PRESSES UNIVERSITAIRES DE BRUXELLES

COURS-LIBRAIRIE

ULB

UNIVERSITÉ LIBRE DE BRUXELLES, UNIVERSITÉ D'EUROPE

Introduction to Language Theory and Compilation

Part 1

Thierry MASSART

D/2011/0098/224
4e édition – Tirage 2011-12/1-S
INFO-F-403_A

« L'accès au savoir n'est plus le seul fait des scientifiques. La connaissance devient le bien sans cesse grandissant d'un nombre croissant d'individus : des individus plus humains, conscients des possibilités de la science contemporaine, exigeant sans cesse plus fermement de pouvoir en bénéficier. »

Willy Peers (1924-1984)
Gynécologue (ULB, 1956), militant pour l'accouchement sans douleur et la législation de l'avortement.

Le label FSC : la garantie d'une gestion responsable des forêts
Les Presses Universitaires de Bruxelles s'engagent !

Les P.U.B. impriment depuis de nombreuses années les syllabus sur du papier recyclé. Les différences de qualité constatées au niveau des papiers recyclés ont cependant poussé les P.U.B. à se tourner vers un papier de meilleure qualité et surtout porteur du label FSC.

Sensibles aux objectifs du FSC et soucieuses d'adopter une démarche responsable, les P.U.B. se sont conformé aux exigences du FSC et ont obtenu en avril 2010 la certification FSC (n° de certificat COC spécifique aux P.U.B. : CU-COC-809718-HA).

Seule l'obtention de ce certificat autorise les P.U.B. à utiliser le label FSC selon des règles strictes. Fortes de leur engagement en faveur de la gestion durable des forêts, les P.U.B. souhaitent dorénavant imprimer tous les syllabus sur du papier certifié FSC. Le label FSC repris sur les syllabus vous en donnera la garantie.

Qu'est-ce que le FSC ?
FSC signifie "Forest Stewardship Council" ou "Conseil de bonne gestion forestière". Il s'agit d'une organisation internationale, non gouvernementale, à but non lucratif qui a pour mission de promouvoir dans le monde une gestion responsable et durable des forêts.
Se basant sur dix principes et critères généraux, le FSC veille à travers la certification des forêts au respect des exigences sociales, écologiques et économiques très poussées sur le plan de la gestion forestière.

Quelles garanties ?
Le système FSC repose également sur la traçabilité du produit depuis la forêt certifiée dont il est issu jusqu'au consommateur final. Cette traçabilité est assurée par le contrôle de chaque maillon de la chaîne de commercialisation/transformation du produit (Chaîne de Contrôle : Chain of Custody – COC). Dans le cas du papier et afin de garantir cette traçabilité, aussi bien le producteur de pâte à papier que le fabricant de papier, le grossiste et l'imprimeur doivent être contrôlés.
Ces contrôles sont effectués par des organismes de certification indépendants.

Les 10 principes et critères du FSC

1. L'aménagement forestier doit respecter les lois nationales, les traités internationaux et les principes et critères du FSC.
2. La sécurité foncière et les droits d'usage à long terme sur les terres et les ressources forestières doivent être clairement définis, documentés et légalement établis.
3. Les droits légaux et coutumiers des peuples indigènes à la propriété, à l'usage et à la gestion de leurs territoires et de leurs ressources doivent être reconnus et respectés.
4. La gestion forestière doit maintenir ou améliorer le bien-être social et économique à long terme des travailleurs forestiers et des communautés locales.
5. La gestion forestière doit encourager l'utilisation efficace des multiples produits et services de la forêt pour en garantir la viabilité économique ainsi qu'une large variété de prestations environnementales et sociales.
6. Les fonctions écologiques et la diversité biologique de la forêt doivent être protégées.
7. Un plan d'aménagement doit être écrit et mis en œuvre. Il doit clairement indiquer les objectifs poursuivis et les moyens d'y parvenir.
8. Un suivi doit être effectué afin d'évaluer les impacts de la gestion forestière.
9. Les forêts à haute valeur pour la conservation doivent être maintenues (par ex : les forêts dont la richesse biologique est exceptionnelle ou qui présentent un intérêt culturel ou religieux important). La gestion de ces forêts doit toujours être fondée sur un principe de précaution.
10. Les plantations doivent compléter les forêts naturelles, mais ne peuvent pas les remplacer. Elles doivent réduire la pression exercée sur les forêts naturelles et promouvoir leur restauration et leur conservation. Les principes de 1 à 9 s'appliquent également aux plantations.

Le label FSC apposé sur des produits en papier ou en bois apporte la garantie que ceux-ci proviennent de forêts gérées selon les principes et critères FSC.

® FSC A.C. FSC-SECR-0045

FSC, le label du bois et du papier responsable

Plus d'informations ?
www.fsc.be
A la recherche de produits FSC ?
www.jecherchedufsc.be

Introduction to Language Theory and Compilation

Thierry Massart
Université Libre de Bruxelles
Département d'Informatique

September 2011

Acknowledgements

I would like to thank
Gilles Geeraerts, Sébastien Collette, Camille Constant, Thomas De Schampheleire and Markus Lindström
for their valuable comments on this syllabus

Thierry Massart

Chapters of the course

Main references

- J. E. Hopcroft, R. Motwani, and J. D. Ullman; *Introduction to Automata Theory, Languages, and Computation*, Second Edition, Addison-Wesley, New York, 2001.
- Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

Other references:

- John R. Levine, Tony Mason, Doug Brown, *Lex & Yacc*, O'Reilly ed, 1992.
- Reinhard Wilhelm, Dieter Maurer, *Compiler Design*, Addison-Wesley, 1995. (P-machine reference)
- Pierre Wolper, *Introduction à la Calculabilité*, InterEditions, 1991.
- Thierry Massart, *Théorie des langages et de la compilation*, Presses Universitaires de Bruxelles or (http://www.ulb.ac.be/di/verif/tmassart/Compil/Syllabus.pdf), 2007.

Chapter 1: Introduction

Outline

What are you going to learn in this course?

1. Reminder on
   - how to formally define a model to describe
     - a language (programming or other)
     - a (computer) system
   - How to deduce properties on this model
2. What is
   - a compiler?
   - a tool for data processing?
3. The notion of metatool = tool to build other tools
   Example: generator of (part of) a compiler
4. How to build a compiler or a tool for data processing
   - through hard coding
   - through the use of tools

Approach

Show the scientific and engineering approach, i.e.

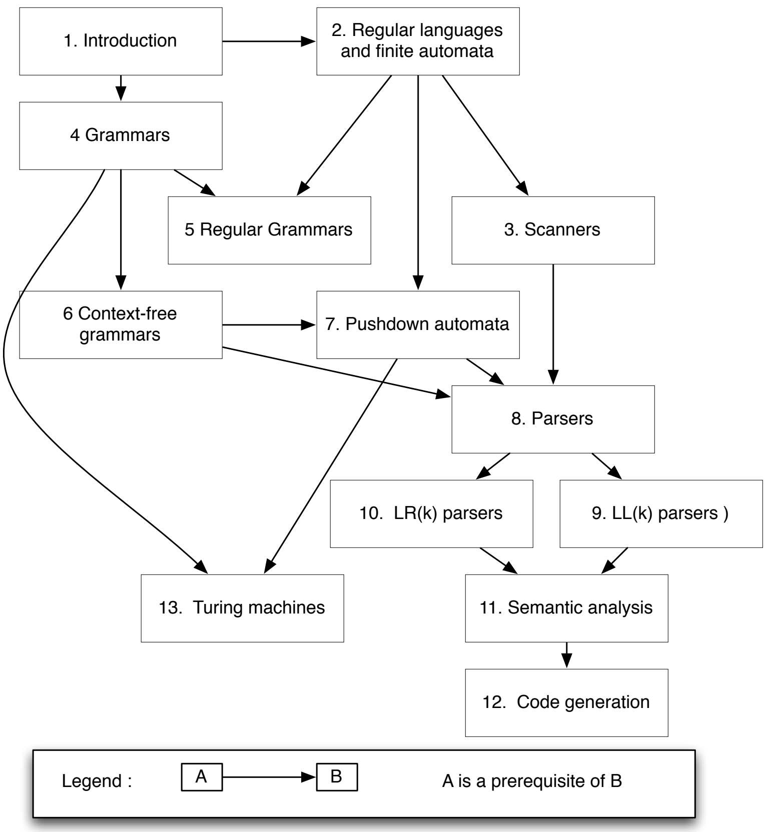
1. Understanding the (mathematical / informatical) tools available to solve the problem
2. Learning to use these tools
3. Designing a system using these tools
4. Implementing this system

The tools used here are

- formalisms to define a language or model a system
- generators of parts of compilers

Outline

Order of the chapters



Examples of reading sequences

- Everything: 1-13 in sequence
- Parts pertaining to “compilers”: 1-2-3-4-6-7-8-9-10-11-12
- Parts pertaining to “language theory”: 1-2-4-5-6-7-13

Outline

The world of language theory

Goal of language theory

Formally understand and process languages as a way to communicate.

Definition (Language - word (string))

- *A language is a set of words.*
- *A word (or token or string) is a sequence of symbols in a given alphabet.*

Alphabets, words, languages

Example (alphabets, words and languages)

Alphabet	Words	Languages
Σ ={0, 1}	ϵ,0,1,00, 01	{00,01, 1, 0, ϵ }, {ϵ}, $\emptyset$
{a, b, c, . . . , z}	bonjour, ca, va	{bonjour, ca, va, ϵ}
{“héron”, “petit”, “pas”}	“héron” “petit” “pas”	{ϵ, “héron” “petit” “pas”}
$\{\alpha, \beta, \gamma, \delta, \mu, \nu, \pi, \sigma, \tau\}$	$\tau\alpha\gamma\alpha\delta\alpha$	{ϵ,$\tau\alpha\gamma\alpha\delta\alpha$}
{0, 1}	ϵ,01,10	{ϵ,01,10, 0011, 0101, . . . }

Notations

We usually use the standard notations:

- Alphabet: Σ (example: $\Sigma = \{0, 1\}$)
- Words: $x, y, z, \ldots$ (example: $x = 0011$)
- Languages: $L, L_1, \ldots$ (example: $L = \{\epsilon, 00, 11\}$)

Aims of the course
Order of the chapters
What is language theory?
What is a compiler?
Compilation phases
Some reminders and mathematical notions

The world of language theory (cont’d)

Studied

- The notion of (formal) grammar which defines (the syntax of) a language,
- The notion of automaton which allows us to determine if a word belongs to a language (and therefore to define a language as the set of recognized words),
- The notion of regular expression which allows us to denote a language.

Motivations and applications

Practical applications of language theory

- formal definition of syntax and semantics of (programming) languages,
- compiler design,
- abstract modelling of systems (computers, electronics, biological systems, ...)

Theoretical motivations

Related to:

- computability theory (which determines in particular which problems are solvable by a computer)
- complexity theory which studies (mainly time and space) resources needed to solve a problem

Outline

Definition (Compiler)

A compiler is a computer program which is a translator $C^{L_C}_{L_S \to L_O}$ with

1. L_C *the language used to write the compiler itself*
2. L_S *the source language to compile*
3. L_O *the target language*

Example (for $C^{L_C}_{L_S \to L_O}$)

L_C	L_S	L_O
C	RISC Assembler	RISC Assembler
C	C	P7 Assembler
C	Java	C
Java	$\LaTeX$	HTML
C	XML	PDF

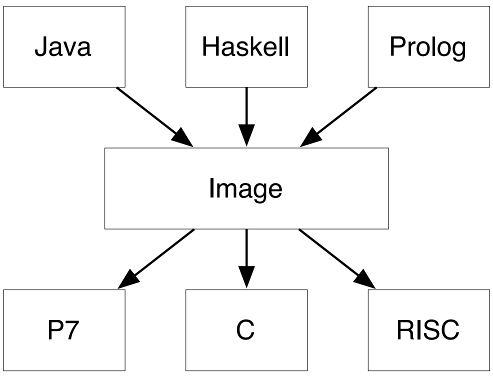
If $L_C = L_S$: bootstrapping can be needed to compile $C^{L_C}_{L_C \to L_O}$

General structure of a compiler

Usually an intermediate language L_I is used.
The compiler is composed of a:

- front-end $L_S \to L_I$
- back-end $L_I \to L_O$

Eases the building of new compilers.



Features of compilers

- Efficiency
- Robustness
- Portability
- Reliability
- Debuggable code

- Single pass
- *n* passes (70 for a PL/I compiler!)
- Optimizing
- Native
- Cross-compilation

Compiler vs. Interpreter

Interpreter = tool that does analysis, translation, but also execution of a program written in a computer language.
An interpreter handles execution *during* interpretation.

