:

$T_1 \to^a = \{3, 8\} \to \varepsilon-$closure$(\{3\}) + \varepsilon-$closure$(\{8\}) \to \{3, 6, 7\} + \{8\} + T_1 = T_2$.

$T_1 \to^b = \{5\} \to \varepsilon-$closure$(\{5\}) \to \{5, 6, 7\} + T_1 = T_3$.

So, we have:

$T_1 = \{0, 1, 2, 4, 7\}$

$T_2 = \{1, 2, 3, 4, 6, 7, 8\}$

$T_3 = \{1, 2, 4, 5, 6, 7\}$

. Repeating what we have done for T_1 with T_2 and T_3, we obtain the following new sets:

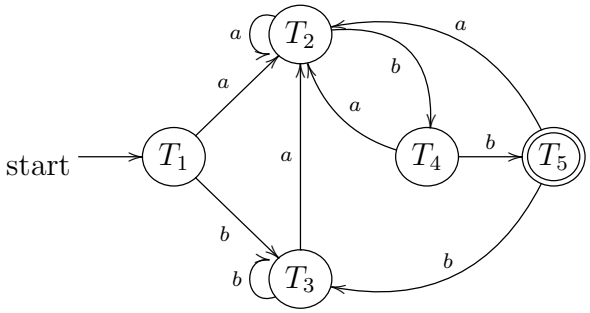
$T_4 = \{1, 2, 4, 5, 6, 7, 9\}$

$T_5 = \{1, 2, 4, 5, 6, 7, 10\}$.

and the following table:

	a	**b**
T_1	T_2	T_3
T_2	T_2	T_4
T_3	T_2	T_3
T_4	T_2	T_5
$\underline{T_5}$	T_2	T_3

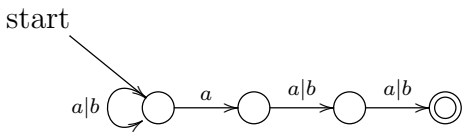
So, the DFA is:



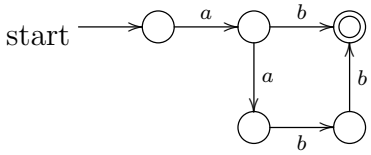
Exercise. *Write an automaton for the following regular expressions:*

- $(a|b)^*a(a|b)(a|b)$
- $(ab)|(aabb)$

The automaton for the regular expression $(a|b)^*a(a|b)(a|b)$ is:



The automaton for the regular expression $(ab)|(aabb)$ is:



4.3 Minimization of DFAs

There can be many DFA's that recognize the same language. Not only do these automata have states with different names, but they don't even have the same number of states.
It turns out that there is always a unique minimum state DFA for any regular language. Moreover, this minimu-state DFA can be constructed from any DFA for the same language by grouping sets of equivalent states.
Before seeing the algorithm, it is important to give some definitions:

Group : a set of states;

Partition : set of groups;

Refinable partition :

given a DFA, a partition P of its states is **refinable** if
$\exists G, G_1, G_2 \in P : G_1 \neq G_2, \exists s_1, s_2 \in G, \exists$ a $\in a$ such that
$\delta(s_1,$ a$) = s_i \wedge s_i \in G_1$
$\delta(s_2,$a$) = s_j \wedge s_j \in G_2$.

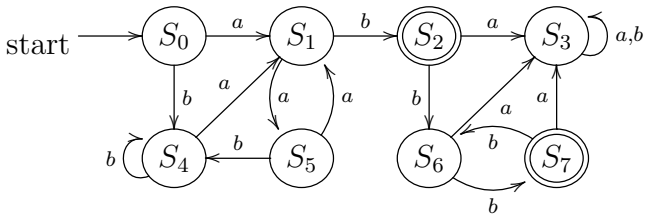
Refinement : given a refinable partition P, its **refinement** consists in subdividing the groups of P till P is modified into a partition called $ref(P)$ which cannot be refined further.

Refinement algorithm

1: a partition P is of the shape $P := (F, S \setminus F)$, where F is the set of final states
2: **if** P is refinable **then**
3: compute ref(P);
4: return ref(P);
5: **else**
6: P;
7: **end if**
8: $\forall$ group in P, find a representative of the group.

Example.

Suppose we have the following automaton:



and we want to apply the partitioning algorithm.

First, divide the set of states into final and non-final states.

	a	**b**
S_0	S_1	S_4
S_1	S_5	S_2
S_3	S_3	S_3
S_4	S_1	S_4
S_5	S_1	S_4
S_6	S_3	S_7
$\underline{S_2}$	S_3	S_6
$\underline{S_7}$	S_3	S_6

	a	**b**
S_0	S_1	S_4
S_3	S_3	S_3
S_4	S_1	S_4
S_5	S_1	S_4
S_1	S_5	S_2
S_6	S_3	S_7
$\underline{S_2}$	S_3	S_6
$\underline{S_7}$	S_3	S_6

Now, we have to see if states in each partition go to the same partition.
S_1 and S_6 are different from the rest of the states in the first partition (the black one), because with a b transition they go in the blue partition, while the others remain in the black partition.
So, we will move them to their own partition.

Now, again, we have to see if states in each partition go to the same partition.
We can note that, in the black partition, with an a transition, S_3 goes to a different partition from S_0, S_4 and S_5.
So, we have to move S_3 to its own partition.

	a	**b**
S_0	S_1	S_4
S_4	S_1	S_4
S_5	S_1	S_4
S_3	S_3	S_3
S_1	S_5	S_2
S_6	S_3	S_7
$\underline{S_2}$	S_3	S_6
$\underline{S_7}$	S_3	S_6

	a	**b**
S_0	S_1	S_4
S_4	S_1	S_4
S_5	S_1	S_4

S_3	S_3	S_3

S_1	S_5	S_2

S_6	S_3	S_7

S_2	S_3	S_6
S_7	S_3	S_6

Now, note that S_6 goes to a different partion on an a from S_1. S_6 gets its own partition. We now have 5 partitions, where all states are identical. So, we have finished and we can simply call the partitions A, B, C, D and E.

$A = \{S_0, S_4, S_5\}$

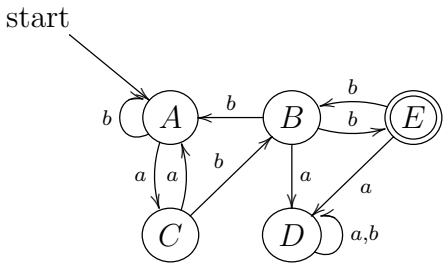
$B = \{S_3\}$

$C = \{S_1\}$

$D = \{S_6\}$

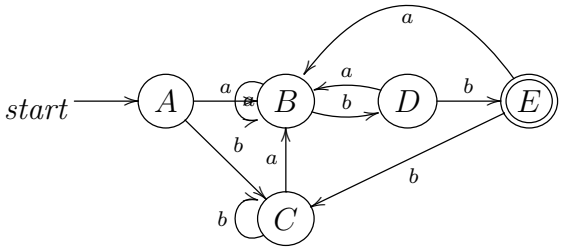
$E = \{S_2, S_7\}$

Thus, the automaton becomes:



4.4 Exercises

Exercise. *Using the refinement algorithm, simiplify the following automaton generated by the language* $(a|b)^*abb$.



First of all, the partition we have is:

$$P = (\{A, B, C, D\}, \{E\})$$

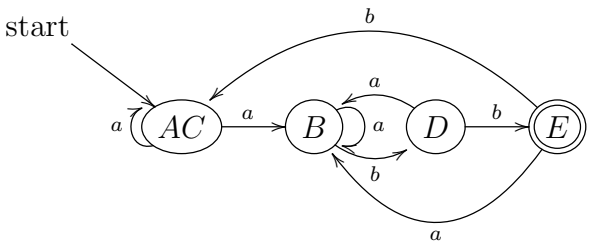
We can see that, with a b transition, D has a different behaviour from the states in its partition; so we move it to an own partition:

$$P^{'} = (\{A, B, C\}, \{D\}, \{E\})$$

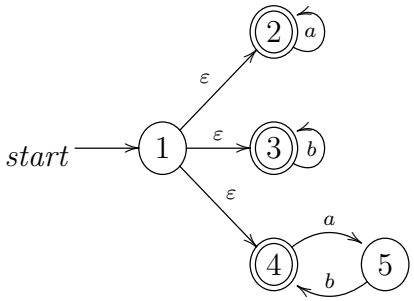
We can note that B goes to a different partition from A and C with a b transition; so we move it to its own partition:

$$P^{''} = (\{A, C\}, \{B\}, \{D\}, \{E\})$$

Now, every state of a partition behaves in the same way; thus, we can rewrite the automaton as following:



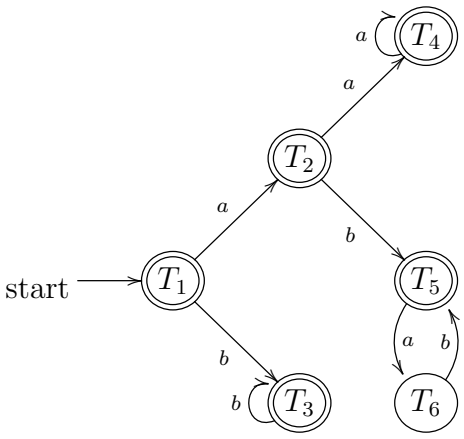
Exercise. *Convert the following NFA into a DFA*



The "transition table" is:

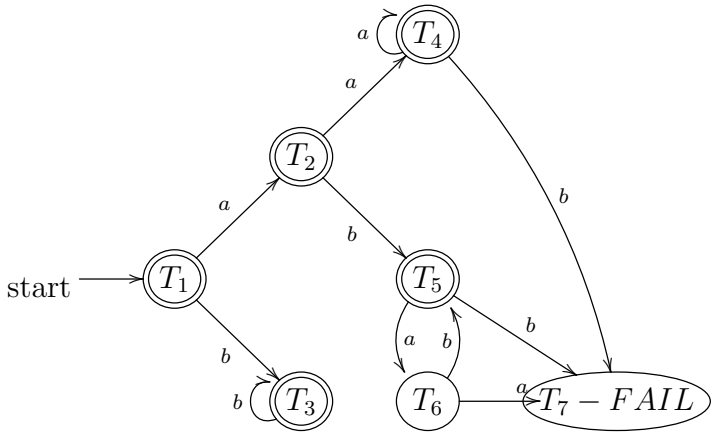
	a	**b**
$\underline{T_1} = \{1, 2, 3, 4\}$	T_2	T_3
$\underline{T_2} = \{2, 5\}$	T_4	T_5
$\underline{T_3} = \{3\}$	$\emptyset$	T_3
$\underline{T_4} = \{2\}$	T_4	$\emptyset$
$\underline{T_5} = \{4\}$	T_6	$\emptyset$
$T_6 = \{5\}$	$\emptyset$	T_5

So, the resulting automaton is:

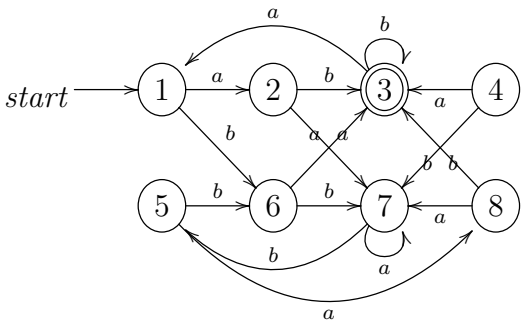


To be a DFA, the δ function has to be total; thus, I have to add a sort of

"fail-state":



Exercise. *Use tha refinement algorithm to simplify the following DFA:*



First partition:

$$P^i = (\{1, 2, 4, 5, 6, 7, 8\}, \{3\})$$

We can note that 4 and 6 behaves differently, with an a transition.

$$P^{ii} = (\{1, 2, 5, 7, 8\}, \{4, 6\}, \{3\})$$

With a b transition, 1 and 5 go to 6, 2 and 8 go to 3 and 7 goes to 5; thus,

$$P^{iii} = (\{1, 5\}, \{2, 8\}, \{7\}, \{4, 6\}, \{3\})$$

Now, we can see that the elements of each partition behave in the same way:

	a	**b**
$A = \{1, 5\}$	B	D
$B = \{2, 8\}$	C	E
$C = \{7\}$	C	A
$D = \{4, 6\}$	E	C
$E = \{3\}$	A	E

4.5 Pumping lemma for regular languages

A **regular language** is a formal language that can be expressed using a regular expression.
In the *Chomsky hierarchy* (a containment hierarchy of classes of formal grammars) regular languages are defined to be the languages that are generated by regular grammars.