Outline

A small program to compile

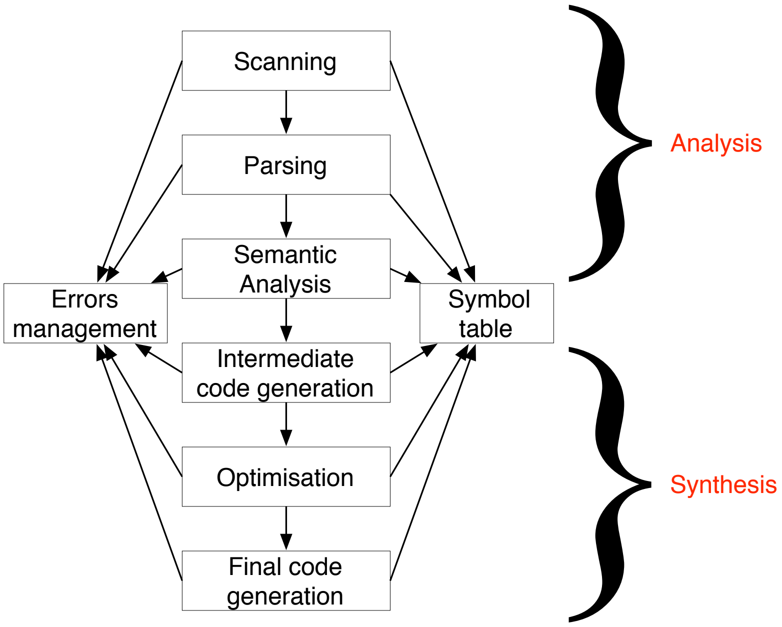
Example (of a C++ program to compile)

```
int main()
// Collatz Conjecture
// Hypothesis : N > 0
{
  long int N;

  cout << "Enter A Number : ";
  cin >> N;
  while (N != 1)
  {
    if (N%2 == 0)
      N = N/2;
    else
      N = 3*N+1;
  }
  cout << N << endl;  //Print 1
}
```


Aims of the course
Order of the chapters
What is language theory?
What is a compiler?
Compilation phases
Some reminders and mathematical notions

6 phases for the compilation: 3 analysis phases - 3 synthesis phases



Compilation steps

Compilation is cut into 2 steps

1. Analysis decomposes and identifies the elements and relationships of the source program and builds its image (structured representation of the program with its relations),
2. Synthesis builds, from the *image*, a program in the target language

Contents of the symbol table

One entry for each identifier of the program to compile: contains its attributes values to describe the identifier.

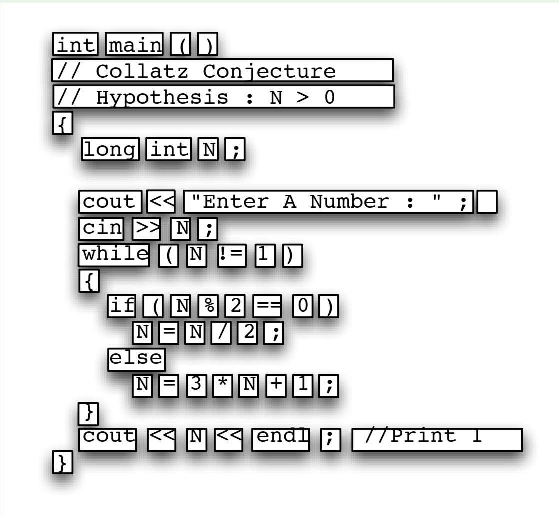
Remark

In case an error occurs, the compiler can try to resynchronize to possibly report other errors instead of halting immediately.

Lexical analysis (scanning)

- A program can be seen as a "sentence"; the main role of lexical analysis is to identify the "words" of that sentence.
- The scanner decomposes the program into tokens by identifying the lexical units of each token.

Example (of decomposition into tokens)



Lexical analysis (scanning)

Definition (Lexical Unit (or type of token))

Generic type of lexical elements (corresponds to a set of strings with a common semantic).
Example: identifier, relational operator, "begin" keyword...

Definition (Token (or string))

Instance of a lexical unit.
Example: `N` *is a token of the* identifier *lexical unit*

Definition (Pattern)

Rule which describes a lexical unit
Generally a pattern is given by a regular expression *(see below)*

Relation between token, lexical unit and pattern

lexical unit = { token | pattern(token) }

Introductory examples for regular expressions

Operators on regular expressions:

- . : concatenation (generally omitted)
- + : union
- * : repetition (0,1,2, ... times) = (Kleene closure (pronounced Klayni!))

Example (Some regular expressions)

- digit = 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9
- nat-nb = digit digit*
- operator = << + != + == + ...
- open-par = (
- close-par =)
- letter = a + b + ... + z
- identifier = letter (letter + digit)*

Scanning result

Example (of lexical units and tokens)

lexical unit	token
identifier	int
identifier	main
open-par	(
close-par	)
...	

Other aims of the scanning phase

Other aims of the scanning phase

- (Possibly) put the (non predefined) identifiers and literals in the symbol table [a]
- Produce the listing / link with clever editor (IDE)
- Clean the source code of the source program (suppress comments, spaces, tabulations, ...)

[a] can be done in a latter analysis phase

Syntactic analysis (parsing)

- The main role of the syntactic analysis is to find the structure of the "sentence" (the program): i.e. to build an image of the syntactic structure of the program that is internal to the compiler and that can also be easily manipulated.
- The parser builds a syntactic tree (or parse tree) corresponding to the code.

The set of possible syntactic trees for a program is defined by a (context-free) grammar.

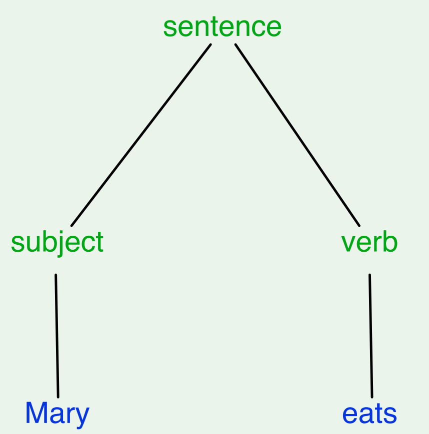
Grammar example (1)

Example (Grammar of a sentence)

- sentence = subject verb
- subject = **"John"** | **"Mary"**
- verb = **"eats"** | **"speaks"**

can provide

- **John eats**
- **John speaks**
- **Mary eats**
- **Mary speaks**



Syntactic tree of the sentence **Mary eats**

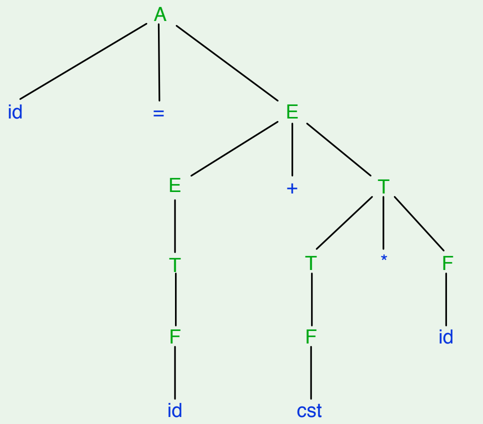
Grammar example (2)

Example (Grammar of an expression)

- A = “**id**” “**=**” E
- E = T | E “**+**” T
- T = F | T “*****” F
- F = “**id**” | “**cst**” | “**(**” E “**)**”

can give:

- **id = id**
- **id = id + cst * id**
- . . .



Syntactic tree of the sentence
id = id + cst * id

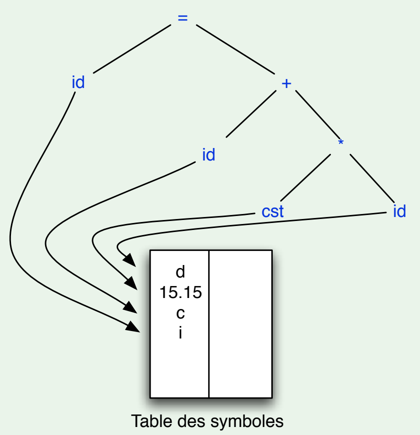
Grammar example (2 cont’d)

Example (Grammar of an expression)

- A = “**id**” “**=**” E
- E = T | E “**+**” T
- T = F | T “*****” F
- F = “**id**” | “**cst**” | “**(**” E “**)**”

can give:

- **id = id**
- **id = id + cst * id**
- . . .



Abstract syntax tree
with references to the symbol table
for the sentence **i = c + 15.15 * d**

Semantic analysis

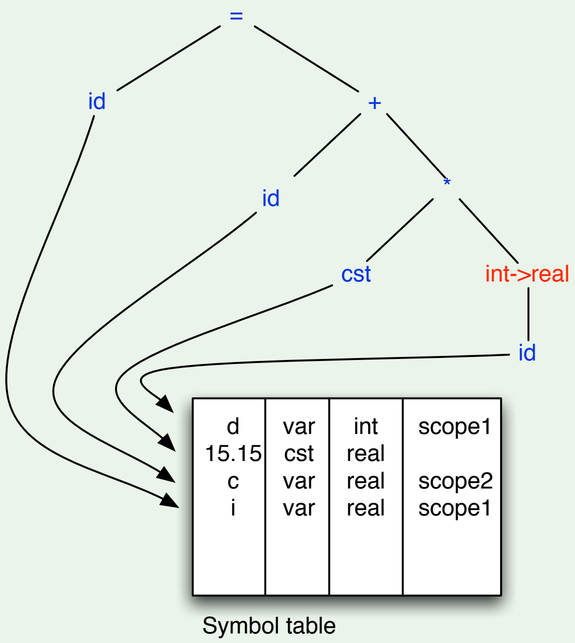
Roles of semantic analysis

For an imperative language, semantic analysis (also called context management) takes care of the *non local* relations; it also takes care of:

1. visibility control and the link between definition and use of identifiers (with the construction and use of the symbol table)
2. type control of the “objects”, number and types of the parameters of the functions
3. flow control (verify for instance that a goto is allowed - see example below)
4. construction of a completed abstract syntax tree with type information and a flow control graph to prepare the synthesis step.

Example of result of the semantic analysis

Example (for the expression i = c + 15.15 * d)



Modified abstract syntax tree
with references to the symbol table
for the sentence **i = c + 15.15 * d**

Synthesis

Synthesis steps

For an imperative language, synthesis is usually made through 3 phases:

1. Intermediate code generation in an intermediate language which
   - uses symbolic addressing
   - uses standard operations
   - does memory allocation (results in temporary variables ...)
2. Code optimisation
   - suppresses “dead” code
   - puts some instructions outside loops
   - suppresses some instructions and optimizes memory access
3. Production of the final code
   - Physical memory allocation
   - CPU register management

Synthesis example

Example (for the code i = c + 15.15 * d)

1. Intermediate code generation

```
temp1 <- 15.15
temp2 <- Int2Real(id3)
temp2 <- temp1 * temp2
temp3 <- id2
temp3 <- temp3 + temp2
id1   <- temp3
```

2. Code optimization

```
temp1 <- Int2Real(id3)
temp1 <- 15.15 * temp1
id1 <- id2 + temp1
```

3. Final code production

```
MOVF  id3,R1
ITOR  R1
MULF  15.15,R1,R1
ADDF  id2,R1,R1
STO   R1,id1
```

Outline

Used notations

- Σ : Language alphabet
- x, y, z, t, x_i (letter at the end of the alphabet) : symbolises strings of Σ (example $x = abba$)
- ϵ : empty word
- $|x|$: length of the string x ($|\epsilon| = 0$, $|abba| = 4$)
- $a^i = aa...a$ (string composed of i times the character a)
- $x^i = xx...x$ (string composed of i times the string x)
- L,L', L_i A, B: languages

Operations on strings

- concatenation: ex: `lent.gage` = `lentgage`
  - $\epsilon w = w = w\epsilon$
- w^R : mirror image of w (ex: $abbd^R = dbba$)
- prefix of w. E.g. if w=`abbc`
  - the prefixes are: ϵ, `a`, `ab`, `abb`, `abbc`
  - the proper prefixes are ϵ, `a`, `ab`, `abb`
- suffix of w. E.g. if w=`abbc`
  - the suffixes are: ϵ, `c`, `bc`, `bbc`, `abbc`
  - the proper suffixes are ϵ, `c`, `bc`, `bbc`

Operations on languages

Definition (Language on the alphabet Σ)

Set of strings on this alphabet

Operations on languages are therefore operations on sets

- $\cup, \cap, \setminus, A \times B, 2^A$
- concatenation or product: ex: $L_1.L_2 = \{xy | x \in L_1 \land y \in L_2\}$
  - $L^0 \stackrel{def}{=} \{\epsilon\}$
  - $L^i = L^{i-1}.L$
- Kleene closure: $L^* \stackrel{def}{=} \bigcup_{i \in \mathbb{N}} L^i$
- Positive closure: $L^+ \stackrel{def}{=} \bigcup_{i \in \mathbb{N} \setminus \{0\}} L^i$
- Complement: $\bar{L} = \{w | w \in \Sigma^* \ \land \ w \notin L\}$

Relations

Definition (Equivalence)

A relation that is:

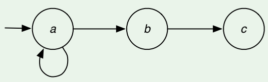
- *reflexive* ($\forall x : xRx$)
- *symmetrical* ($\forall x, y : xRy \rightarrow yRx$)
- *transitive* ($\forall x, y, z : xRy \wedge yRz \rightarrow xRz$)

Definition (Closure of relations)

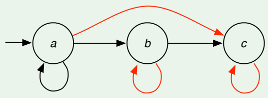
Given $\mathcal{P}$ *a set of properties of a relation* $\mathbf{R}$, *the* $\mathcal{P}$*-closure of* $\mathbf{R}$ *is the smallest relation* $\mathbf{R}'$ *which includes* $\mathbf{R}$ *and has the properties* $\mathcal{P}$

Example (of reflexo-transitive closure)

The transitive closure $\mathbf{R}^+$, the reflexo-transitive closure $\mathbf{R}^*$



gives for $\mathbf{R}^*$



Closure property of a class of languages

Definition (Closure of a class of languages)

A class of languages $\mathcal{C}$ *is closed for an operation op, if the language resulting from this operation on any language(s) of* $\mathcal{C}$ *remains in this class of languages* $\mathcal{C}$.

Example: *suppose op is a binary operator*

$$\mathcal{C} \text{ is closed for } op$$

$$\textit{iff}$$

$$\forall L_1, L_2 \in \mathcal{C} \Rightarrow L_1 \; op \; L_2 \in \mathcal{C}$$

Cardinality

Definition (Same cardinality)

Two sets have the same cardinality if there exists a bijection between both of them.

- $\aleph_0$ *denotes the cardinality of the countably infinite sets (such as* $\mathbb{N}$*)*
- $\aleph_1$ *denotes the cardinality of the uncountably infinite sets (such as* $\mathbb{R}$*)*

We assume that uncountably infinite sets are continuous.

Cardinality of Σ^* and 2^{Σ^*}

Given a finite non empty alphabet Σ,

- Σ^*: the set of strings of Σ, is countably infinite
- $\mathcal{P}(\Sigma^*)$ denoted also 2^{Σ^*}: the set of languages from Σ, is uncountably infinite

Chapter 2: Regular languages and finite automata

PUB Cours-Librairie, av. P. Héger 42, B-1000 Bruxelles / INFO-F-403_A

Outline

1. Regular languages and regular expressions
2. Finite state automata
3. Equivalence between FA and RE
4. Other types of automata
5. Some properties of regular languages

Introduction

Motivation

- Regular expressions allow us to easily denote regular languages
- For instance, UNIX-like systems intensively use extended regular expressions in their shells
- They are also used to define lexical units of a programming language

Definition of regular languages

Preliminary remark

- Every finite language can be enumerated (even if it can take very long)
- For infinite languages, an exhaustive enumeration is not possible
- The class of regular languages (defined below) includes *all* finite languages and *some* infinite ones

Definition (class of regular languages)

The set $\mathcal{L}$ of regular languages on an alphabet Σ is the smallest set which satisfies:

1. $\emptyset \in \mathcal{L}$
2. $\{\epsilon\} \in \mathcal{L}$
3. $\forall a \in \Sigma, \{a\} \in \mathcal{L}$
4. *if $A, B \in \mathcal{L}$ then $A \cup B, A.B, A^* \in \mathcal{L}$*

Notation of regular languages

Definition (set of regular expressions (RE))

The set of regular expressions (RE) on an alphabet Σ is the smallest set which includes:

1. $\emptyset$ *: denotes the empty set,*
2. ϵ *: denotes the set $\{\epsilon\}$,*
3. $\forall a \in \Sigma$, **a***: denotes the set $\{a\}$,*
4. *with r and s which resp. denote R and S:*
   $r + s$, rs and r^* resp. denote $R \cup S$, R.S and R^*

We suppose $^* < . < +$ and add () if needed

Example (of regular expressions)

- $\mathbf{00}$
- $(\mathbf{0} + \mathbf{1})^*$
- $(\mathbf{0} + \mathbf{1})^*\mathbf{00}(\mathbf{0} + \mathbf{1})^*$
- $\mathbf{0^4 1 0^4}$ notation for **000010000**
- $(\mathbf{01})^* + (\mathbf{10})^* + \mathbf{0}(\mathbf{10})^* + \mathbf{1}(\mathbf{01})^*$
- $(\epsilon + \mathbf{1})(\mathbf{01})^*(\epsilon + \mathbf{0})$

Properties of regular languages

Properties of ϵ and $\emptyset$

- $\epsilon w = w = w\epsilon$
- $\emptyset w = \emptyset = w\emptyset$
- $\emptyset + r = r = r + \emptyset$
- $\epsilon^* = \epsilon$
- $\emptyset^* = \epsilon$
- $(\epsilon + r)^* = r^*$

Outline

Automata

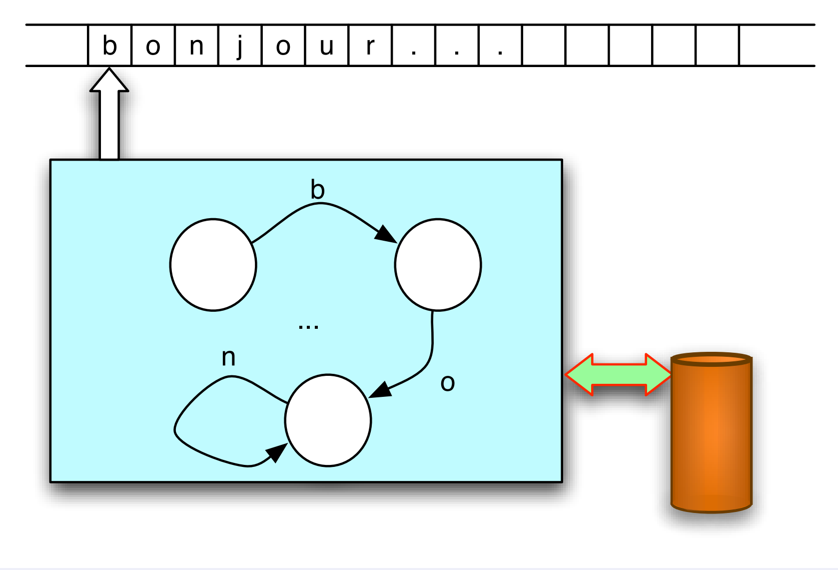
Informal presentation

An automaton *M* is a mathematical model of a system with discrete input and output.
It generally contains

- control states (at any time, *M* is in one of these states)
- a data tape which contains symbols
- a (read/write) head
- a memory

Automata

Informal presentation



Example: an e-commerce protocol with e-money

Example (Possible events)

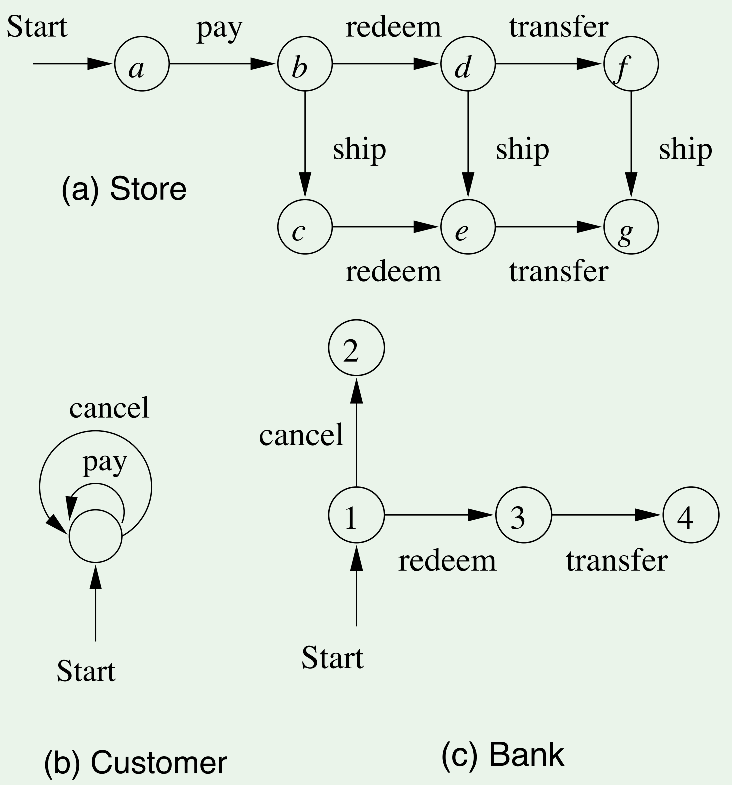
1. pay: the customer pays the shop
2. cancel: the customer stops the transaction
3. ship: the shop sends the goods
4. redeem: the shop asks for money from the bank
5. transfer: the bank transfers money to the shop

Remark

The example is formalized with finite automata (see below)

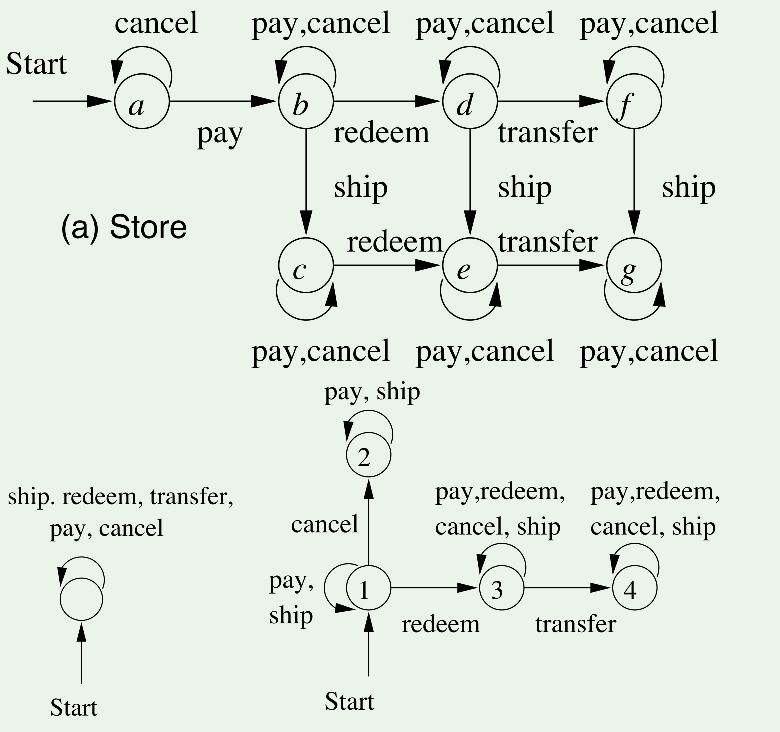
Example: an e-commerce protocol with e-money (2)

Example (Protocol for each participant)



Example: an e-commerce protocol with e-money (2)

Example (Complete protocol)



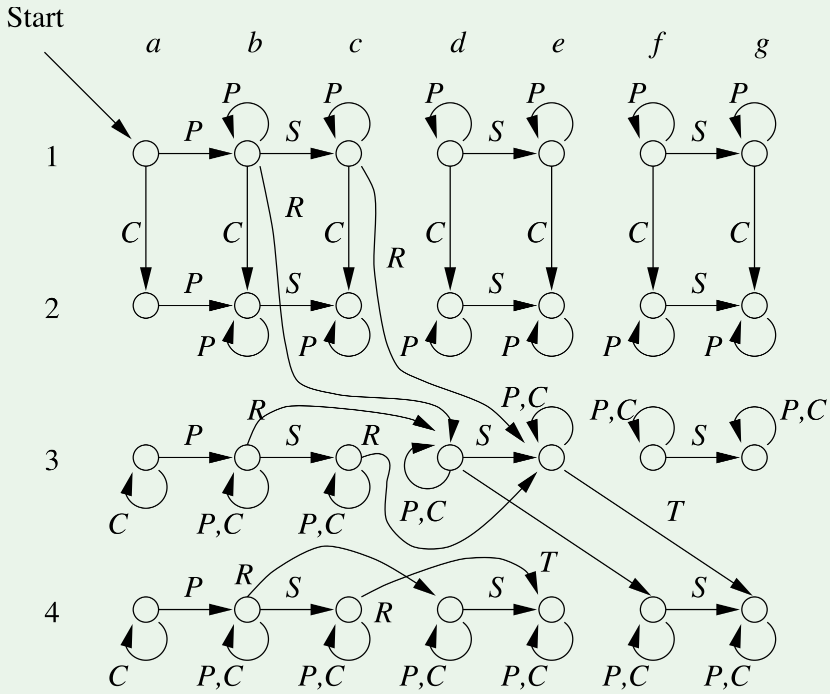
(a) Store

(b) Customer

(c) Bank

Example: an e-commerce protocol with e-money (2)

Example (complete system)



Finite Automaton (FA)

Remark

Finite automata are used in this course as a formalism to define sets of strings of a language