Pumping lemma for regular languages. *Let L be a regular language. Then $\exists p \in N^+$ such that $\forall z \in L. |z| > p$ $\exists u, v, w$ such that*

- $z = uvw$
- $|uv| \leq p$
- $|v| > 0$
- $\forall i \geq 0, uv^i w \in L$

Proof. By L regular, L can be recognized by a finite state automaton. Let M be the minimum DFA such that $L(M) = L$; take $p =$ (number of states of M) $- 1$.
Note that p is the minimal length of words that can be accepted by M taking walks that pass through any of its states at most once. If we take a word $z \in L$ and such that $|z| > p$, then z is accepted by means of a walk that traverses a node at least twice.
We can call S this node. The transitions that take the machine from the first encounter of state S to the second encounter of state S match some string. This string is called v in the lemma, and since the machine will match a string without the v portion, or the string v can be repeated any number of times, the conditions of the lemma are satisfied. □

This lemma is often used by contradiction to prove that a particular language is non-regular.

Proposition. $\{a^n b^n, n > 0\}$ *is not regular.*

Proof. Suppose $\{a^n b^n\}$ is regular. Then let p be the constant of the pumping lemma. Choose $z = a^p b^p$, $|z| > p$; then, by the pumping lemma, there must be some decomposition $z = uvw$ with $|uv| \leq p$ and $|v| > 0$ such that $uv^i w \in L$ for every $i \geq 0$. Using $|uv| \leq p$, we know v only consists of instances of a. Moreover, because $|v| \geq 1$, it contains at least one instance of the letter a. We now pump v up: uv^2w has more instances of the letter a than the letter b, since we have added some instances of a without adding instances of b. Therefore uv^2w is not in L. We have reached a contradiction. Therefore, the assumption that L is regular must be incorrect. Hence L is not regular. □

Chapter 5

Parsing

Parsing is the process of determining how a string of terminals can be generated by a grammar.
Mostr parsing methods fall into one of two classes:

- top-down parsing
- bottom-up parsing

These terms refer to the order in which nodes in the parse tree are constructed. In top-down parsers, construction starts at the root and proceeds towards the leaves, while in bottom-up parsers, construction starts at the leaves and proceed towards the root.
The popularity of top-down parsers is due to the fact that efficient parsers can be constructed more easily by hand using top-down methods. Bottom-up parsing, however, can handle a larger class of grammars and translation schemes.

5.1 Top-Down Parsing

The top-down construction of a parse tree is done by starting with the root, labeled with the starting nonterminal, and repeatedly performing the following two steps:

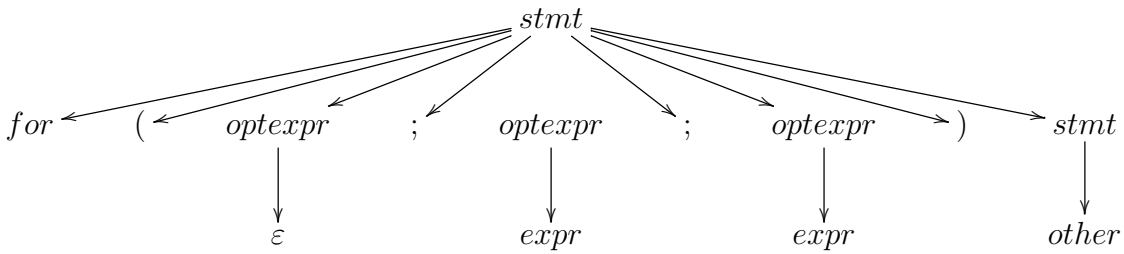
1. At node N, labeled with nonterminal A, select one of the productions for A and construct children at N for the symbols in the production body;

2. Find the next node at which a subtree is to be constructed, typically the leftmost unexpanded nonterminal of the tree.

For example, given the grammar

stmt	$\rightarrow$	expr;
	|	if (expr) stmt
	|	for (optexpr ; optexpr ; optexpr) stmt
	|	other
optexpr	$\rightarrow$	ε
	|	expr

A parse tree according to the grammar could be the following:



At each step of a top-down parse, the key problem is that of determining the production to be appplied for a nonterminal, say A. Once an A-production

is chosen, the rest of the parsing process consists of "matching" the terminal symbols in the production body with the input string.

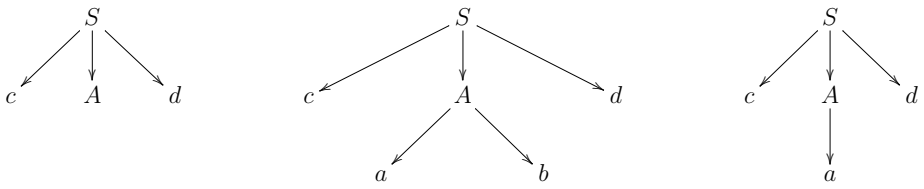
A **recursive-descendent parsing** program consists of a set of procedures, one for each nonterminal. Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string.
General recursive-descendent may require backtracking; that is, it may require repeated scans over the input.

Example.
Consider the grammar

$$S \to cAd$$

$$A \to ab|a$$

To construct a parse tree top-down for the input string $w = cad$, begin with a tree consisting of a single node labeled S, and the input pointer pointing to c, the first symbol of w. S has only one production, so we use it to expand S and obtain the tree (a) of the figure. The leftmost leaf, labeled c, mathces the first symbol of input w, so we advance the input pointer to a, the second symbol of w, and consider the next leaf, labeled A.



(a) (b) (c)

Now, we expand A using the first alternative $A \to ab$ to obtain the tree (b) of the figure. We have a match for the second input symbol, a, so we advance the input pointer to d, the third input symbol, and compare d against the next leaf, labeled b. Since b does not match d, we report failure and go back

to A to see whether there is another alternative for A that has not been tried, but that might produce a match.
The second alternative for A produces the tree (c) of the figure. The leaf a matches the second symbol of w and the leaf d matches the third symbol. Since we have produced a parse tree for w, we halt and announce successful completion of parsing.

We speak about **predictive top-down parsing** when, starting from the start symbol S, having some left-most derivations, we produce the word we want to recognize.

5.1.1 Left Recursion

It is possible for a recursive-descendent parser to loop forever. A problem arises with **left recursive** productions like

$$expr \rightarrow expr + term$$

where the leftmost symbol of the body is the same as the nonterminal at the head of the production. In other words, there is a **left recursion** when, starting from a symbol S, in some steps we reach a string which starts with the same symbol.

A left-recursive production can be eliminated by rewriting the offending production.

We speak about **immediate left recursion** when we have productions of the following shape:

$$A \rightarrow A\alpha|\beta$$

where α and β are sequences of terminals and nonterminals that do not start with A.
The nonterminal A and its production are said to be *left recursive*, because the production $A \rightarrow A\alpha$ has A itself as the leftmostr symbol on the right

side. Repeated application of this production builds up a sequence of *alpha*′s to the right of A; when A is finally replaced by β, we have a β followed by a sequence of zero or more α′s.

$$A \to A\alpha \to A\alpha\alpha \to ... \to \beta\alpha^k$$

The same result can be achieved by replacing

$$A \to A\alpha|\beta$$

by

$$A \to \beta A^{'}$$

$$A^{'} \to \alpha A^{'}|\varepsilon$$

where $A^{'}$ is a new symbol. Nonterminal $A^{'}$ and its production $A^{'} \to \alpha A^{'}$ are *right recursive* because this production for R has R itself as the last symbol on the right side.

NB: Left recursion is not necessarly immediate.

Using the grammar

$$S \to Aa|b$$

$$A \to Ac|Sd|b$$

we can produce the following production:

$$S \to Aa \to Sda$$

Elimination of Left Recursion

We have seen that a grammar is *left recursive* if it has a nonterminal A such that there is a derivation $A \to^+ A\alpha$ for some string a. Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion.

Immediate left recursion can be eliminated by the following tecnique, which works for any number of $A-$productions. First, group the productions as

$$A \to A\alpha_1|A\alpha_2|...|A\alpha_m|\beta_1|\beta_2|...|\beta_n$$

where no β_i begins with an A. Then, replace the $A-$productions by

$$A \to \beta_1 A^{'}|\beta_2 A^{'}|...|\beta_n A^{'}$$

$$A^{'} \to \alpha_1 A^{'}|\alpha_2 A^{'}|...|\alpha_m A^{'}|\varepsilon$$

The nonterminal A generates the same string as before but is no longer left recursive. This procedure eliminates all left recursion from the A and $A^{'}$ productions (provided no α_1 is ε''), but it does not eliminate left recursion involving derivation of two or more steps.

For example, consider the grammar

$$S \to Aa|b$$

$$A \to Ac|Sd|\varepsilon$$

The nonterminal S is left recursive because $S \to Aa \to Sda$, but it is **not immediately left recursive**.

The algorithm below systematically eliminates left recursion from a grammars It is guaranteed to work if the grammar has no cycles (derivation of the form $A \to^{+} A$) or $\varepsilon-$ productions ($A \to \varepsilon$). Cycles can be eliminated systematically from a grammar, as can $\varepsilon-$productions.

Elimination of left recursion

1: choose some ordering of the nonterminals (say $A_1, ...A_n$);
2: **for** i=1 to n **do**
3: **for** j=1 to i-1 **do**
4: substitute $A_i \to A_j$ by $A_i \to \delta_1\gamma|...|\delta_k\gamma$
5: where A_j is such that $A_j \to \delta_1|...|\delta_k$;
6: **end for**
7: eliminate immediate left recursion by A_i;
8: **end for**

Example.

If we have the following grammar

$$A \to Ba|b$$

$$B \to Bc|Ad|b$$

and we want to eliminate the left recursion we have to:

1. rewrite:
$$A_1 \to A_2a|b$$
$$A_2 \to A_2c|A_1d|b$$

2. $i = 1$: no immediate left recursion in A_1;

3. $i = 2; j = 1$: replace $A_2 \to A_1d$ with $A_2 \to A_2ad|bd$; in practice we have to replace A_1 with the possible possible productions of the rule $A_1 \to A_2a|b$.
So, the grammar is:
$$A_1 \to A_2a|b$$
$$A_2 \to A_2c|A_2ad|bd|b$$

4. eliminate left recursion for A_2:

$$A_2 \rightarrow bdA_2^{'}|bA_2^{'}$$

$$A_2^{'} \rightarrow cA_2{'}|adA_2{'}|\varepsilon$$

NB: **Ambiguity**
No ambiguous grammar can be parsed. There's no algorithm to disambiguate a grammar.
Worse, some languages er inherently ambiguous (languages of arbitrarly mixed if-then and if-then-else constructs.
Disambiguation of inherently ambiguous languages goes through the isolation of appropriately expressive sub-languages.
By definition, ambiguous grammars can generate more trees.

5.1.2 Left Factorization

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing. When the choiche between two alternative $A-$productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.

In general, if

$$A \rightarrow \alpha\beta_1|\beta_2$$

are two $A-$productions, and the input begins with a nonempty string derived from α, we do not know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$. However, we may defer the decision by expanding A to $aA^{'}$. Then, after seeing the input

derived from α, we expand A' to β_1 or to β_2. That is, left-factored, the original productions become

$$A \to \alpha A'$$

$$A' \to \beta_1 | \beta_2$$

Algorithm for left refactoring a grammar : For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \varepsilon$, i.e, there is a nontrivial common prefix, replace all of the $A-$productions $A \to \alpha\beta_1 | \alpha\beta_2 | ... | \alpha\beta_n | \gamma$, where γ represents all the alternatives that do not begin with α, by

$$A \to \alpha A' | \gamma$$

$$A' \to \beta_1 | \beta_2 | ... | \beta_n$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

Example.

Given the grammar

$$S \to \text{if } b \text{ then } S \text{ else } S \mid \text{if } b \text{ then } S \mid a$$

the left factorization is

$$S \to \text{if } b \text{ then } SS' \mid a$$
$$S' \to \text{else } S \mid \varepsilon$$

5.1.3 Nonrecursive Predictive Parsing

A nonrecursive predictive parser can be built by maintaining a stack explicitly, rather than implicity via recursive calls.

There is:

- an input buffer, which contains the string to be parsed, followed by the endmarker \$.
- a stack, containing a sequence of grammar symbols; we reuse the symbol \$ to mark the bottom of the stack, which initially contains the start symbol of the grammar on top of \$.
- a parsing table.
- an output stream.

The parser is controlled by a program that considers X, the symbol on top of the stack, and a, the current input symbol. If X is a nonterminal, the parses chooses an $X-$production by consulting entry $M[X, a]$ of the parsing table M (which can contain a production $A \to \alpha$ or an error message). Otherwise, it checks for a match between the terminal X and current input symbol a.

Non-recursive Predictive Parsing Algorithm

1: set ip to point to the first symbol of w;
2: set X to the top stack symbol
3: **while** $X \neq \$$ **do** ▷ stack is not empty
4: **if** X is a **then** pop the stack and advance ip;
5: **else if** X is a terminal **then** error();
6: **else if** $M[X, a]$ is an error entry **then** error();
7: **else if** $M[X, a] = X \to Y_1 Y_2 ... Y_k$ **then**
8: output the production $X \to Y_1 Y_2 ... Y_k$;
9: pop the stack;
10: push $Y_k, Y_{k-1}, ..., Y_1$ onto the stack, with Y_1 on top;
11: **end if**
12: set X to the top stack symbol;
13: **end while**

Example.

Consider the grammar

$$E \to TE^{'}$$

$$E^{'} \to +TE^{'}|\varepsilon$$

$$T \to FT^{'}$$

$$T^{'} \to *FT^{'}|\varepsilon$$

$$F \to (E)|id$$

and its parsing table

Non-terminal	**Input Symbol**					
	id	+	*	(	)	$
E	$E \to TE^{'}$			$E \to TE^{'}$		
$E^{'}$		$E^{'} \to +TE^{'}$			$E^{'} \to \varepsilon$	$E^{'} \to \varepsilon$
T	$T \to FT^{'}$			$T \to FT^{'}$		
$T^{'}$		$T^{'} \to \varepsilon$	$T^{'} \to *FT^{'}$		$T^{'} \to \varepsilon$	$T^{'} \to \varepsilon$
F	$F \to id$			$F \to (E)$		

The moves made by a predictive parser on input **id + id * id** are:

Matched	**Stack**	**Input**	**Action**
	$E\$$	id + id * id$	
	$TE'\$$	id + id * id$	output $E \to TE'$
	$FT'E'\$$	id + id * id$	output $T \to FT'$
	id $T'E'\$$	id + id * id$	output $F \to$ id
id	$T'E'\$$	+ id * id$	match id
id	$E'\$$	+ id * id$	output $T' \to \varepsilon$
id	$+TE'\$$	+ id * id$	output $E' \to +TE'$
id +	$TE'\$$	id * id$	match +
id +	$FT'E'\$$	id * id$	output $T \to FT'$
id +	id $T'E'\$$	id * id$	output $F \to$ id
id + id	$T'E'\$$	* id$	match id
id + id	$*FT'E'\$$	* id$	output $T' \to *FT'$
id + id *	$FT'E'\$$	id$	match *
id + id *	id $T'E'\$$	id$	output $F \to$ id
id + id * id	$T'E'\$$	$	match id
id + id * id	$E'\$$	$	output $T' \to \varepsilon$
id + id * id	$	$	output $E' \to \varepsilon$

Exercises

Exercise. *Say whether or not*

$$\{a^i a^i b^j | i, j \geq 0 \wedge \textit{ if } i, j > 0 \textit{ then } i \neq j\}$$

is a context-free language.

The language can be seen as $\{a^{2i}b^j | i, j \geq 0 \wedge \text{ if } i, j > 0 \text{ then } i \neq j\}$; are possible three different options, and so L can be seen as the union of 3 languages.

$$L = L_1 \cup L_2 \cup L_3$$

where

- $L_1 = \{a^2 i b^j | i = j = 0\} = \{\varepsilon\}$
- $L_2 = \{a^2 i b^j | i > 0 \wedge j > i\}$
- $L_3 = \{a^2 i b^j | j \geq 0 \wedge i > j\}$

Let say that B is the starting symbol of $L_2 : B \to aaBb|Bb|b$.
Let say that A is the starting symbol of $L_3 : A \to aa|aaAb|aaA$.
As we have built 3 context-free language, it is possible to define a context-free language L as the union of the three languages

$$L : S \to A|B|\varepsilon$$

Exercise. *Design the minimum DFA that recognizes all the words over the alphabet* $a = \{ a, b, c \}$ *where* b *occurs either zero or an odd number of times.*

The grammar can be written as

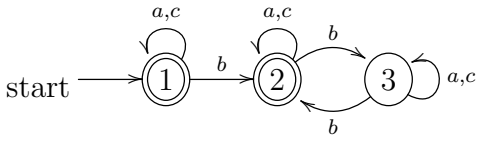
$$S \to ABC|AC$$

$$A \to aA|a|\varepsilon$$

$$C \to cC|c|\varepsilon$$

$$B \to bbB|b$$

The corresponding DFA is:



Exercise. *Let the regular expressions* r_1 *and* r_2 *be defined as* $r_1 = (r|s)^*$ *and* $r_2 = (r^*|s^*)$ *with* r, s *arbitrary regular expressions.*
Is $L(r_1) = L(r_2)$ *?*

Not, they are not the same. In fact, if we take $r = ab$ and $s = ba$, we can see that:
"abba" $\in L(r_1)$
"abba" $\notin L(r_2)$

Exercise. *Given the grammar*

$$S \to aBc$$

$$B \to aBc|b$$

and the parsing table

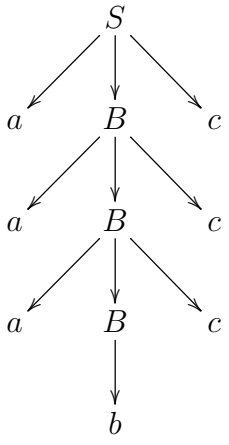
	a	***b***	***c***	$
S	$S \to aBc$	*error*	*error*	*error*
B	$B \to aBc$	$B \to b$	*error*	*error*

say whether or not the input aaabccc belongs to the language.

The moves made by a predictive parser are

stack	**input**	**output**
$S	aaabccc$	
cB\underline{a}$	$\underline{a}$aabcc$	$S \to aBc$
ccB\underline{a}$	$\underline{a}$abcc$	$B \to aBc$
$cccB$\underline{a}$	$\underline{a}$bccc$	$B \to aBc$
ccc\underline{b}$ $\underline{b}$ccc$	$B \to b$	
...	...	..

From now, we can see that the three c correspond in the pattern matching. Thus, the word belongs to the grammar, and it is generated by the following parse tree



5.1.4 FIRST and FOLLOW

The construction of both top-down and bottom-up parsers is aided by two functions, **FIRST** and **FOLLOW**, associated with a grammar G. During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol.

FIRST

Define FIRST(α), where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α.

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ε can be added to any FIRST set.

1. if X is a terminal, then FIRST(X) = $\{X\}$.
2. if X is a nonterminal and $X \rightarrow Y_1 Y_2 ... Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ε is in all off FIRST(Y_1), ... , FIRST(Y_{i-1}). If ε is in FIRST(Y_j) for all $j = 1, 2, ..., k$, then add ε to FIRST(X).
3. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X).

Now, we can compute FIRST for any string $X_1, X_2, ...X_n$ as follows. Add to FIRST($X_1, X_2, ...X_n$) all non$-\varepsilon$ symbols of FIRST(X_1). Also add the non$-\varepsilon$ symbols of FIRST(X_2), if ε is in FIRST(X_1); the non$-\varepsilon$ symbols of FIRST(X_3), if ε is in FIRST(X_1) and FIRST(X_2); and so on.

FIRST(α)

1: **if** $\alpha = \varepsilon$ **then** FIRST(α) = $\{\varepsilon\}$;
2: **else if** $\alpha =$a **then** FIRST(α) = {a};
3: **else if** $\alpha = A$ **then**
4: check the productions for A;
5: **if** $A \to Y_1 ... Y_k$ **then**
6: **if** $\forall j : 1..k$ $\varepsilon \in$ FIRST(Y_j) **then**
7: insert ε into FIRST(α);
8: **end if**
9: **if** $\varepsilon \in$ FIRST(Y_1) $\cap$ ... $\cap$ FIRST(Y_{j-1}) $\wedge$ a $\in$ FIRST(Y_j) for some $j < k$ **then**
10: insert a into FIRST(α);
11: **end if**
12: **end if**
13: **end if**

Example.

Given the following grammar

$$E \to TE'$$

$$E' \to +TE'|\varepsilon$$

$$T \to FT'$$

$$T' \to *FT'|\varepsilon$$

$$F \to (E)|id$$

We can compute the FIRST as following:

FIRST(E)	$\sim$	FIRST(T)	
		if $\varepsilon \in$ first(T), first(E')	false
		if $\varepsilon \in$ first$(T) \cap$ first(E'), ε	false
FIRST(E')		$\{+, \varepsilon\}$	
FIRST(T)	$\sim$	FIRST(F)	
		if $\varepsilon \in$ first(F), first(T')	false
		if $\varepsilon \in$ first(F) $\cap$ first(T'), ε	false
FIRST(T')		$\{*, \varepsilon\}$	
FIRST(F)		{(, id }	

Thus,

Nonterminal	FIRST
E	(id
E'	$+ \varepsilon$
T	(id
T'	$* \varepsilon$
F	(id

FOLLOW

Define FOLLOW(A), for nonterminal A, to be the set of terminals α that can appear immediately to the right of A in some sentential form; that is, the set of terminals α such that there exists a derivation of the form $S \rightarrow^* \alpha A \mathrm{a} \beta$, for some α and some β.

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input tight endmarker.