Restrictions of finite automata

A finite automaton (FA):

- has no memory
- can only read on the tape (input)
- The reading head can only go from left to right

3 kinds of FA exist

- Deterministic finite automata (DFA)
- Nondeterministic finite automata (NFA)
- Nondeterministic finite automata with epsilon transitions (ϵ-NFA)
  -> an ϵ symbol is added to denote these transitions

Finite automaton: formal definition

Definition (Finite automaton)

$$M = \langle Q, \Sigma, \delta, q_0, F \rangle$$

with

1. *Q: a finite set of states*
2. Σ *: alphabet (allowed symbols)*
3. δ*: transition function*
4. q_0 *: initial state*
5. $F \subseteq Q$ *: set of accepting states*

δ *is defined for*

- *M DFA :* $\delta : Q \times \Sigma \rightarrow Q$
- *M NFA :* $\delta : Q \times \Sigma \rightarrow 2^Q$
- *M* ϵ*-NFA :* $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$

Examples of finite automata

Example

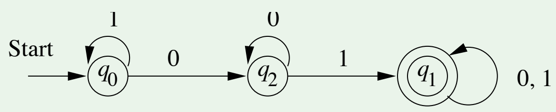
A deterministic automaton A which accepts $L = \{x01y : x, y \in \{0,1\}^*\}$

$$A = \langle \{q_0, q_1, q_2\}, \{0,1\}, \delta, q_0, \{q_1\} \rangle$$

with transition function δ :

	0	1
$\rightarrow q_0$	q_2	q_0
$\star q_1$	q_1	q_1
q_2	q_2	q_1

Graphical representation (transition diagram with labelled transitions):



Accepted strings

A string $w = a_1 a_2 \dots a_n$ is accepted by the FA if there exists a path in the transition diagram which starts at the initial state, terminates in an accepting state and has a sequence of labels $a_1 a_2 \dots a_n$

Configuration and accepted language

Definition (Configuration of a FA)

Couple $\langle q, w \rangle \in Q \times \Sigma^*$

- *Initial configuration :* $\langle q_0, w \rangle$ *where* w *is the string to accept*
- *Final (accepting) configuration :* $\langle q, \epsilon \rangle$ *with* $q \in F$

Definition (Configuration change)

$\langle q, aw \rangle \underset{M}{\vdash} \langle q', w \rangle$ *if*

- $\delta(q, a) = q'$ *for a DFA*
- $q' \in \delta(q, a)$ *for an NFA*
- $q' \in \delta(q, a)$ *for an* ϵ*-NFA with* $a \in \Sigma \cup \{\epsilon\}$

Language of M: $L(M)$

Definition ($L(M)$)

$$L(M) = \{w \mid w \in \Sigma^* \ \wedge\ \exists q \in F \,.\, \langle q_0, w\rangle \overset{*}{\underset{M}{\vdash}} \langle q, \epsilon\rangle\}$$

where

$\overset{*}{\underset{M}{\vdash}}$ *is the reflexo-transitive closure of* $\underset{M}{\vdash}$

Definition (Equivalence of automata)

M and M' are equivalent if they define the same language ($L(M) = L(M')$)

Example of DFA

Example

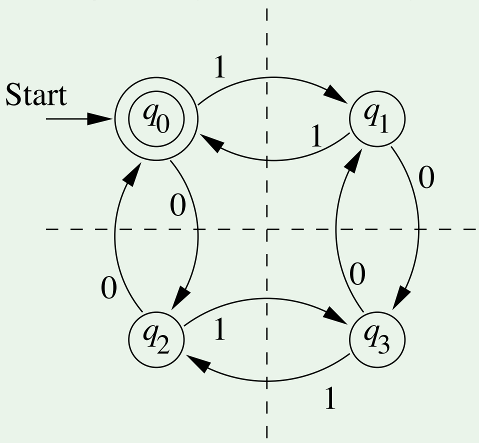
The DFA M accepts the set of strings on the alphabet $\{0, 1\}$ with an even number of 0 and 1.

$$M = \langle\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_0\}\rangle$$

with δ

	0	1
$\star \rightarrow q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Corresponding transition diagram:



Example of NFA

Example

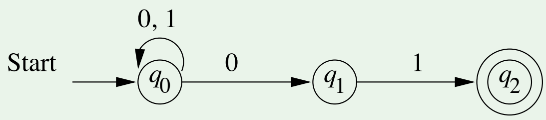
The NFA M accepts the set of strings on the alphabet $\{0, 1\}$ which end with 01.

$$M = \langle \{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\} \rangle$$

with δ

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$\star q_2$	$\emptyset$	$\emptyset$

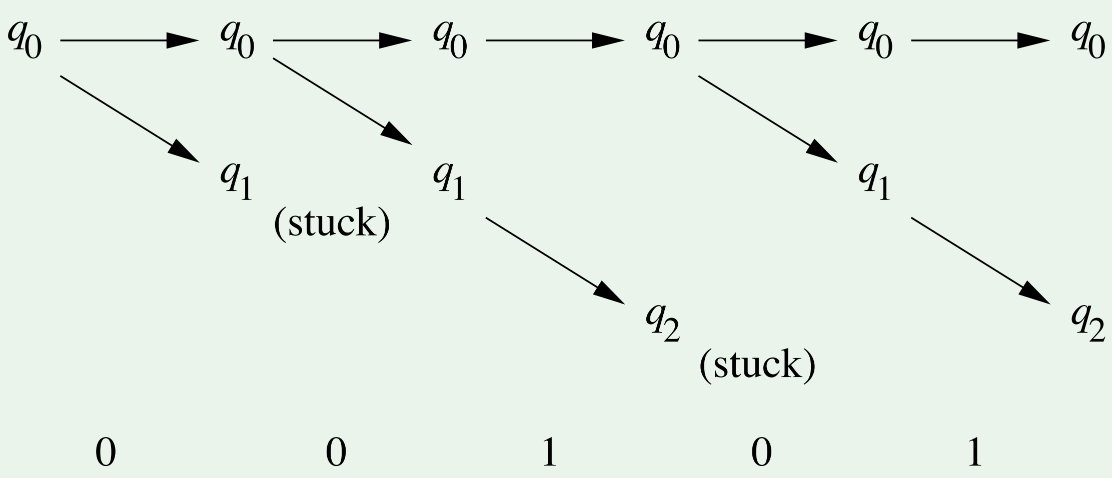
Corresponding transition diagram:



Example of NFA (cont'd)

Example

For the string 00101 the possible paths are :



Example of ϵ-NFA

Example

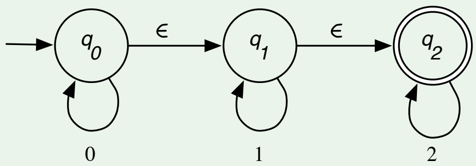
The ϵ-NFA M accepts the set of strings on the alphabet $\{0, 1, 2\}$ corresponding to the regular expression $0^*1^*2^*$.

$$M = \langle \{q_0, q_1, q_2\}, \{0, 1, 2\}, \delta, q_0, \{q_2\} \rangle$$

with δ

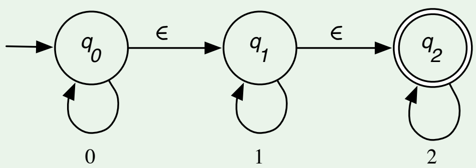
	0	1	2	ϵ
$\rightarrow q_0$	$\{q_0\}$	$\emptyset$	$\emptyset$	$\{q_1\}$
q_1	$\emptyset$	$\{q_1\}$	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$	$\{q_2\}$	$\emptyset$

Corresponding transition diagram:

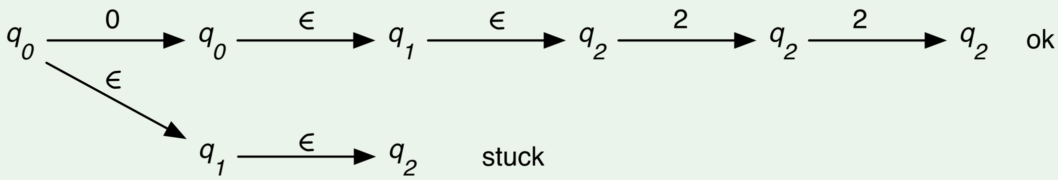


Example of ϵ-NFA (cont'd)

Example



For the string 022, the possible paths are :



Constructive definition of $L(M)$

Definition ($\hat{\delta}$: Extension of the transition function)

If one defines for a set of states S : $\delta(S,a) = \bigcup_{p \in S} \delta(p,a)$

For DFA: $\hat{\delta} : Q \times \Sigma^* \to Q$ • basis: $\hat{\delta}(q,\epsilon) = q$ • ind.: $\hat{\delta}(q,xa) = \delta(\hat{\delta}(q,x),a)$	$L(M) = \{w \mid \hat{\delta}(q_0,w) \in F\}$
For NFA: $\hat{\delta} : Q \times \Sigma^* \to 2^Q$ • basis: $\hat{\delta}(q,\epsilon) = \{q\}$ • ind.: $\hat{\delta}(q,xa) = \delta(\hat{\delta}(q,x),a)$	$L(M) = \{w \mid \hat{\delta}(q_0,w) \cap F \neq \emptyset\}$
For ϵ-NFA: $\hat{\delta} : Q \times \Sigma^* \to 2^Q$ • basis: $\hat{\delta}(q,\epsilon) = eclose(q)$ • ind.: $\hat{\delta}(q,xa) = eclose(\delta(\hat{\delta}(q,x),a))$ with $eclose(q) = \bigcup_{i \in \mathbb{N}} eclose^i(q)$ • $eclose^0(q) = \{q\}$ • $eclose^{i+1}(q) = \delta(eclose^i(q),\epsilon)$	$L(M) = \{w \mid \hat{\delta}(q_0,w) \cap F \neq \emptyset\}$

Outline

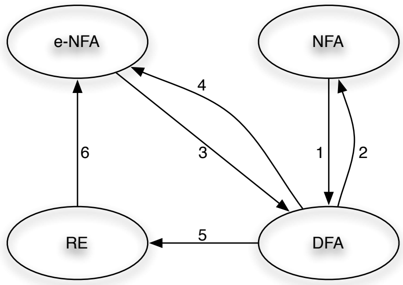
Equivalences between finite automata (FA) and regular expressions (RE)

For every

- DFA
- NFA
- ϵ-NFA
- RE

it is possible to translate it into the other formalisms.

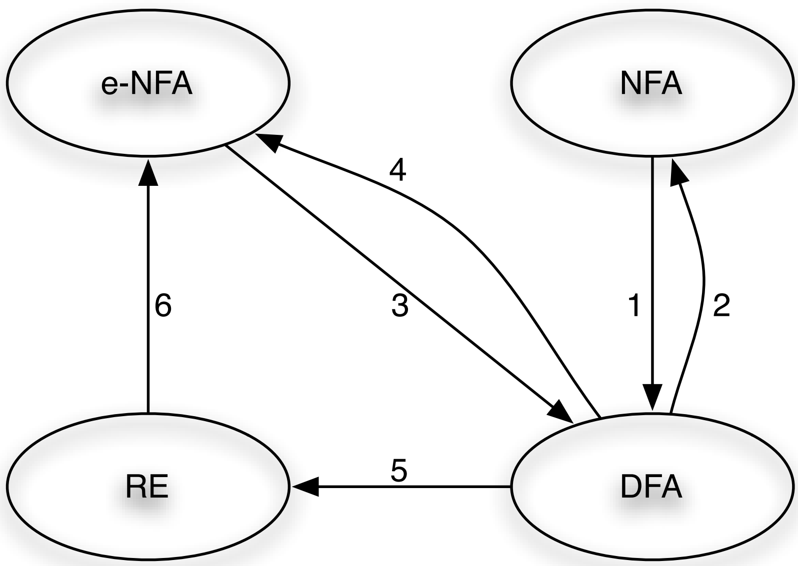
$\Rightarrow$ The 4 formalisms are equivalent and define the same class of languages: the regular languages



Arrows 2 and 4: straightforward

$\mathcal{C}$(NFA) $\subseteq$ $\mathcal{C}$(DFA)

Arrow 1:



Equivalence between DFA and NFA

- Defining an NFA suppresses the determinism constraint
- but we show that from every NFA N one can build an equivalent DFA D (i.e. $L(D) = L(N)$) and vice versa.
- the technique used is called subset construction: each state in D corresponds to a *subset* of states in N

$\mathcal{C}$(NFA) $\subseteq$ $\mathcal{C}$(DFA)

Theorem (For each NFA N, there exists a DFA D with $L(N) = L(D)$)