2. If there is a production $A \rightarrow \alpha B \beta$, then everything is in FIRST(β) except ε is in FOLLOW(B).

3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$, where FIRST($\beta$) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

Example.

Consider the grammar

$$E \to TE^{'}$$

$$E^{'} \to +TE^{'}|\varepsilon$$

$$T \to FT^{'}$$

$$T^{'} \to *FT^{'}|\varepsilon$$

$$F \to (E)|id$$

then:

1. FOLLOW(E) = FOLLOW($E^{'}$= $\{), \$\}$. Since E is the start symbol, FOLLOW(E) must contain \$. The production body (E) explains why the right parenthesis is in FOLLOW(E). For $E^{'}$, note that this non-terminal appears only at the ends of bodies of E-productions. Thus, FOLLOW($E^{'}$) must be the same as FOLLOW(E).

2. FOLLOW(T) = FOLLOW($T^{'}$) = $\{+,), \$\}$. Notice that T appears in bodies only followed by $E^{'}$. Thus, everything except ε that is in FIRST($E^{'}$) must be in FOLLOW(T); that explains the symbol +. However, since FIRST($E^{'}$) contains ε, and $E^{'}$ is the entire string following T in the bodies of the E-productions, everything in FOLLOW(E) must also be in FOLLOW(T). That explains the symbols \$ and the right parenthesis. As for $T^{'}$, since it appears only at the ends of the T-productions, it must be that FOLLOW($T^{'}$) = FOLLOW(T).

3. FOLLOW(F) = $\{+, *,), \$\}$. The reasoning is analogous to that for T in point (2).

Thus, the FOLLOW is:

Nonterminal	FOLLOW
E	\$)
T	+ \$)
E'	\$)
F	* + \$)
T'	+ \$)

5.1.5 LL(1) Grammars

Predictive parsers, that is, recursive-descendent parsers needing no baktracking, can be constructed for a class of grammar called LL(1). The first "L" stands for scanning the input from left to right, the second "L" for producing a leftmost derivation, and the "1" for using one input symbol of lookahead at each step to make parsing action decisions.

NB: No left-recursive or ambiguous grammar can be LL(1)

A grammar G is **LL(1)** if and only if whenever $A \to \alpha|\beta$ are two distinct productions of G, the following conditions hold:

1. For no terminal a do both α and β derive strings beginning with a.
2. At most one of α and β can derive the empty string.
3. If $\beta \to^* \varepsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A). Lokewise, if $\alpha \to^* \varepsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A).

The next algorithm collects the information from FIRST and FOLLOW sets into a predictive parsing table $M[A, a]$, a two-dimensional array, where A is a nonterminal and a is a terminal or the symbol \$, the input endmaker.

The algorithm is based on the following idea: the production $A \to a$ is chosen if the next input symbol a is in FIRST(a). The only complication occurs when $a = \varepsilon$ or, more generally, $a \to^* \varepsilon$. In this case, we should again choose $A \to a$ if the current input symbol is in FOLLOW(A), or if the \$ on the input has been reached and \$ is in FOLLOW(A).

Construction of a non-recursive top-down (predictive) parsing table

---

1: **for each** $A \to \alpha$ in the grammar G **do**
2: $\quad \forall a \in \text{first}(\alpha)$, set $M[A, a] = A \to \alpha$;
3: $\quad$ **if** $\varepsilon \in \text{first}(\alpha)$ **then**
4: $\quad\quad \forall x \in \text{follow}(A)$, set $M[A, x] = A \to \alpha$; $\quad \triangleright\ x$ can be \$
5: $\quad$ **end if**
6: **end for**
7: set all the empty entries to *error()*;

NB: A grammar G can be an LL(1) Grammar if its parsing table has no multiple defined entries.

Example.

If we consider the grammar

$$E \to TE^{'}$$

$$E^{'} \to +TE^{'}|\varepsilon$$

$$T \to FT^{'}$$

$$T^{'} \to *FT^{'}|\varepsilon$$

$$F \to (E)|id$$

and we use the previous algorithm to fill the parsing table, we obtain:

	id	+	*	(	)	$
E	$E \to TE^{'}$			$E \to TE^{'}$		
$E^{'}$		$E^{'} \to +TE^{'}$			$E^{'} \to \varepsilon$	$E^{'} \to \varepsilon$
T	$T \to FT^{'}$			$T \to FT^{'}$		
$T^{'}$		$T^{'} \to \varepsilon$	$T^{'} \to *FT^{'}$		$T^{'} \to \varepsilon$	$T^{'} \to \varepsilon$
F	$F \to$ id			$F \to (E)$		

Consider production $E \to TE^{'}$. Since FIRST(T$E^{'}$) = FIRST(T) = $\{(, id\}$, this production is added to $M[E,)]$ and $M[E, id]$. Production $E^{'} \to +TE^{'}$ is added to $M[E^{'}, +]$ since FIRST(+T$E^{'}$*) = $\{+\}$. Since FOLLOW($E^{'}$) = $\{), \$\}$, production $E^{'} \to \varepsilon$ is added to $M[E^{'},)]$ and $M[E^{'}, \$]$.

NB:

Predictive Parsing Table : a table where we put on one side nonterminals of the grammar, and on the other side we put terminals and the special symbol \$.

FIRTS(α) : all the possible starting terminals of strings derivable from α; if ε is derivable from α, then εin FIRST(α).

FOLLOW(A) (A is a nonterminal): tge first(α) for all the possible α that can occurr at the right of A in any derivation.

- $A = S$ insert \$ in FOLLOW(A);
- otherwise:
  - $B \rightarrow \alpha A \beta$: insert FIRST($\beta$) $\setminus$ $\{\varepsilon\}$ and FOLLOW(β) if $\varepsilon \in$ first(β);
  - $\beta \rightarrow \alpha A$: insert FOLLOW($\beta$).

NB: A language L is LL(1) if there exists an LL(1) grammar G such that L(G) = L.

A language L can be generated by more than one grammar.

- If G is LL(1), then the predictive paring algorithm recognized a word ω if and only if ωin L(G).
- Every ambiguous grammar is not LL(1).
- Every left-recursive grammar is not LL(1).
- G is LL(1) if and only if G has productions $A \to \alpha|\beta$ then:
  1. first(α) $\cap$ first(β) = $\emptyset$;
  2. if $\varepsilon \in$ first(α), then first(β) $\cap$ follow(A) = $\emptyset$ and viceversa.

5.1.6 Exercises

Exercise. *Say whether or not the following grammars is LL(1)*

$$S \to AaAb|BbBa$$

$$A \to \varepsilon$$

$$B \to \varepsilon$$

---

1. Calculate the FIRST of the nonterminals.

   - As $A \to \varepsilon$ and $B \to \varepsilon$ are two productions, we have to insert ε in FIRTS(A) and FIRST(B).
   - Consider the production $S \to AaAb$; we have to insert:
     - all not-empty symbols of FIRST(A) $\to \{\}$;
     - all not-empty symbols of FIRST(a) if $\varepsilon \in$ FIRST(A) $\to \{a\}$.
     - all not-empty symbols of FIRST(A) if $\varepsilon \in$ FIRST(A) $\wedge$ $\varepsilon \in$ FIRST(a) $\to \{\}$.
     - ...
   - Consider the production $S \to BbBa$; we have to insert:
     - all not-empty symbols of FIRST(B) $\to \{\}$;
     - all not-empty symbols of FIRST(b) if $\varepsilon \in$ FIRST(B) $\to \{b\}$.
     - all not-empty symbols of FIRST(B) if $\varepsilon \in$ FIRST(B) $\wedge$ $\varepsilon \in$ FIRST(b) $\to \{\}$.
     - ...

So, the FIRST is:

	FIRST
S	a b
A	ε
B	ε

2. Calculate the FOLLOWING of non-terminals.

   - Place $ in FOLLOW(S).
   - $S \to AaAb$: is a production of the shape $C \to \alpha D \beta$, so everything in FIRST(β) except ε is in FOLLOW(D); so we can add b to FOLLOW(A).
   - $S \to AaAb$: it can be also seen as a production $C \to \alpha D \beta$, where α is empty and β is aAb. So, we have to insert into FOLLOW(A) the FIRST(aAb), which is FIRST(a) and so a; thus, we can add a to FOLLOW(A).
   - The same considerations hold for the production rule $S \to BbBA$.

The FIRST and FOLLOW sets of the nonterminals are:

	FIRST	**FOLLOW**
S	a b	$
A	ε	a b
B	ε	b a

3. Bulding the parsing table.

   - $S \to AaAb$: the FIRST(AaAb) is given by the FIRST(A) and, as FIRST(A) contains ε, by the FIRST(a); so, it is $\{a, \varepsilon\}$. Thus, we have to insert this production rule in $M[S, a]$ and, as FIRST(AaAb) contains ε, we have to insert this production also in the $M[S, t]$, foreach $t \in$ FOLLOW(S), and so in $M[S, \$]$.
   - $S \to BbBa$: we have to insert this production in the row S for each column representing an element of the FIRST(BbBa), which is $\{b, \varepsilon\}$. As FIRST(BbBa) also contains ε, we have to insert this production also in the row S for each column representing an element of FOLLOW(S), which is $.

- $A \to \varepsilon$; as the FIRST(ε) is ε, which obviously contains ε, we have to add this production in row A for each column representing an element of FOLLOW(A), which is $\{a, b\}$.
- $B \to \varepsilon$: the previous consideration holds also for this production.

So, the parsing table is:

	a	**b**	$
S	$S \to AaAb$	$S \to BbBa$	$S \to AaAb, S \to BbBa$
A	$A \to \varepsilon$	$A \to \varepsilon$	err
B	$B \to \varepsilon$	$B \to \varepsilon$	err

This is not an LL(1) grammar, because we have at least one multiple defined entry.

Exercise. *Say whether or not the following grammar is LL(1):*

$$Z \to d|XYZ$$

$$Y \to \varepsilon|c$$

$$X \to Y|a$$

We have to calculate the first and the follow of the items.
About the follow, we know that:

- FOLLOW(Z) = $;
- FOLLOW(Y) = FOLLOW(X);
- FOLLOW(X) = FIRST(YZ) = FIRST(Y) $\cup$ FIRST(Z) $\setminus\{\varepsilon\}$;
- FOLLOW(Y) = FIRST(Z);
- FOLLOW(Z) = FOLLOW(Z).

So, we have that;

	FIRST	**FOLLOW**
X	c a ε	c a d
Y	c ε	c a d
Z	d c a	$

The grammar is not LL(1), because the table has at leas one multiple defined entry:

$$M[Y, c] = \{Y \to \varepsilon, Y \to c\}$$

Exercise. *Say whether or not the following grammar is LL(1):*

$$S \to iBtS|iBtSeS|a$$

$$B \to b$$

This grammar is not LL(1), because there is the same prefix ($iBts$) for at least two productions.
We can try to factorize this grammar, obtaining:

$$S \to iBtSS'|a$$

$$S' \to \varepsilon|eS$$

$$B \to b$$

Unfortunately, this grammar is ambiguous and thus is not LL(1).

Exercise. *Say whether the following grammar is LL(1):*

$$B \to TB^{'}$$

$$B^{'} \to orTB^{'}|\varepsilon$$

$$T \to FT^{'}$$

$$T^{'} \to andFT^{'}|\varepsilon$$

$$F \to notF|(B)|true|false$$

- Computation of FIRST and FOLLOW:

	FIRST	**FIRST**
B	not (true false	$)
B′	or ε	$)
T	not (true false	or $)
T′	and ε	or $
F	not (true false	and or $)

If we contruct the parsing table, we can see that this grammar is LL(1).

Exercise. ***Computation of FIRST and FOLLOW***

Given the grammar

$$E \to TE'$$

$$E' \to +TE'$$

$$E' \to \varepsilon$$

$$T \to FT'$$

$$T' \to *FT'$$

$$T' \to \varepsilon$$

$$F \to (E)|id$$

calculate the FIRST and the FOLLOW.

FIRST

- If there is a production $X \to \varepsilon$, then add ε to FIRST(X).
  FIRST(E') = $\{\varepsilon\}$
  FIRST(T') = $\{\varepsilon\}$

- $T' \to *FT$
  We have to add everything in FIRST(*FT) to FIRST(T') (FIRST(*FT) = FIRST(*) = *); as ε doesn't appear in FIRST(*), we don't add FIRST(F) and FIRST(T).

- $F \to (E)$
  We have to add everything in FIRST((E)) to FIRST(F). (FIRST((E)) = FIRST(() = (; as ε doesn't appear in FIRST((), we can't add FIRST(E) and FIRST()).

- $F \to id$
  As above, we add to FIRST(F) the symbol id.

- $T \rightarrow FT^{'}$
  We cann add FIRST(FT$^{'}$) to FIRST(T). Since FIRST(F) doesn't contain ε, that means that FIRST(FT$^{'}$) is just FIRST(F) = $\{(, id\}$.

- $E \rightarrow TE^{'}$
  We can add FIRST(TE$^{'}$) to FIRST(E). Since FIRST(T) doesn't contain ε, that means that FIRST(TE$^{'}$) is just FIRST(T) = $\{(, id\}$.

Thus, the FIRST is:

Symbol	**FIRST**
$E^{'}$	ε, +
$T^{'}$	ε, *
F	(, id
E	(, id
T	(, id

FOLLOW

There are 4 main rules to compute the FOLLOW:

1. Add $ to the start symbol.
2. If there is a production $A \rightarrow aBb$, everything in FIRST(b) except for ε is placed in FOLLOW(B).
3. If there is a production $A \rightarrow aB$, then everything in FOLLOW(A) is in FOLLOW(B).
4. If there is a production $A \rightarrow aBb$, where FIRST(b) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

- $E^{i} \rightarrow +TE^{'}$
  Because of rule 2: FOLLOW(T) = FIRST($E^{'}$) = {+}.

- $E \rightarrow TE^{'}$
  Because of rule 3: FOLLOW($E^{'}$) = FOLLOW(E) = {$}.

- $T \rightarrow FT^{'}$
  Because of rule 3: FOLLOW($T^{'}$) = FOLLOW(T) = {+}.

- $T' \rightarrow *FT'$
  Because of rule 2: FOLLOW(F) = FIRST(T') $\setminus \varepsilon = \{*\}$.

- $F \rightarrow (E)$
  Because of rule 2: FOLLOW(E) = FIRST()) = {)}. Now, we have to update E'.
  $FOLLOW(F) = \{)\}$.

- $E' \rightarrow +TE'$
  Because of rule 4: FOLLOW(T) = FOLLOW(E') = {$,)}. Now, we have to update T'.

- $T' \rightarrow *FT'$
  Because of rule 4: FOLLOW(F) = FOLLOW(T') = {+, $}.

Thus, the FOLLOW is:

SYMBOL	**FOLLOW**
E	$,)
E'	$,)
T	+, $,)
T'	+, $,)
F	*, +, $,)

5.2 Bottom-Up Parsing

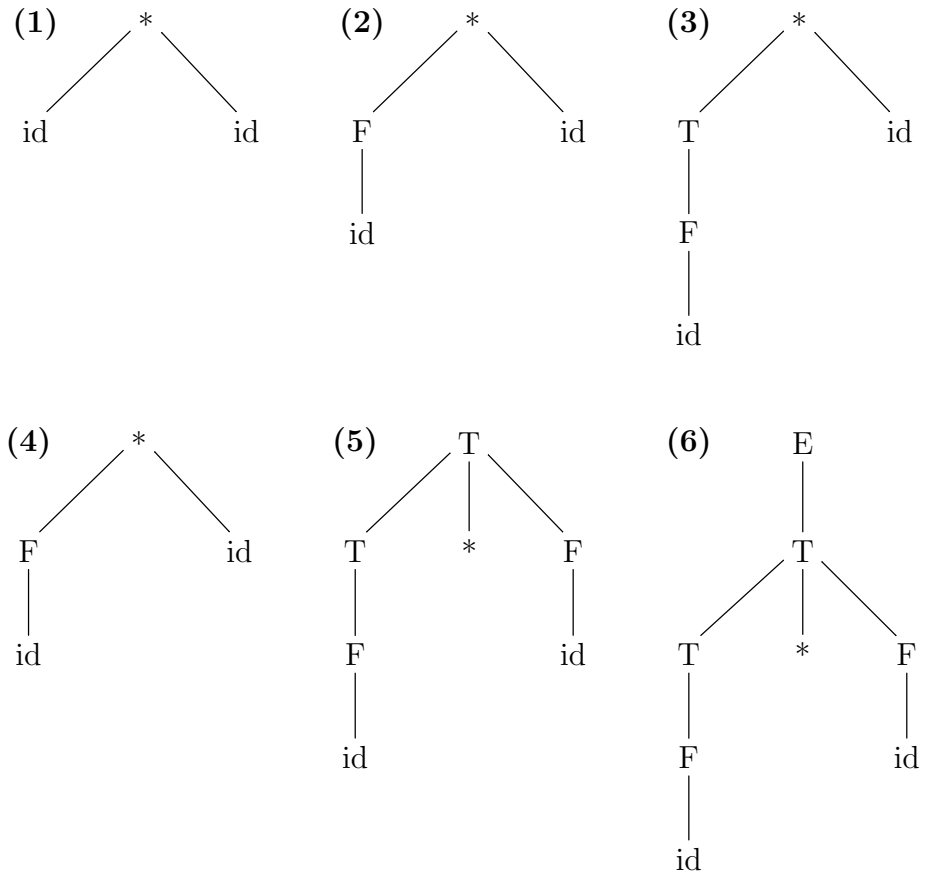
A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and qorking up towards the root (the top).

The following sequence of tree snapshots illustrates a bottom-up parse of the token stream `id * id`, with respect to the expression grammar

$$E \to E + T | T$$

$$T \to T * F | F$$

$$F \to (E) | id$$



5.2.1 Reductions

We can think of bottom-up parsing as the process of "reducing" a string w to the start symbol of the grammar. At each *reduction* step, a specific substring matching the body of a production is replaces by the nonterminal at the head of that production.
The key decision during bottom-up parsing are about when to redcue and about what production to apply, as the parse proceeds.
In the previous example, the reductions will be discussed in terms of the sequence of strings

$$id * id, F * id, T * id, T * F, T, E$$

By definition, a reduction is the reverse of a step in a derivation (**NB**: in a derivation, a nonterminal in a sentential form is replaced by the body of one of its productions). The goal of bottom-up parsing is therefore to construct a derivation in reverse. The following derivation corresponds to the previous parse:

$$E \to T \to T * F \to T * id \to F * id \to id * id$$

Tis derivation is in fact a rightmost derivation.

5.2.2 Handle Pruning

Bottom-up parsing during a left-to-right scan of the input constructs a rightmost derivation in reverse.
Informally, a **handle** is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.

For example, adding subscripts to the tokens `id` for clarity, the handles during the parse of $\texttt{id}_1$ `*` $\texttt{id}_2$ according to the expressione grammar

$$E \to E + T | T$$

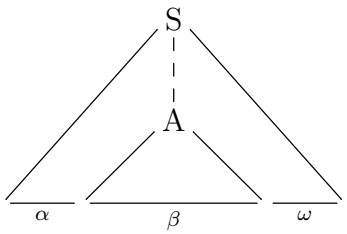
$$T \to T * F | F$$

$$F \to (E) | id$$

are as following:

Right Sentential Form	**Handle**	**Reducing Production**
id_1 * id_2	id_1	F → id
F * id_2	F	T → F
T * id_2	id_2	F → id
T * F	T * F	E → T * F

Formally, if $S \to^*_{rm} \alpha A \omega \to_{rm} \alpha\beta\omega$, then production A → β in the position following α is a handle of $\alpha\beta\omega$.



Alternatively, a handle of a right-sentential form γ is a production A → β and a position of γ where the string β may be found, such that replacing β at the position by A produces the previous right-sentential form in a rightmost derivation of γ.

Notice that the string ω tot he right of the handle must contain only terminal symbols. For convenience, we refer to the body β rather than A → β as a handle.

Note that we say "a handle" rather than "the handle", because the grammar could be ambiguous, with more than one rightmost derivation of $\alpha\beta\omega$. If a grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.

A rightmost derivation in reverse can be obtained by "handle pruning". That

is, we start with a string of terminals ω to be parsed. If ω is a sentence of the grammar at hand, then let $\omega = \gamma_n$, where γ_n is the nth right-sentential form of some as yet unknown rightmost derivation

$$S = \gamma_0 \rightarrow_{rm} \gamma_1 \rightarrow_{rm} \gamma_2 \rightarrow ... \rightarrow \gamma_n = \omega$$

To reconstruct this derivation in reverse order, we locate the handle β_n in γ_n and replace β_n by the head of the relevant production $A_n \rightarrow \beta_n$ to obtain the previous right- sentential form γ_{n-1}. We the repeat this process, that is, we locate the handle $\beta n - 1$ in γ_{n-1} and reduce this handle to obtain the right-sentential form γ_{n-1}. If by continuing this process we produce a right-sentential form consisting only of the start symbol S, then we halt and announce successful completion of parsing.

5.2.3 Shift-Reduce Parsing

Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.
We use $ to mark the bottom of the stack and also the right end of the input.

Initially, the stack is empty, and the string ω is on the input, as follows:

Stack	**Input**
$	$\omega$$

During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until this is ready to reduce a string β of grammar symbosl on top of the stack. It then reduces β to the head of the appropriate production. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty.

If ha shift-reduce parser had taken in parsing the input string $\texttt{id}_1$ `*` $\texttt{id}_2$ according to the expression grammar

$$E \rightarrow E + T|T$$

$$T \rightarrow T * F | F$$

$$F \rightarrow (E) | id$$

, it would execute the following steps:

Stack	**Input**	**Action**
$	id_1 * id_2 $	shift
$ id_1	* id_2 $	reduce by F → id
$ F	* id_2 $	reduce by T → F
$ T	* id_2 $	shift
$ T *	id_2 $	shift
$ T * id_2	$	reduce by F → id
$ T * F	$	reduce by T → T * F
$ T	$	reduce by E → T
$ E	$	accept

There are actually four possible actions a shift-reduce parser can make:

Shift : shift the next input symbol onto the top of the stack;

Reduce : the right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string;

Accept : announce successful completion of parsing.

Error : discover a syntax error and call an error recovery routine.

The use of a stack in shift-reduce parsing is justified by an important fact: the handle will always eventually appear on top of the stack, never inside.

Conflicts during shift-reduce parsing

There are context-free grammars for which shift-reduce parsing cannot be used. Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack contents and the next input

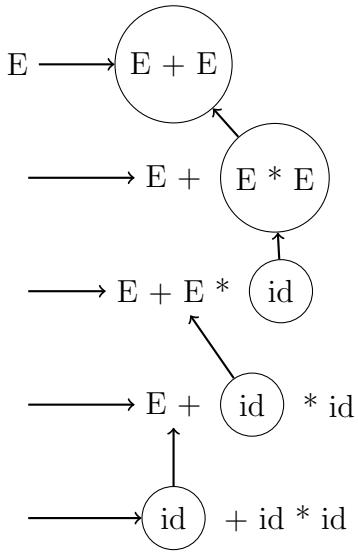
symbol, cannot decide whether to shift or to reduce (*shift/reduce conflict*), or cannot decide which of several reductions to make (a *reduce/reduce conflict*).