Proof:
Given an NFA N:

$$N = \langle Q_N, \Sigma, \delta_N, q_0, F_N \rangle$$

let us define (build) the DFA D:

$$D = \langle Q_D, \Sigma, \delta_D, \{q_0\}, F_D \rangle$$

with

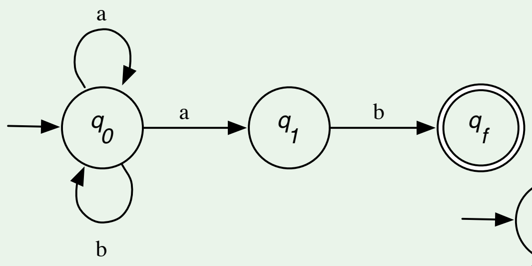
- $Q_D = \{S \mid S \subseteq Q_N\}$ *(i.e.* $Q_D = 2^{Q_N}$*)*
- $F_D = \{S \subseteq Q_N \mid S \cap F_N \neq \emptyset\}$
- $\forall S \subseteq Q_N$ *and* $a \in \Sigma$,

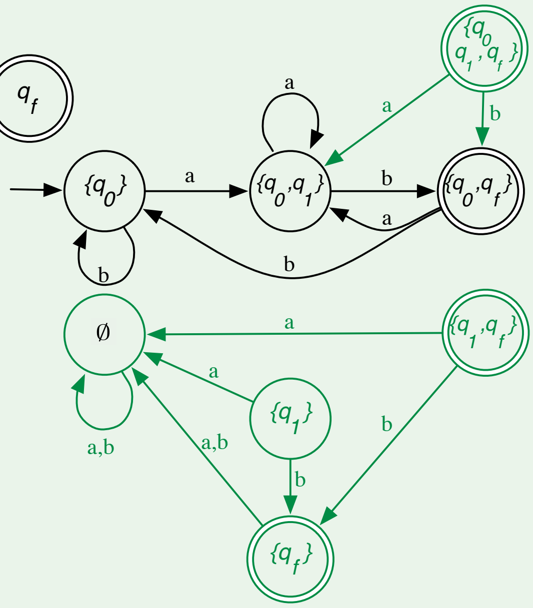
$$\delta_D(S, a) = \delta_N(S, a) \ (= \bigcup_{p \in S} \delta_N(p, a))$$

Notice that $|Q_D| = 2^{|Q_N|}$ *(however, many states are generally useless and unreachable)*

$\mathcal{C}$(NFA) $\subseteq$ $\mathcal{C}$(DFA) (cont'd)

Example (NFA N and equivalent DFA D)





5 states are unreachable

$\mathcal{C}$(NFA) $\subseteq$ $\mathcal{C}$(DFA) (end)

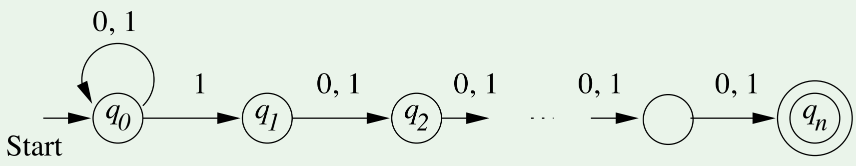
Theorem (For each NFA N, there exists a DFA D with $L(N) = L(D)$)

Sketch of proof:
One can show that $L(D) = L(N)$
It is sufficient to show that:

$$\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$$

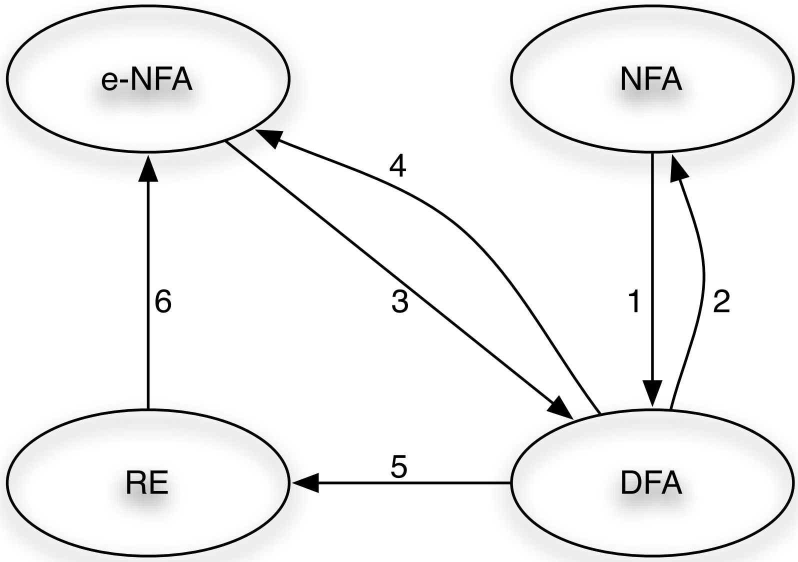
$\mathcal{C}$(NFA) $\subseteq$ $\mathcal{C}$(DFA) (cont'd)

Example (NFA N with $n+1$ states with an equivalent DFA D with 2^n states)



$\mathcal{C}$(ϵ-NFA) $\subseteq$ $\mathcal{C}$(DFA)

+ Arrow 3:



$\mathcal{C}(\epsilon\text{-NFA}) \subseteq \mathcal{C}(\text{DFA})$

Theorem (For all ϵ-NFA E, there exists a DFA D with $L(E) = L(D)$)

Sketch of proof:
Given an ϵ-NFA E:

$$E = \langle Q_E, \Sigma, \delta_E, q_0, F_E \rangle$$

let us define (build) the DFA D:

$$D = \langle Q_D, \Sigma, \delta_D, q_D, F_D \rangle$$

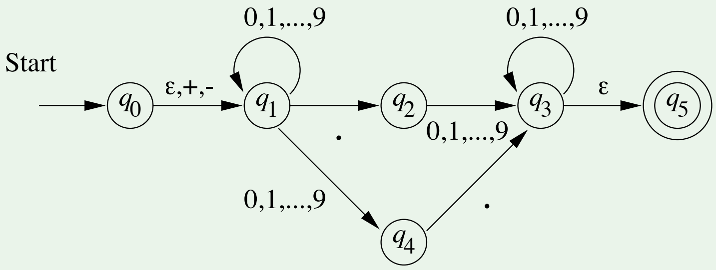
with:

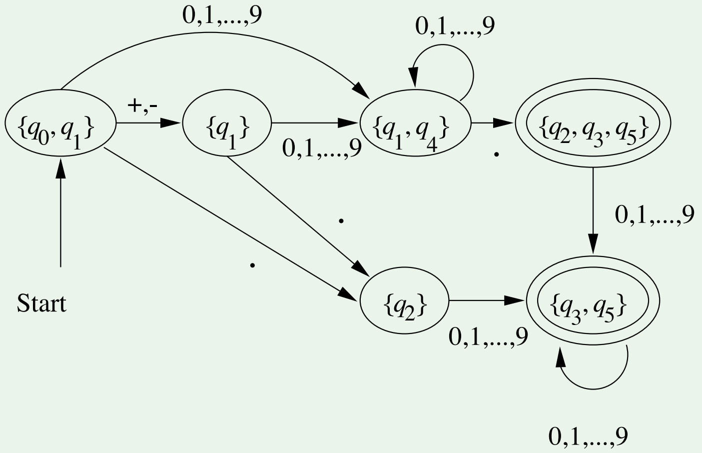
- $Q_D = \{S | S \subseteq Q_E \wedge S = eclose(S)\}$
- $q_D = eclose(q_0)$
- $F_D = \{S \mid S \in Q_D \wedge S \cap F_E \neq \emptyset\}$
- *For all $S \in Q_D$ and $a \in \Sigma$,*

$$\delta_D(S, a) = eclose(\delta_E(S, a))$$

$\mathcal{C}(\epsilon\text{-NFA}) \subseteq \mathcal{C}(\text{DFA})$ (cont'd)

Example (ϵ-NFA E and equivalent DFA D)





$\mathcal{C}(\epsilon\text{-NFA}) \subseteq \mathcal{C}(\text{DFA})$

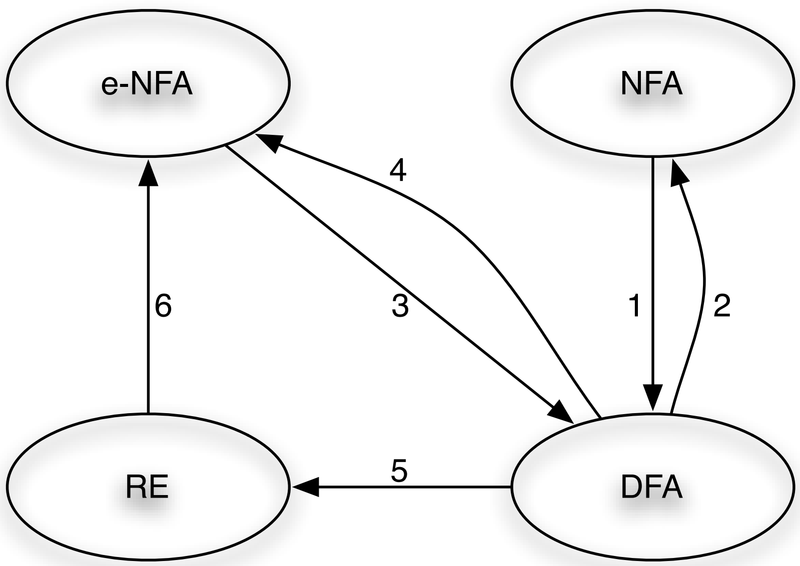
Theorem (For all ϵ-NFA E, there exists a DFA D with $L(E) = L(D)$)

Sketch of proof (cont'd):
To show that $L(D) = L(E)$, it is sufficient to show that :

$$\hat{\delta}_E(\{q_0\}, w) = \hat{\delta}_D(q_D, w)$$

$\mathcal{C}(\text{DFA}) \subseteq \mathcal{C}(\text{RE})$

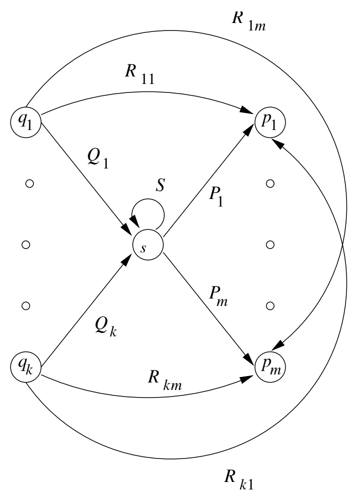
+ Arrow 5:



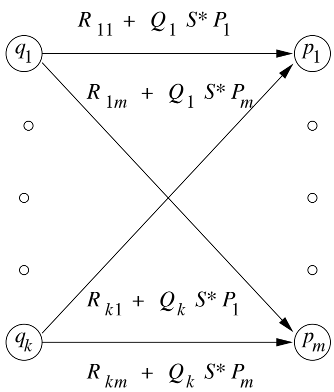
$\mathcal{C}$(FA) $\subseteq$ $\mathcal{C}$(RE) : by state elimination

Technique:

1. replace symbols labelling the FA with regular expressions
2. suppress the states (s)



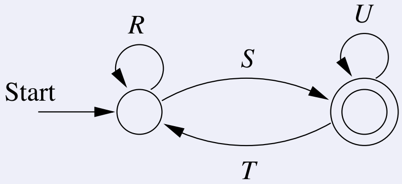
$\Rightarrow$



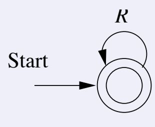
$\mathcal{C}$(FA) $\subseteq$ $\mathcal{C}$(RE) : by state elimination (cont'd)

Method

- For every accepting state q, a 2 states automaton with q_0 and q is built by removing all the other states
- For each $q \in F$ we obtain
  - either A_q:



    with the corresponding RE : $E_q = (R + SU^*T)^*SU^*$
  - or A_q:

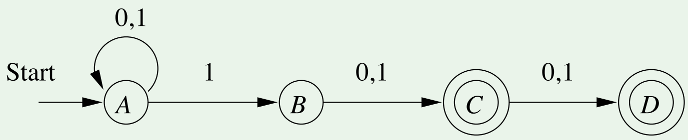


    with the corresponding RE : $E_q = R^*$
- The final RE is : $\underset{q \in F}{+} E_q$

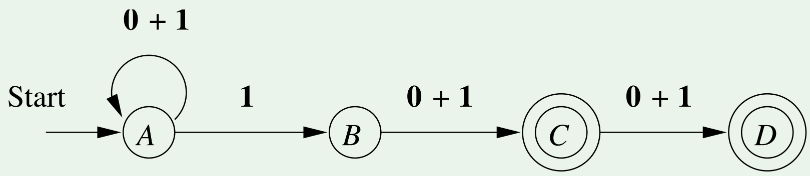
$\mathcal{C}(\text{FA}) \subseteq \mathcal{C}(\text{RE})$: by state elimination

Example (let us build a RE for the NFA *A* by state elimination)

NFA $\mathcal{A}$



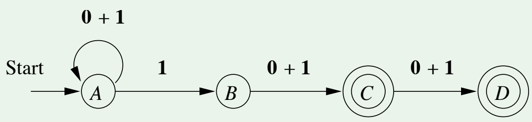
Transformation of $\mathcal{A}$:



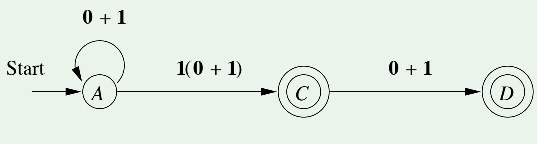
$\mathcal{C}(\text{FA}) \subseteq \mathcal{C}(\text{RE})$: by state elimination

Example (cont'd)

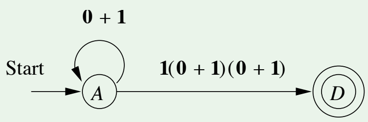
$\mathcal{A}$ modified:



Elimination of the state *B*



Elimination of the state *C* to obtain $\mathcal{A}_D$

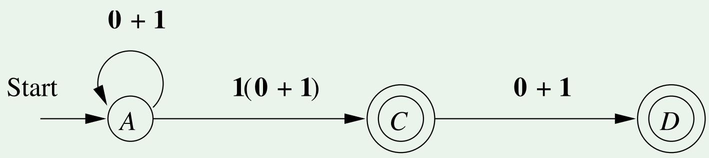


Corresponding RE: $(0 + 1)^*1(0 + 1)(0 + 1)$

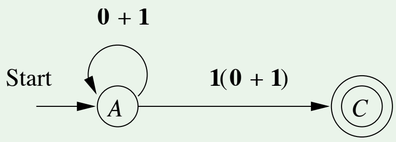
$\mathcal{C}$(FA) $\subseteq$ $\mathcal{C}$(RE) : by states elimination

Example (Let us find a RE for the FA A by states elimination)

From the automaton with B suppressed:



Elimination of the state D to obtain $\mathcal{A}_C$

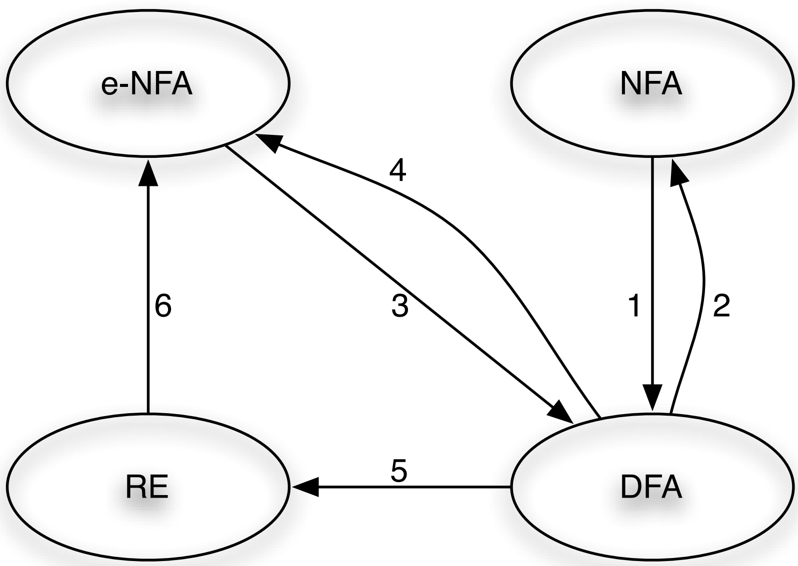


Corresponding RE: $(0+1)^*1(0+1)$

Final RE: $(0+1)^*1(0+1)(0+1) + (0+1)^*1(0+1)$

$\mathcal{C}$(RE) $\subseteq$ $\mathcal{C}$(ϵ-NFA) = $\mathcal{C}$(FA)

+ Arrow 6:

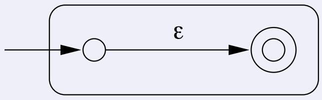


$\mathcal{C}$(RE) $\subseteq$ $\mathcal{C}$(ϵ-NFA)

Theorem (For all RE r, there exists an ϵ-NFA R with $L(R) = L(r)$)

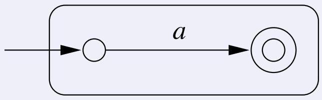
Construction

- *Base cases: automata for ϵ, $\emptyset$ and* **a***:*



(a)

(b)

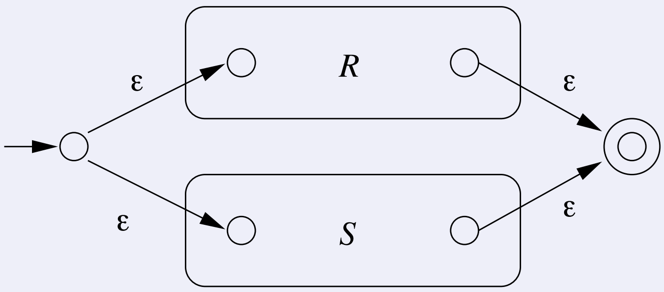


(c)

$\mathcal{C}$(RE) $\subseteq$ $\mathcal{C}$(ϵ-NFA) (cont'd)

Theorem (For all RE r, there exists an ϵ-NFA R with $L(R) = L(r)$)

- *Induction: automaton for $r + s$:*

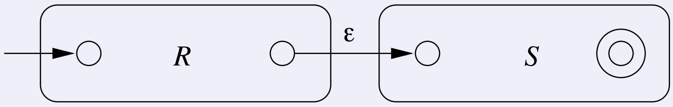


(a)

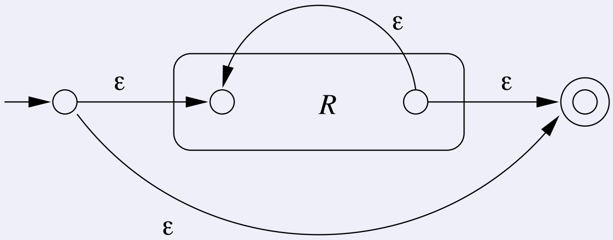
$\mathcal{C}$(RE) $\subseteq$ $\mathcal{C}$(ϵ-NFA) (cont'd)

Theorem (For all RE r, there exists an ϵ-NFA R with $L(R) = L(r)$)

- *Induction: automata for rs and r^* :*



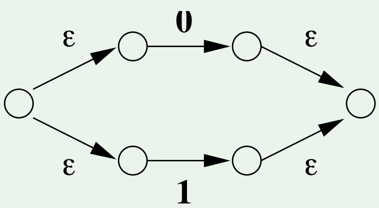
(b)



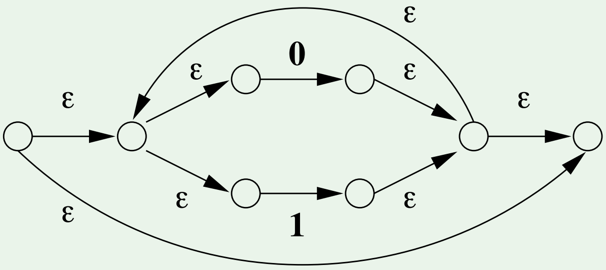
(c)

$\mathcal{C}$(RE) $\subseteq$ $\mathcal{C}$(ϵ-NFA) (cont'd)

Example (ϵ-NFA corresponding to $(0 + 1)^*1(0 + 1)$)



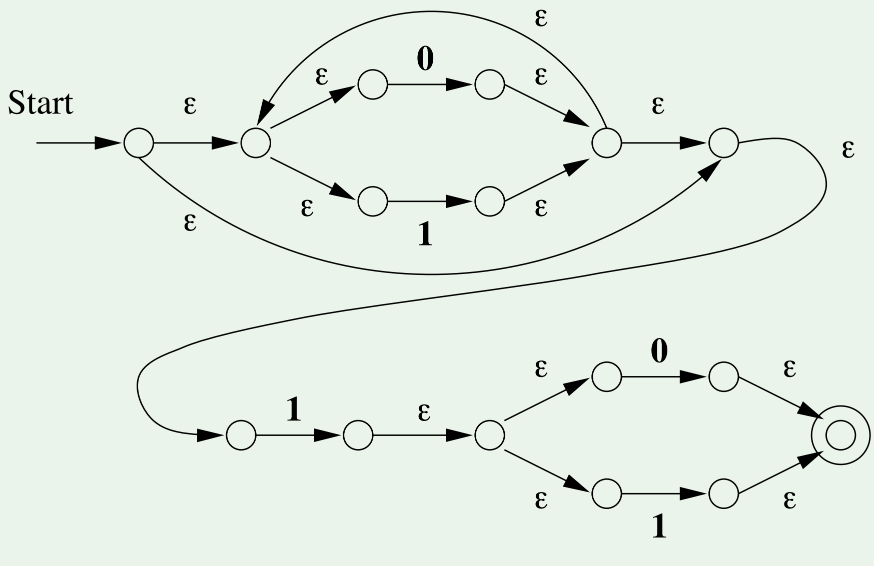
(a)



(b)

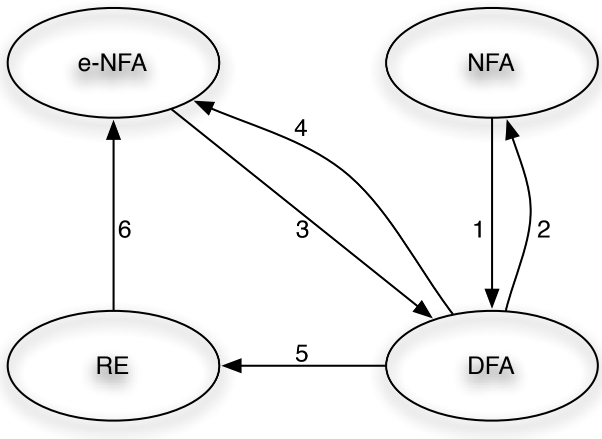
$\mathcal{C}(\text{RE}) \subseteq \mathcal{C}(\epsilon\text{-NFA})$ (cont'd)

Example (ϵ-NFA corresponding to $(0+1)^*1(0+1)$ (cont'd))



(c)

$\mathcal{C}(\text{RE}) = \mathcal{C}(\epsilon\text{-NFA}) = \mathcal{C}(\text{NFA}) = \mathcal{C}(\text{DFA})$



In conclusion,

- The 4 formalisms are equivalent and define the class of regular languages
- One can go from one formalism to the other through *automatic* translations

Outline

1. Regular languages and regular expressions
2. Finite state automata
3. Equivalence between FA and RE
4. Other types of automata
5. Some properties of regular languages

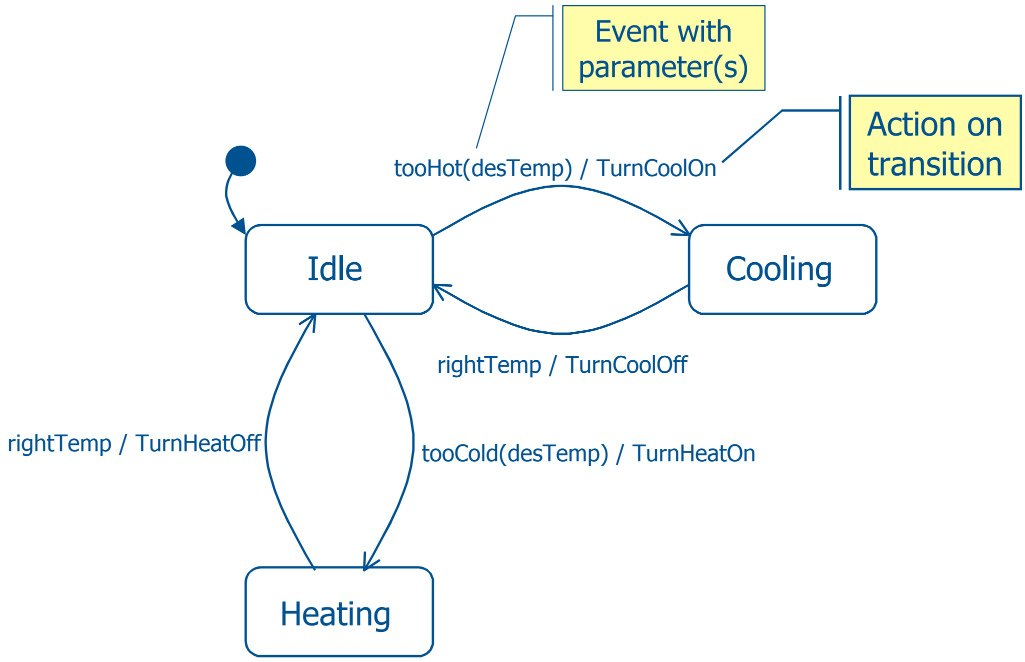
Machines with output (actions)

- Moore machines: one output for each control state
- Mealy machines: one output for each transition