Example.

Given the grammar

$$E \to E + E | E * E | (E) | id$$

, if we would parse the string `id + id * id`, we would find the following handles:



The parsing of the word would be:

stack	**input**	
$	$\underline{\text{id}}$ + id * id $	(S)
$ id	+ id * id $	(R: $E \to id$)
$ E	$\underline{+}$ id * id $	(S)
$ E +	$\underline{\text{id}}$ * id $	(S)
$ E + id	* id $	(R: $E \to id$)
$ E + E	* id $	(R: $E \to E + E$)
$ E	$\underline{*}$ id $	(S)
$ E *	id $	(S)
$ E * id	$	(R: $E \to id$)
$ E * E	$	(R: $E \to E * E$)
$ E	$	

5.3 LR Parsing

The most prevalent type of bottom-up parser today is based on a concept called LR(k) parsing; the "L" is for left-to-right scanning of the input, the "R" for constructing a rightmost derivation in reverse, and the "k" for the number of input symbols of lookahead that are used in making parsing decisions. The cases $k = 0$ and $k = 1$ are of practical interest, and we shall consider only LR parsers with $k \leq 1$; when (k) is omitted, k is assumed to be 1.

For a grammar to be LR it is sufficient that a left-to-right shift-reduce parses be able to recognize handles of right-sentential forms when they appear on top of the stack.

Items and LR(0) Automaton

How does a shift-reduce parser know when to shift and when to reduce?
An LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in a parse. States represent *sets of items*; an LR(0) item (**item** for short) of a grammar G is a production of G with a dot at some position of the body. Thus, the production $A \to XYZ$ yelds the four items

$$A \to .XYZ$$

$$A \to X.YZ$$

$$A \to XY.Z$$

$$A \to XYZ.$$

The production $A \to \varepsilon$ generates only one item, $A \to .$
Intuitively, an items indicates how much of a production we have seen at a given point in the parsing process. For example:

- the item $A \to .XYZ$ indicates that we hope to see a string derivable from XYZ next on the input.

- $A \rightarrow X.YZ$ indicates that we have just seen on the input a string derivable from X and that we hope next to see a string derivable from YZ.

- $A \rightarrow XYZ.$ indicates that we have seen the body XYZ and that it may be time to reduce XYZ to A.

One collection of sets of LR(0) items, called the *canonical* LR(0) collection, provides the basis for constructing a determinstic finite automaton that is used to make parsing decisions; in particular, each state of the LR(0) automaton represents a set of items in the canonical LR(0) collection.

To contruct the canonical LR(0) collection for a grammar, we deine an augmented grammar and two functions, CLOSURE and GOTO. If G is a grammar with the start symbol S, then $G^{'}$, the *augmented* grammar for G is G with a new start symbol $S^{'}$ and production $S^{'} \rightarrow S$. The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input. That is, acceptance occurs when and only when the parser is about to reduce by $S^{'} \rightarrow S$.

Closure of Item Sets : if I is a set of items for a grammar G, then CLOSURE(I) is the set of items constructed from I by the two rules:

1. Initially, add every item in I to CLOSURE(I).
2. If $A \rightarrow \alpha.B\beta$ is in CLOSURE(I) and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow .\gamma$ to CLOSURE(I), if it is not already there. Apply this rule until no more new items can be added to CLOSURE(I).

Example.

Consider the augmented expression grammar:

$$E^{'} \rightarrow E$$

$$E \rightarrow E + T|T$$

$$T \rightarrow T * F|F$$

$$E \rightarrow (E)|id$$

To see how closure is compute, $E^{'} \rightarrow E$ is put in CLOSURE(I) by rule (1). Since there is an E immediately to the right of a dot, we add the E-productions with dots at the left ends: $E \rightarrow .E + T$ and $E \rightarrow .T$. Now, there is a T immediately to the right of a dot in the latter item, so we add $T \rightarrow .T * F$ and $T \rightarrow .F$. Next, the F to the right of a dot forces us to add $F \rightarrow .(E)$ and $F \rightarrow .id$, but no other items need to be added.

Function Closure(I)

```
1: J := I
2: repeat
3:     foreach A → α.β in J ∧
4:     foreach B → γ in G do
5:     if B → .γ is not in J then
6:         add it to J;
7:     end if
8: until saturarion                    ▷ nothing more can be added
```

The Function GOTO : the second useful function si GOTO(I, X) where I is a set of items and X is a grammar symbol. GOTO(I, X) is defined to be the closure of the set of all items $[A \rightarrow \alpha X.\beta]$ such that $[A \rightarrow \alpha.X\beta]$

is in I. Intuitively, the GOTO function is used to define the transitions in the LR(0) automaton for a grammar. The states of the automaton correspond to sets of items, and GOTO(I, X) specifies the transition from the state for I under input X.

Example.

If I is the set of two items $\{[E' \rightarrow E.], [E \rightarrow E. + T]\}$, then GOTO(I, +) contains the items:

$$E \rightarrow E + .T$$

$$T \rightarrow .T * F$$

$$T \rightarrow .F$$

$$F \rightarrow .(E)$$

$$F \rightarrow .id$$

We computed GOTO(I, +) by examining I for items with + immediately to the right of the dot. $E' \rightarrow E.$ is not such an item, but $E \rightarrow E. + T$ is. We moved the dot over the + to get $E \rightarrow E + .T$ and then took the closure of this singleton set.

We are now ready for the algorithm to construct C, the canonical collection of sets of LR(0) items for an augmented grammar G'.

Generation of the collection of items for G'

```
1: C = [CLOSURE{S' → S}]
2: repeat
3:     for (each set of items I in C) do
4:         for (each grammar symbol X ) do
5:             if (GOTO(I, X) is not empty and not in C yet) then
6:                 add GOTO(I, X) to C;
7:             end if
```

8: **end for**
9: **end for**
10: **until** Saturation

The central idea behind "Simple LR", or **SLR**, parsing is the construction from the grammar of the LR(0) automaton. The states of this automaton are the sets of items from the canonical LR(0) collection, and the transitions are given by the GOTO function.
The start state of LR(0) automaton is CLOSURE($\{[S' \to .S]\}$), where S' is the start symbol of the augmented grammar.

How can LR(0) automata help with shift-reduce decisions? Shift-reduce decisions can be made as follows; suppose that the string γ of grammar symbols takes the LR(0) automaton from the start state 0 to some state j. Then, shift on next symbol a if state j has a transition on a. Otherwise, we choose to reduce; the items in state j will tell us which production to use.

Example.
Given tha grammar

$$E \to E + T|T$$

$$T \to T * F|T$$

$$F \to (E)|id$$

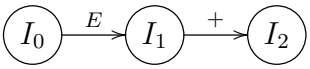
we have to:

- enrich the grammar with the production: $E' \to E$;
- compute the closure $\{E' \to E\}$: $I_o = \{E' \to .E, E \to .E + T,$ $E \to .T, T \to .T * F, T \to .F, F \to .(E), F \to .id\}$
- compute the sets:
  - $I_1 = \text{goto}(I_0, \text{E}) = \{E' \to E., E \to E. + T\}$

- $I_2 = \text{goto}(I_1, +) = \{E \to E + .T, T \to .T * F, T \to .F,$
  $F \to .(E), F \to .id\}$

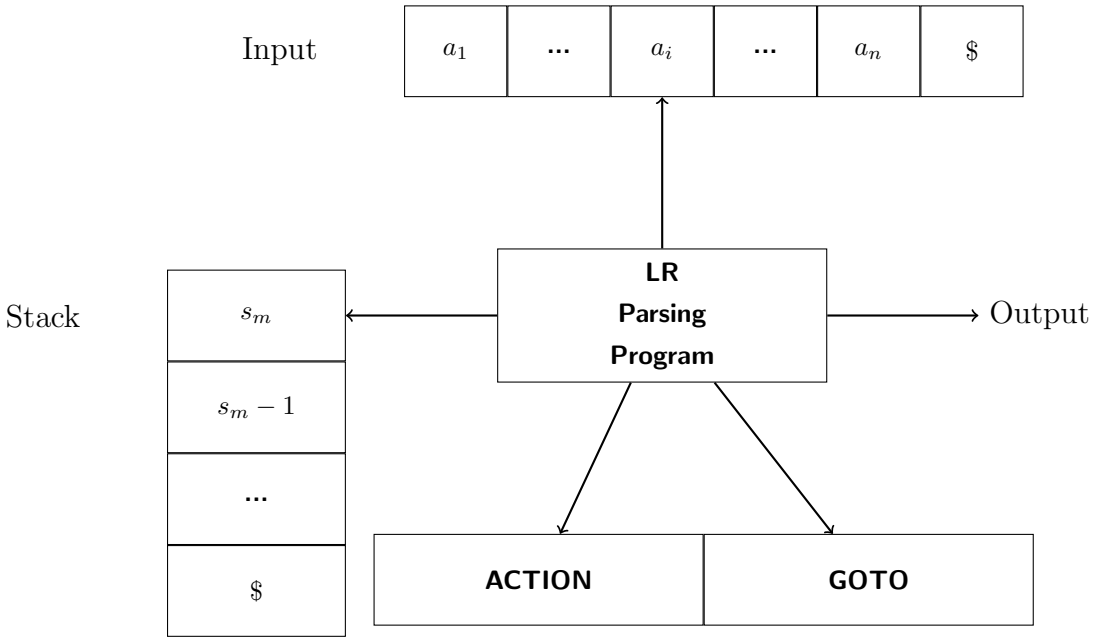
- continue in computing the other sets ...

So, we will obtain an automaton which will "start" with this shape:



5.4 The LR-Parsing Algorithm

An LR parser consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (ACTION and GOTO).



The driver program is the same for all LR parsers; only the parsing table changes from one parser to another.

Structure of the LR Parsing Table

The parsing table consists of two parts: a parsing-action function ACTION and a goto function GOTO.

1. The ACTION function takes as arguments a state i and a terminal a (or \$, the input endmarker). The value of ACTION$[i, a]$ can have one of four forms:

   (a) Shift j, where j is a state. The action taken by the parser effectively shifts input a to the stack, but uses state j to represent a.

   (b) Reduce $A \rightarrow \beta$. The action of the parser effectively reduces β on the top of the stack to head A.

   (c) Accept. The parser accepts the input and finishes parsing.

   (d) Error. The parser discovers an error in its input and takes some corrective action.

2. We extend the GOTO function, defined on sets of items, to states: if GOTO$[I_i, A] = I_j$, then GOTO also maps a state i and a nonterminal A to state j.

The next move of the parser is determined by reading a_i, the current input symbol, and s_m, the state on top of the stack, and then consulting the entry ACTION$[s_m, a_i]$ in the parsing action table. The LR-parsing algorithm is summarized below:

INPUT : An input string ω and an LR-parsing table with functions ACTION and GOTO for a grammar G.