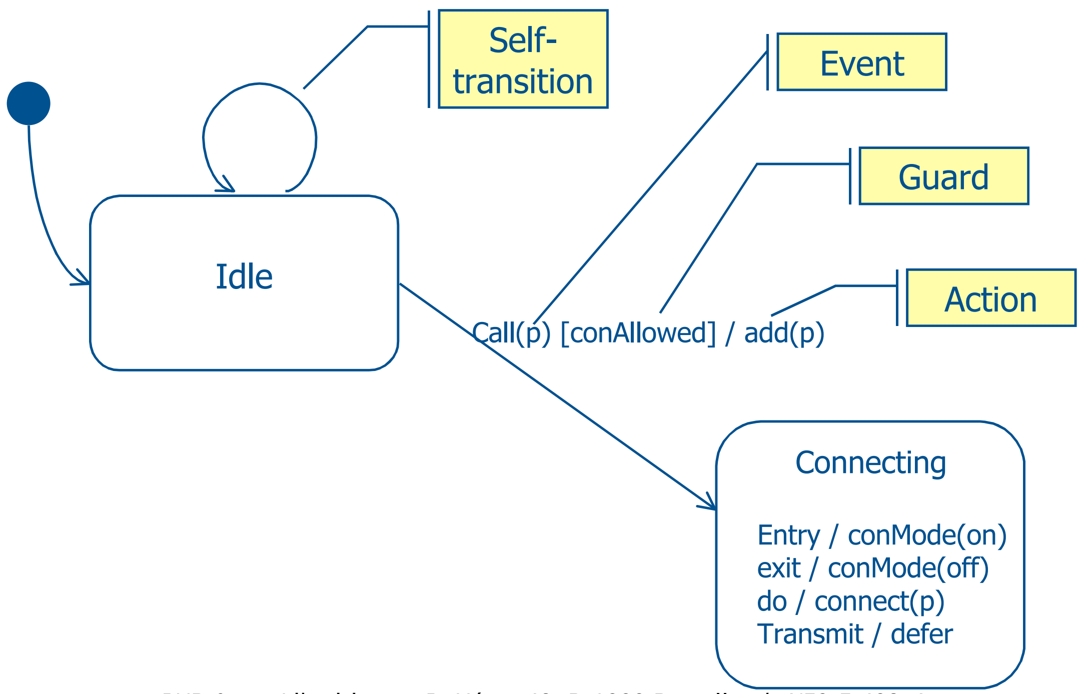
Found in UML

- statecharts
- activity diagrams

Example of UML statechart



Example of UML statechart (2)



Outline

Possible questions on languages L, L_1, L_2

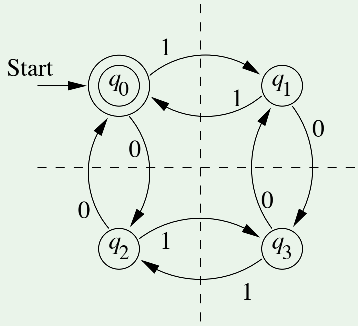
- Is L regular?
- For which operators are regular languages closed?
- $w \in L$?
- Is L empty; finite, infinite?
- $L_1 \subseteq L_2$, $L_1 = L_2$?

Is L regular ?

Example (Proving L is regular)

$L = \{w \mid w$ has an even number of 0 and 1$\}$

One can, e.g. define the DFA M and prove (generally by induction) that $L = L(M)$



Proving L is not regular

Proving that L is *not* regular requires use of the *pumping lemma for regular languages* (not seen in this course).

For which operators are regular languages closed?

Theorem

If L and M are regular, then the following languages are regular:

- *Union :* $L \cup M$
- *Concatenation :* $L.M$
- *Kleene closure :* L^*
- *Complement :* $\overline{L}$
- *Intersection :* $L \cap M$
- *Difference :* $L \setminus M$
- *Mirror image :* L^R

Chapter 3: Lexical analysis (scanning)

Outline

PUB Cours-Librairie, av. P. Héger 42, B-1000 Bruxelles / INFO-F-403_A

Roles and place of lexical analysis (scanning)

1. Identifies tokens and corresponding lexical units (Main role)
2. (Possibly) puts (non predefined) identifiers and literals in the symbol table[1]
3. Produces the listing / is linked to an intelligent editor (IDE)
4. Cleans the source program (suppresses comments, spaces, tabulations, upper-cases, etc.): acts as a filter

[1] can be done in a later analysis phase

Token, Lexical Unit, Pattern

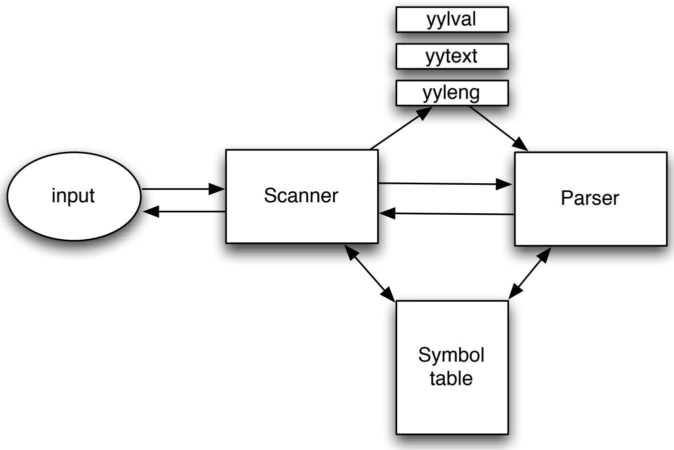
Definitions

- Lexical Unit: Generic type of lexical elements (corresponds to a set of strings with the “same” or similar semantics).
  Example: identifier, relational operator, “begin” keyword...
- Token : Instance of a lexical unit.
  Example: `N` is a token from the *identifier* lexical unit
- Pattern : Rule to describe the set of tokens of one lexical unit
  Example: identifier = letter (letter + digit)*

Relation between token, lexical unit and pattern

lexical unit = { token | pattern(token) }

Place of the scanning



- work with the input ⇒ reading the input must be optimized (buffering) to not spend too much time
- co-routine of the parser which asks the scanner each time for the next token, and receives:
  1. the recognized lexical unit
  2. information (name of the corresponding token) in the symbol table
  3. values in specific global variables (e.g.: yylval, yytext, yyleng in lex/yacc)

Boundary between scanning and parsing

The boundary between scanning and parsing is sometimes blurred

- From a logical point of view:
  - During scanning: tokens and lexical units are recognized
  - During parsing: the syntactical tree is built
- From a technical point of view:
  - During scanning: regular expressions are handled and the analysis is local
  - During parsing: context free grammar is handled and the analysis is global

Remarks:

- Sometimes scanning counts parentheses (link with an intelligent editor)
- Complex example for scanning: in FORTRAN
  `DO 5 I = 1,3` is not equivalent to `DO 5 I = 1.3`
  ⇒ look-ahead reading is needed

Outline

Elements to deal with

1. Lexical units: general rules:
   - The scanner recognises the longest possible token :
     - For <= the scanner must not stop at <
     - For a variable called `x36isa`, the scanner must not stop at `x`
   - The “keywords” (if, then, while) are in the “identifier” pattern
     ⇒ the scanner must recognize keywords in priority (`if36x` must of course be recognized as an identifier)
2. Separators: (space, tabulation, <CR>), are either discarded, or treated as empty tokens (recognized as tokens by the scanner but not transmitted to the parser)
3. Errors: the scanner can try to resynchronize in order to possibly detect further errors (but no code will be generated)

Outline

In Lex or UNIX

```
Regular expressions in Lex use the following operators:

    x         the character "x"
    "x"       an "x", even if x is an operator.
    \x        an "x", even if x is an operator.
    [xy]      the character x or y.
    [x-z]     the characters x, y or z.
    [^x]      any character but x.
    .         any character but newline.
    ^x        an x at the beginning of a line.
    x$        an x at the end of a line.
    x?        an optional x.
    x*        0,1,2, ... instances of x.
    x+        1,2,3, ... instances of x.
    x|y       an x or a y.
    (x)       an x.
    x/y       an x but only if followed by y.
    {xx}      the translation of xx from the
              definitions section.
    x{m,n}    m through n occurrences of x
```

Example of pattern of lexical units

Example (of patterns of lexical units defined as extended regular expressions)

```
spaces     [\t\n ]+
letter     [A-Za-z]
digit [0-9]      /* base 10 */
digit16 [0-9A-Fa-f]    /* base 16 */
keywords-if   if
identifier  {letter}(_|{letter}|{digit})*
integer  {digit}+
exponent   [eE][+-]?{integer}
real    {integer}("."{integer})?{exponent}?
```

All these extended regular expressions can be translated into basic regular expressions (hence into FAs)

Outline

PUB Cours-Librairie, av. P. Héger 42, B-1000 Bruxelles / INFO-F-403_A

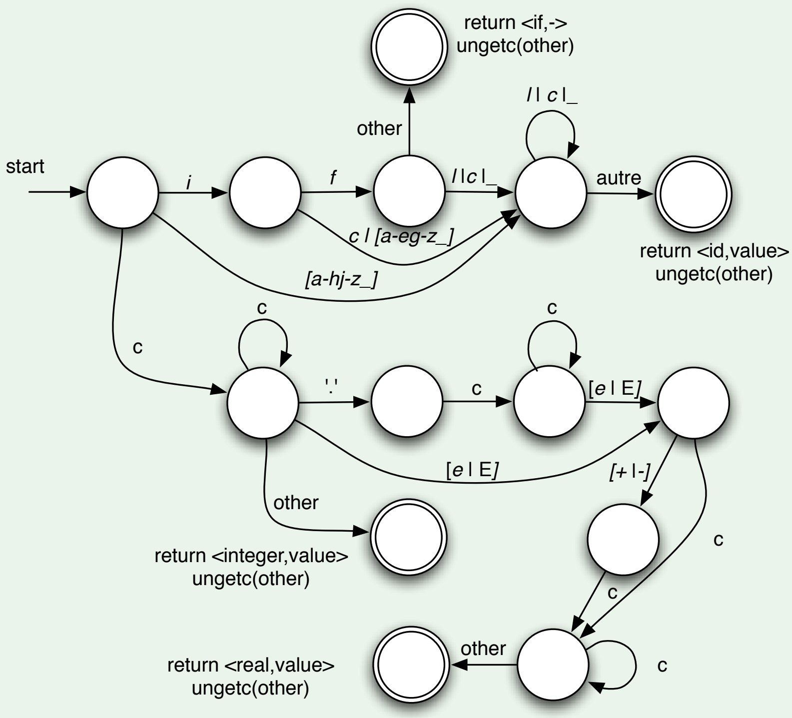
Construction of a scanner “by hand”

Principle of the construction of a scanner

- We start from the descriptions made using extended regular expressions (ERE)
- We “translate” ERE into DFA (“deterministic” finite automata)
- This DFA is decorated with actions (possible return to the last accepting state and return results and send back the possible last character(s) received)

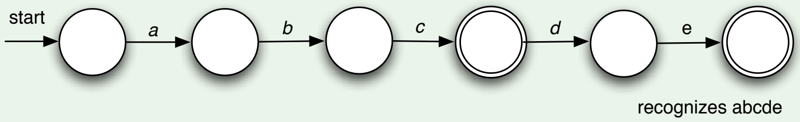
Example

Example (scanner which recognizes: if, identifier, integer and real)



Example where the last accepting configuration must be remembered

Example (scanner for $abc|abcde|\ldots$)

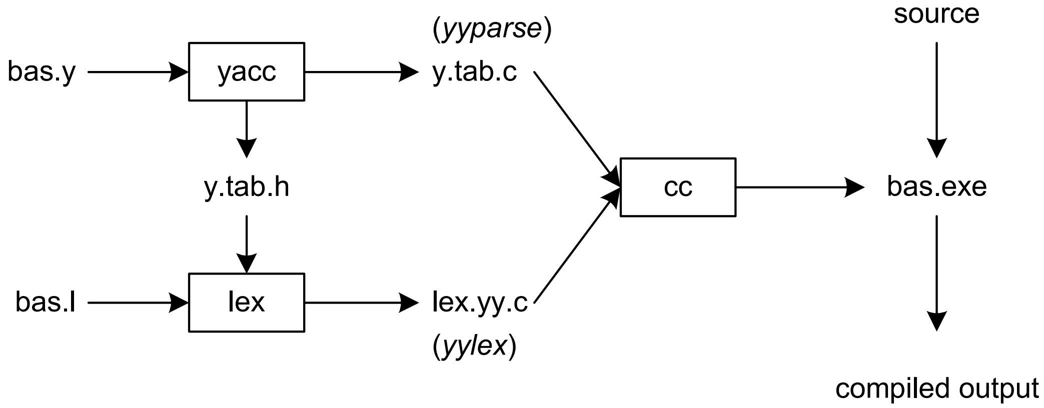


For the string *abcdx*, *abc* must be accepted and *dx* must be sent back to input (and read again later)

Outline

PUB Cours-Librairie, av. P. Héger 42, B-1000 Bruxelles / INFO-F-403_A

general procedure for the use of `Lex` (`Flex`) and `Yacc` (`Bison`)