OUTPUT : If ω is in L(G), the reduction steps of a bottom-up parse for ω; otherwise, an error indication.

```
1: init 0 in the stack                          ▷ we start in state number 0
2: init ω $ is the input buffer
3: init ip pointing to the leftmost symbol of ω
4: repeat
5:     n:= top; a:= ↑ ip                        ▷ a is the symbol pointed by ip
6:     if T[n, a] = s_m then
7:         push(a);
8:         push(m);
9:         move ip to the right;
10:    else if T[n, a] = r_k with (k): A → β then
11:        pop 2 * |β| symbols;
12:        n' := top;
13:        push(A);
14:        push m with m such that T[A, n'] = g_m;
15:        output "A → β";
16:    else if T[n, a] = accept then
17:        accept();
18:    else
19:        error();
20:    end if
21: until accept() or error()
```

5.4.1 Constructing SLR-Parsing Tables

The SLR method begins with LR(0) items and LR(0) automata; that is, given a grammar G, we augment G to produce $G^{'}$, with a new start symbol S'. From $G^{'}$, we construct C, the canonical collection of sets of items for $G^{'}$ together with the GOTO function.

The ACTION and GOTO entries in the parsing table are then constructed using the following algorithm. It requires us to know FOLLOW(A) for each nonterminal A of a grammar.

This algorithm takes:

Input : $G^{'}$ (the enriched grammar)

Output : parsing table or error "G is not SLR"

Generation of SLR parsing table

- **Define** the collection $[I_0, ..., I_n]$ of LR(0) sets of items for $G^{'}$
- **Define** the state i in the table (i.e. row i) after the elements of I_i as follows:
  - * **if** $A \rightarrow \alpha.a\beta \in I_i$ and goto(I_i, a) = I_j **then**
    - T$[i, a] := s_j$; ▷ shift
  - * **if** $A \rightarrow \alpha. \in I_i$ **then**
    - **foreach** x ∈ FOLLOW(A)
      - T$[i, X]$:= r"A→ α"; ▷ reduce
  - * **if** $S^{'} \rightarrow S. \in I_i$ **then**
    - T$[i, \$]$:= accept;
- **if** the previous stage generates conflicts **then**
  - output "G is not SLR";
- **else**
  - **if** goto(I_i, A) = I_j **then**
    - - T$[i, a] = g_j$; ▷ goto
  - **end if**
  - - **Put** error() in every empy entry;
  - - I_0 is CLOSURE($S^{'} \rightarrow .S$).
- **end if**

Example.

Let us construct the SLR table for the following grammar:

$$(1) E \to E + T$$

$$(2) E \to T$$

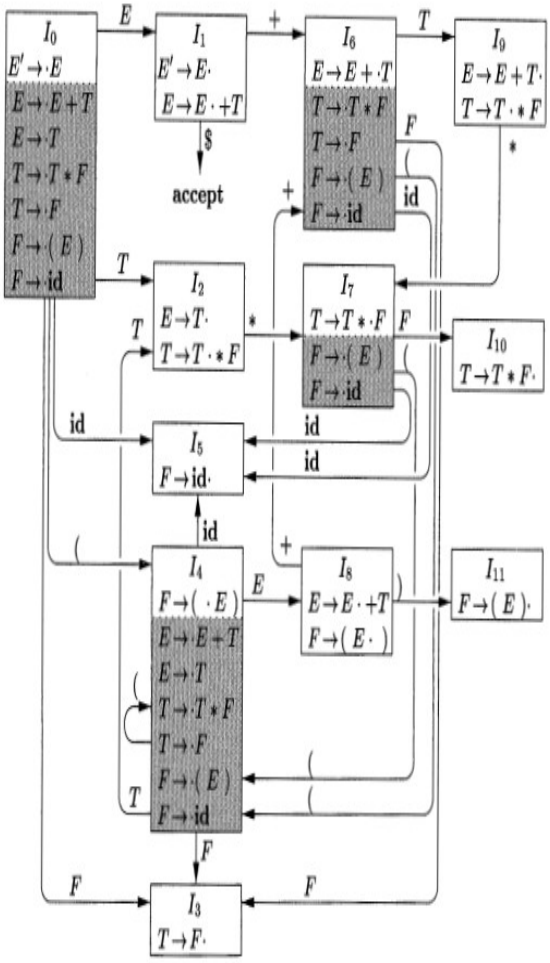
$$(3) T \to T * F$$

$$(4) T \to F$$

$$(5) F \to (E)$$

$$(6) F \to id$$

After we have enriched the grammar with the rule $E^{'} \to E$, we can write the LR(0) automaton for this expression grammar, which is:



First consider the set of item I_0; the item $F \to .(E)$ gives rise to the entry ACTION[0,(] = shift 4, and the item $F \to .id$ to the entry ACTION[0,id] = shift 5. Other items in I_0 yeld no actions.

Now consider I_1; the first item yields ACTION[1,$] = accept, and the second yelds ACTION[1,+] = shift 6.

Next consider I_2; since FOLLOW(E) = $\{\$, +,)\}$, the first item makes ACTION[2,$] = ACTION[2,+] = ACTION[2,)] = reduce $E \to T$. The second item makes ACTION[2,*] = shift 7. Continuing in this fashion, we obtain the following table:

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

On output `id * id + id`, the sequence of stack and input contents is shown below:

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		id * id + id $	shift
(2)	0 5	id	* id + id $	reduce by $F \to$ id
(3)	0 3	F	* id + id $	reduce by $T \to F$
(4)	0 2	T	* id + id $	shift
(5)	0 2 7	T *	id + id $	shift
(6)	0 2 7 5	T * id	+ id $	reduce by $F \to$ id
(7)	0 2 7 10	T * F	+ id $	reduce by $T \to T * F$
(8)	0 2	T	+ id $	reduce by $E \to T$
(9)	0 1	R	+ id $	shift
(10)	0 1 6	E +	id $	shift
(11)	0 1 6 5	E + id	$	reduce by $F \to$ id
(12)	0 1 6 3	E + F	$	reduce by $T \to$ F
(13)	0 1 6 9	E + T	$	reduce by $E \to E + T$
(14)	0 1	E	$	accept

For example, at line (1) the LR parser is in state 0, the initial state with no grammar symbol, and with is the first input symbol. The action in row 0 and column id of the action field of the table is s5, meaning shift by pushing state 5. That is what has happened at line (2); the state symbol 5 has been pushed onto the stack, and id has been removed frome the input.
Then, * becomes the current input symbol, and the action of state 5 on input * is to reduce by $F \to id$. One state symbol is popped off the stack. State 0 is then exposed. Since the goto of state 0 on F is 3, state 3 is pushed onto the stack; we now have the configuration in line (3). Each remaining moves is determined similarly.

Shift/reduce conflicts

Every SLR(1) grammar is unambiguous, but there are many unambiguous grammars that are not SLR(1). Consider the grammar with the productions

$$S \to L = R|R$$

$$L \to *R|id$$

$$R \to L$$

The canoncial collection of sets of LR(0) items for this grammar is the following:

- $I_0 : \{S' \to .S, S \to .L = R, S \to .R,\ L \to . * R, L \to .id, R \to .L\}$
- $I_1 : \{S' \to S.\}$
- $I_2 : \{S \to L. = R, R \to L.\}$
- $I_3 : \{S \to R.\}$
- $I_4 : \{L \to *.R, R \to .L, L \to . * R,\ L \to .id\}$
- $I_5 : \{L \to id.\}$

- $I_6 : \{S \to L = .R, R \to .L, L \to . * R,\ L \to .id\}$
- $I_7 : \{L \to *R.\}$
- $I_8 : \{R \to L.\}$
- $I_9 : \{S \to L = R.\}$

Consider the set of items I_2; the first item in this set makes ACTION(2, =) be "shift 6". Since FOLLOW(R) contains = (to see why, consider the derivation $S \to L = R \to *R = R$), the second item sets ACTION(2, =) to "reduce $R \to L$". Since there is both a shift and a reduce entry in ACTION(2, =), state 2 has a shift/reduce conflict on input symbol =.
Anyway, this grammar is not ambiguous. This shift/reduce conflict arises from the fact that the SLR parser construction method is not powerful enough to remember enough left context to decide what action the parser should take on input =, having seen a string reducible to L.

Viable Prefixes

The LR(0) automaton for a grammar characterizes the strings of grammar symbols that can appear on the stack of a shift-reduce parser for the grammar. The stack contents must be a prefix of a right-sentential form. If the stack holds α and the rest of the input is x, then a sequence of reductions will take αx to S.
Not all prefixes of right-sentential forms can appear on the stack, however, since the parse must not shift past the handle. For example, suppose

$$E \to^*_{rm} F * id \to (E) * id$$

Then, at various times during the parse, the stack will hold (, (E, and (E), but it most not hold (E)*, since (E) is a handle, which the parser must reduce to before shifting *.
The prefixes of right sentential forms that can appear on the stack of a shift-reduc parser are called *viable prefixes*. They are defined as follows: a **viable**

prefix is a prefix of a right-sentential form that does not continue past the right end of the rightmost handle of that sentential form. By this definition, it is always possible to add terminal symbols to the end of a viable prefix to obtain a right-sentential form.
We say item $A \to \beta_1 \times \beta_2$ is *valid* for a viable prefix $\alpha\beta_1$ if there is a derivation $S' \to^*_{rm} \alpha A\omega \to \alpha\beta_1\beta_2\omega$.
The fact that $A \to \beta_1 \times \beta_2$ is valid for $\alpha\beta_1$ tells us a lot about whether to shift or to reduce when we find $\alpha\beta_1$ on the parsing stack.

- if $\beta_1 \neq \varepsilon$, then it suggests that we have not yet shifted the handle onto the stack, so shift is our move.
- if $\beta_1 = \varepsilon$, then it looks as if $A \to \beta_1$ is the handle, and we should reduce by this production.

Unfortunately, we should not suppose that all parsing action conflicts can be resolved if the LR method is appled to an arbitrary grammar.

Resolving shift/reduce conflicts

Given the augmented grammar G'

$$(0)E' \to E$$

$$(1)E \to E + E$$

$$(2)E \to E * E$$

$$(3)E \to id$$

where FOLLOW(E) = $\{+, *, \$\}$, we can calculate C, the canonical collection of sets of LR(0) items for G', which is:

- I_0

  $E' \to .E$

  $E \to .E + E$

$E \to .E * E$

$E \to .id$

- I_1

  $E' \to E.$

  $E \to E. + E$

  $E \to E. * E$

- $I_2 : E \to id.$

- I_3

  $E \to E + .E$

  $E \to .E + E$

  $E \to .E * E$

  $E \to .id$

- I_4

  $E \to E * .E$

  $E \to .E + E$

  $E \to .E * E$

  $E \to .id$

- I_5

  $E \to E + E.$

  $E \to E. + E$

  $E \to E. * E$

- I_6

  $E \to E * E.$

$$E \to E. + E$$

$$E \to E. * E$$

The SLR(1) parsing table for G' is:

	id	+	*	$	**E**
0	s2				1
1		s3	s4	acc	
2		r3	r3	r3	
3	s2				5
4	s2				6
5		s3/r1	s4/r1	r1	
6		s3/r2	s4/r2	r2	

We can note that there is a shift/reduce conflict in action[5, +], because the associativity of the operator + is not defined by the grammar. This conflict can be resolved in favour of r1 if we want + to be left associative.
There is another shift/reduce conflict in action[5, *], because the relative precedence of the operators + and * is not defined by the grammar. This conflict can be resolved in favour of s4 if we want * to have higher precedence than +.

5.5 More Powerful LR Parsers

It is possible to extend the previous LR parsing tecniques to use one symbol of lookahead on the input. There are two different methods:

1. The canonical-LR or hust LR method, which makes full use of the lookahead symbols(s). This metod uses a large set of items, called the LR(1) items.

2. The lookahead-LR or LALR method, which is based on the LR(0) set of items, and has many fewer states than typical parsers based on the LR(1) items.

5.5.1 Canonical LR(1) Items

Recall that in the SLR method, state i calls for reduction by $A \to \alpha$ if the set of items I_i contains item $[A \to \alpha+]$ and α is in FOLLOW(A).

It is possible to carry more information in the state that will allow us to rule out some of these invalid reductions by $A \to \alpha$. By splitting states when necessary, we can arrange to have each state of an LR parser indicate exactly which input symbols can follow a handle α for which there is a possible reduction to A.
The extra information is incorpored into the state by redefining items to include a terminal symbol as a second component.
The general form of an item becoms $[A \to \alpha\beta, a]$, where $A \to \alpha\beta$ is a production and a is a terminal or the right endmarker \$. We call such an object an **LR(1) item**. The 1 refers to the length of the second component, called the **lookahead** of the item.
An item of the form $[A \to \alpha, a]$ calss for a refuction by $A \to \alpha$ only if the next input symbol is a. The set of such a's will always be a subset of FOLLOW(A), but it could be a proper set.

Constructing LR(1) Sets of Items

The method for building the collection of sets of valid LR(1) items is essentially the same as the one for building the canonical collection of sets of LR(0) items; we need only to modify the two procedures CLOSURE and GOTO.

---

closure function of LR(1) items

1: **repeat**
2: **for** each item $[A \to \alpha B\beta, a]$ in I **do**
3: **for** each production $B \to \gamma$ in G' **do**

```
4:              for each terminal b in FIRST(βα) do
5:                  add [B → .γ, b] to set I;
6:              end for
7:          end for
8:      end for
9:  until no more items are added to I; return I;
```

goto function of LR(1) items

```
1: initialize J to be the empty set;
2: for each item [A → αXβ, a] in I do
3:     add item [A → αXβ, a] to set J;
4: end forreturn CLOSURE(J);
```

items(G')

```
1: initialise C to CLOSURE({[S' → .S, $]});
2: repeat
3:     for each set of items I in C do
4:         for each grammar symbol X do
5:             if GOTO(I, X) is not empty and not in C then
6:                 add GOTO(I, X) to C;
7:             end if
8:         end for
9:     end for
10: until no new sets of items are added to C;
```

For example, consider the following augmented grammar:

$$S' \to S$$

$$S \to CC$$

$$C \to cC|d$$

We begin by computing the closure of $\{[S' \to .S, \$]\}$. To close, we match the item $[S' \to .S, \$]$ with the item $[A \to \alpha B\beta, a]$ in the procedure CLOSURE. That is, $A = S'$, $\alpha = \varepsilon$, B = S, $\beta = \varepsilon$ and a = \$. Function CLOSURE tells us to add $[B \to .\gamma, b]$ for each production $B \to \gamma$ and terminal b in FIRST($\beta\alpha$). In terms of the present grammar, $B \to \gamma$ must be $S \to CC$, and since β is ε and a is \$, b may only be \$. Thus we add $[S \to .CC, \$]$.
We continue to compute the closure by adding all items $[C \to .\gamma, b]$ for b in FIRST(C\$). That is, matching $[S \to .CC, \$]$ against $[A \to \alpha B\beta, a]$, wa have A = S, $\alpha = \varepsilon$, B = C, β = C and a = \$. Since C does not derive the empty string, FIRST(C\$) = FIRST(C). Since FIRST(C) contains terminals c and d, we add items $[C \to .cC, c]$, $[C \to .cC, d]$, $[C \to .c, d]$ and $[C \to .d, d]$. None of the items has a nonterminal immediately to the right of the dot, so we have completed our first set of LR(1) items, which is:

- I_0

  $S \to .S, \$$

  $S \to .CC, \$$

  $C \to .cC, c/d$

  $C \to .d, c/d$

Now we compute GOTO(I_0, S); we must close the item $[S^i \to S.\$]$. No additional closure is possible, since the dot is at the right end. Thus we have the next set of items

$$I_1 : S' \to S.\$$$

For GOTO(I_0, C), we close $[S \to C.C, \$]$. We add the C-productions with second component \$ and then can add no more, yelding

- I_2

  $S \to C.C, \$$

  $C \to .cC, \$$

  $C \to .d, \$$

For GOTO(I_0, c), we must close $\{[C \to c.C, c/d]\}$. We add the C-productions with second component c/d, yelding

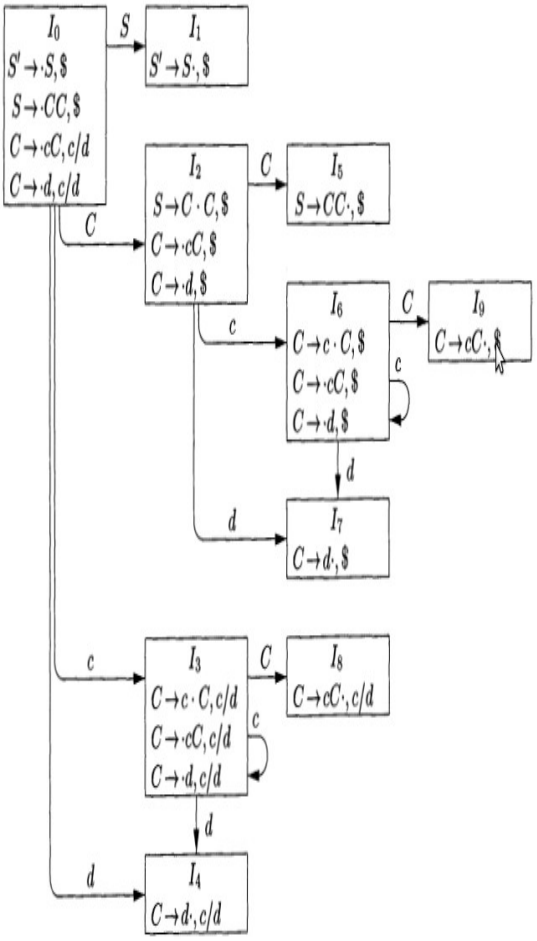
- I_3

  $C \to c.C, c/d$

  $C \to .CC, c/d$

  $C \to .d, c/d$

Finally, for GOTO(I_0, d) we get $I_4 = \{C \to d., c/d\}$.
Computing the other GOTO functions, we get the following automaton:



5.5.2 Canonical LR(1) Parsing Tables

Construction of canonical-LR parsing tables

1. Construct $C' = \{I_0, I_1, ..., I_n\}$, the collection of sets of LR(1) items for the augmented grammar G'

2. State i of the parser is constructed from I_i. The parsing action for state i is determined as follows.

(a) If $[A \to \alpha a \beta, b]$ is in I_i and GOTO(I_i, a) = I_j, then set ACTION[i, a] to "shift j". Here a must be a terminal.

(b) If $[A \to \alpha., a]$ is in I_i, $A \neq S'$, then set ACTION[i, a] to reduce $A \to \alpha''$.

(c) If $[S' \to S., \$]$ is in I_i; then set ACTION[i, $] to "accept".

If any conflicting actions result from the above rules, we say the grammar is not LR(1).

3. The goto transitions for state i are constructed for all nonterminals A using the rule: If GOTO(I_i, A) = I_j, then GOTO[i, a] = j.

4. All entries not defined by rules (2) and (3) are made "error".

5. The initial state of the parser is the one constructed from the set of items containing $[S' \to -S, \$]$.

Using this algorithm, the canonical parsing table for the grammar

$$S' \to S$$

$$S \to CC$$

$$C \to cC|d$$

is:

	c	**d**	$	**S**	**C**
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

5.6 Constructing LALR Parsing Tables

The LALR (lookahead LR) tecnique is often used in practice, because the tables obtained by it are considerably smaller than the canonical LR tables.

Generally, we can look for sets of LR(1) items having the same **core**, that is, set of first components, and we may merge these sets with common cores into one set of items.
For example, given the following sets of items:

I_3

$C \to c.C, c/d$

$C \to .cC, c/d$

$C \to .d, c/d$

I_4

$C \to .d, c/d$

I_6

$C \to c.C, \$$

$C \to .cC, \$$

$C \to .d, \$$

I_7

$C \to .d, \$$

Moreover, we know that GOTO(I_3, c) = I_3, GOTO(I_3, d) = I_4, GOTO(I_6, c) = I_6 and GOTO(I_6, d) = I_7.
In this case, I_4 and I_7 form such a pair, with core $\{C \to d.\}$. . Similarly, I_3 and I_6 form another pair, with core $\{C \to c.C, C \to .cC, C \to .d\}$. Note that, in general, a core is a set of LR(0) items for the grammar at hand, and that an LR(1) grammar may produce more than two sets of items with the same core.
Since the core of GOTO(I, X) depends only on the core of I, the goto's of merged sets can themselves be merged. Thus, there is no problem revising the goto function as we merge sets of items. Suppose we have an LR(1) grammar, that is, one whose sets of LR(1) items produce no parsing-action conflicts; if we replace all states having the same core with their union, it is possible that the resulting union will have a conflict (only "reduce/reduce"), but it is unlikely.

---

generation of LALR parsing table (space-consuming version)

---

1. Generate the collection C=$\{I_0, ..., I_n\}$ of the set of LR(1) items.

2. Find all the sets of items in C which have the same core and substitute them by their union; call the new collection $C' = \{J_0, ..., J_n\}$.

3. Generate the "action" part of the table in the same way as for cLR starting from C'.

If there is a conflict, G is not LALR and the and the items in C' are called LALR(1) items for G.

4. If $I_n \subset J$, then GOTO(J, X) = gk, where k is the union of the set of LR(1) items that have the same core as GOTO(I_n, X)

The table produce by the previous algorithm is called the *LALR parsing table* for G. If there are no parsing action conflicts, then the given grammar is saif to be an *LALR(1) grammar*. The collection of sets of items constructed in step (3) is called the *LALR(1) collection*.

Again, considering the grammar

$$S' \to S$$

$$S \to CC$$

$$C \to cC|d$$

and, after obtaining the collection of sets of items, finding that there are some pairs wich have the same core (I_3 and I_6, I_4 and I_7, I_8 and I_9), che would get the following LALR parsing table:

	c	**d**	$	**S**	**C**
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

To see how GOTO's are computed, consider GOTO(I_{36}, C). In the original set of LR(1) items, GOTO(I_3, C) = I_8, and I_8 is now part of I_{89}, so we make GOTO(I_{36},, C) be I_{89}. We could have arrived at the same conclusion if we

considered I_6, the other part of I_{36}. That is, GOTO(I_6, C) = I_9, and I_9 is now part of I_{89}.
When presented with a string from the language c^*dc^+d, both the LR parser and the LALR parser make exactly the same sequence of shifts and reductions, although the name of the states on the stack may differ. For instance, if the LR parser puts I_3 or I_6 on the stack, the LALR parser will put I_{36}.
When presented with erroneus input, the LALR parser may proceed to do some reductions after the LR parser has declared an error.

Efficient Construction of LALR Parsing Table

There are several modification that can be made to the previous algorithm to avoid constructing the full collection of sets of LR(1) items in the process of creating an LALR(1) parsing table:

- First, we can represent any set of LR(0) or LR(1) items I by its kernel, that is, by those items that are either the initial item or that have the dot somewhere other than at the beginning of the production body.
- We can construct the LALR(1)-item kernels from the LR(0)-item kernels, by a process of propagation and spontaneous generation of lookaheads.