Compilation :

```
yacc -d bas.y                      # creates y.tab.h and y.tab.c
lex bas.l                          # creates lex.yy.c
cc lex.yy.c y.tab.c -ll -o bas.exe # compiles and links
                                   # creates bas.exe
```


`Lex` specification

```
definitions
%%
rules
%%
additional code
```

The resulting scanner (`yylex()`) tries to recognize tokens and lexical units
It can use global variables :

Name	**function**
char *yytext	pointer to the recognized token (i.e. string)
yyleng	length of the token
yylval	value of the token

Predefined global variables

Lex example (1)

Example (1 of use of Lex)

```
%{
   int yylineno;
%}

%%

^(.*)\n   printf("%4d\t%s", ++yylineno, yytext);

%%

int main(int argc, char *argv[]) {
   yyin = fopen(argv[1], "r");
   yylex();
   fclose(yyin);
}
```

Remark:

In this example, the scanner (`yylex()`) runs until it reaches the end of the file

Lex example (2)

Example (2 of use of Lex)

```
digit     [0-9]
letter    [A-Za-z]
%{
   int count;
%}

%%

   /* match identifier */
{letter}({letter}{digit})*   count++;

%%

int main(void) {
   yylex();
   printf("number of identifiers = %d\n", count);
   return 0;
}
```

Lex example (3)

Example (3 of use of Lex)

```
%{
   int nchar, nword, nline;
%}

%%

\n         { nline++; nchar++; }
[^ \t\n]+ { nword++; nchar += yyleng; }
.          { nchar++; }

%%

int main(void) {
   yylex();
   printf("%d\t%d\t%d\n", nchar, nword, nline);
   return 0;
}
```


Lex example (4)

Example (4: scanner and simple expressions evaluator)

```
/*  expressions evaluator with  '+' and '-'    */
/*  Thierry Massart - 28/09/2005  */
%{
#define  NUMBER 1
int yylval;
%}

%%

[0-9]+     {yylval = atoi(yytext); return NUMBER;}
[ \t]      ;         /* ignore spaces and  tabulations */
\n         return 0; /* allows to stop at eol */
.          return yytext[0];
```

Example (4 (cont'd))

```
%%
int main() {
  int val;
  int tot=0;
  int sign=1;

  val = yylex();
  while(val !=0){
     if(val=='-')  sign *=-1;
     else if (val != '+') /* number */
     {
       tot += signe*yylval;
       sign = 1;
     }
     val=yylex();
  }
  printf("%d\n",tot);
  return 0;
}
```


Chapter 4: Grammars

Outline

Why do we use grammars?

Why do we use grammars?

- A lot of languages we want to define / use are not regular
- Context-free languages are used since the 50's (1950) to define the syntax of programming languages
- In particular, the `BNF` syntax (Backus Naur Form) is based on the notion of context-free grammars
- Most of the formal languages are defined with grammars (example: `XML`).

PUB Cours-Librairie, av. P. Héger 42, B-1000 Bruxelles / INFO-F-403_A

Outline

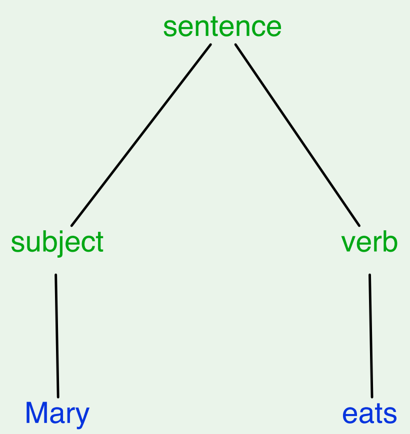
Example of a grammar

Example (Grammar of a sentence)

- sentence = subject verb
- subject = **"John"** | **"Mary"**
- verb = **"eats"** | **"speaks"**

can provide

- **John eats**
- **John speaks**
- **Mary eats**
- **Mary speaks**



Syntactic tree of the sentence
Mary eats

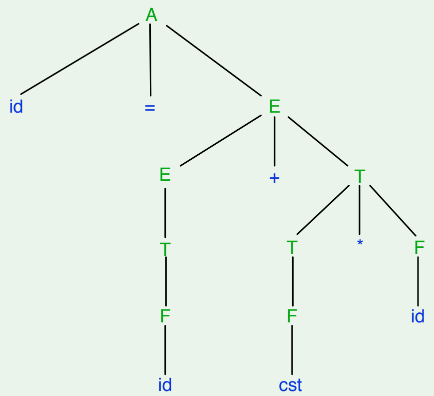
Grammar example (2)

Example (Grammar of an expression)

- A = **“id”** **“=”** E
- E = T | E **“+”** T
- T = F | T **“*”** F
- F = **“id”** | **“cst”** | **“(”** E **“)”**

can give:

- **id = id**
- **id = id + cst * id**
- ...



Syntactic tree of the sentence
id = id + cst * id

Other example

Example (The palindrome language)

Given $L_{pal} = \{w \in \Sigma^* | w = w^R\}$
For instance (abstracting upper/lower cases and spaces):

A man, a plan, a canal: Panama
Rats live on no evil star
Was it a car or a cat i saw
Ressasser
Hannah
Et la marine va, papa, venir a Malte
A Cuba, Anna a bu ça
A Laval elle l’avala
Aron, au Togo, tua Nora
SAT ORA REPO TENETO PERARO ... TAS

The last example is the sacred Latin magic square :

S	A	T	O	R
A	R	E	P	O
T	E	N	E	T
O	P	E	R	A
R	O	T	A	S

Possible literal translation: "The farmer Arepo has [as] works wheels [a plough]" Traduction littérale possible: Le semeur subreptissement tient l'oeuvre dans la rotation (des temps)

Other grammar example

Example (The palindrome language)

Let us limit to $\Sigma = \{0, 1\}$. The grammar follows an inductive reasoning:

- basis: ϵ, 0 and 1 are palindromes
- induction: suppose w is a palindrome: $0w0$ and $1w1$ are palindromes

1. $P \rightarrow \epsilon$
2. $P \rightarrow 0$
3. $P \rightarrow 1$
4. $P \rightarrow 0P0$
5. $P \rightarrow 1P1$

Another grammar's example

Terminals and variables

In the previous example:

- 0 and 1 are terminals (symbols of the terminal alphabet)
- P is a variable (also called nonterminal symbol) (additional symbol used to define the language)
- P is also the start symbol (or start variable)
- 1-5 are production rules of the grammar

Outline

PUB Cours-Librairie, av. P. Héger 42, B-1000 Bruxelles / INFO-F-403_A

Grammar: formal definition

Definition (Grammar)

Quadruplet:

$$G = \langle V, T, P, S \rangle$$

where

- V *is a finite set of variables*
- T *is a finite set of terminals*
- P *is a finite set of production rules of the form* $\alpha \rightarrow \beta$ *with*

$$\alpha \in (V \cup T)^* V (V \cup T)^* \text{ and } \beta \in (V \cup T)^*$$

- S *is a variable* ($\in V$) *called start symbol*

Formally P is a relation $P : (V \cup T)^* V (V \cup T)^* \times (V \cup T)^*$

Remark

The previous examples use context-free grammars i.e. a subclass of grammars where the production rules have the form $A \rightarrow \beta$ with $A \in V$

Formal definition of the set of palindromes on $\{0, 1\}$

Example (The palindrome language)

$$G = \langle \{A\}, \{0, 1\}, P, A \rangle$$

with $P = \{A \rightarrow \epsilon, A \rightarrow 0, A \rightarrow 1, A \rightarrow 0A0, A \rightarrow 1A1\}$

One compactly denotes the rules with the same variable as left part (here, all 5 rules) as such:

- $A \rightarrow \epsilon \mid 0 \mid 1 \mid 0A0 \mid 1A1$

Definition (A-production)

The set of rules whose left-part is the variable A is called the set of A-productions

Derivation (relation)

Definition (Derivation)

Given a grammar $G = \langle V, T, P, S\rangle$ *Then*

$$\gamma \underset{G}{\Rightarrow} \delta$$

iff

- $\exists \alpha \rightarrow \beta \in P$
- $\gamma \equiv \gamma_1 \alpha \gamma_2$ *for* $\gamma_1, \gamma_2 \in (V \cup T)^*$
- $\delta \equiv \gamma_1 \beta \gamma_2$

Remarks:

- Grammars are rewrite systems: the derivation $\gamma \equiv \gamma_1 \alpha \gamma_2 \underset{G}{\Rightarrow} \gamma_1 \beta \gamma_2 \equiv \delta$ rewrites the α part into β in the string γ which becomes δ
- When G is clearly identified, one, more simply, writes: $\gamma \Rightarrow \delta$
- $\overset{*}{\Rightarrow}$ is the reflexo-transitive closure of $\Rightarrow$
- $\alpha \overset{i}{\Rightarrow} \beta$ is a notation for a derivation of length i between α and β
- every string α which can derived from the start symbol ($S \overset{*}{\Rightarrow} \alpha$) is called sentential form

Derivation (cont'd)

With $G = \langle \{E, T, F\}, \{i, c, +, *, (,)\}, P, E\rangle$ and P :

- $E \rightarrow T \mid E + T$
- $T \rightarrow F \mid T * F$
- $F \rightarrow i \mid c \mid (E)$

One has

$$E \overset{*}{\Rightarrow} i + c * i$$

Several derivations are possibles: examples:

1. $E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow i + T$
   $\Rightarrow i + T * F \Rightarrow i + F * F \Rightarrow i + c * F \Rightarrow i + c * i$
2. $E \Rightarrow E + T \Rightarrow E + T * F \Rightarrow E + T * i$
   $\Rightarrow E + F * i \Rightarrow E + c * i \Rightarrow T + c * i \Rightarrow F + c * i \Rightarrow i + c * i$
3. $E \Rightarrow E + T \Rightarrow T + T \Rightarrow T + T * F \Rightarrow T + F * F$
   $\Rightarrow T + c * F \Rightarrow F + c * F \Rightarrow F + c * i \Rightarrow i + c * i$
4. ...

Language of G

Definition (Language of a grammar $G = \langle V, T, P, S \rangle$)

$$L(G) = \{w \in T^* \mid S \stackrel{*}{\Rightarrow} w\}$$

Definition ($L(A)$)

For a grammar $G = \langle V, T, P, S \rangle$ *with* $A \in V$

$$L(A) = \{w \in T^* \mid A \stackrel{*}{\Rightarrow} w\}$$

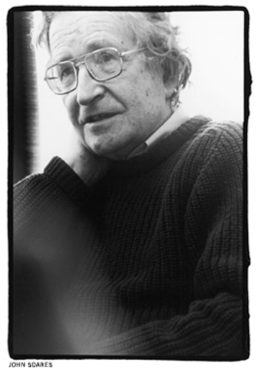
Outline

PUB Cours-Librairie, av. P. Héger 42, B-1000 Bruxelles / INFO-F-403_A

Noam Chomsky

Noam Chomsky (www.chomsky.info) (born December 7, 1928) is Institute Professor and Professor Emeritus of linguistics at the Massachusetts Institute of Technology. Chomsky is credited with the creation of the theory of generative grammars, often considered the most significant contribution to the field of theoretical linguistics of the 20th century. He also helped spark the cognitive revolution in psychology through his review of B. F. Skinner's Verbal Behavior, which challenged the behaviorist approach to the study of mind and language dominant in the 1950s. His naturalistic approach to the study of language has also impacted the philosophy of language and mind (see Harman, Fodor).

He is also credited with the establishment of the so-called Chomsky hierarchy, a classification of formal languages in terms of their generative power. Chomsky is also widely known for his political activism, and for his criticism of the foreign policy of the United States and other governments. Chomsky describes himself as a libertarian socialist, a sympathizer of anarcho-syndicalism.



The Chomsky hierarchy

Definition (The Chomsky hierarchy)

This hierarchy defines 4 classes of grammars (and of languages).

- *Type 0: Unrestricted grammars*
  The most general definition given above
- *Type 1: Context-sensitive grammars*
  Grammars where all the rules have the form:
  - $S \rightarrow \epsilon$ *and S does not appear in a right part of a rule*
  - $\alpha \rightarrow \beta$ *with* $|\alpha| \leq |\beta|$

The Chomsky hierarchy

Definition (The Chomsky hierarchy (cont'd))

- *Type 2: Context-free grammars*
  Grammars where the rules have the form:
  - $A \longrightarrow \alpha \quad$ with $\alpha \in (T \cup V)^*$
- *Type 3: Regular grammars*
  Class of grammars composed of the following 2 subclasses :
  1. *the right-linear grammars, where all the rules have the form :*
     $A \longrightarrow wB$
     $A \longrightarrow w \quad$ with $A, B \in V \wedge w \in T^*$
  2. *the left-linear grammars, where all the rules have the form :*
     $A \longrightarrow Bw$
     $A \longrightarrow w \quad$ with $A, B \in V \wedge w \in T^*$

The Chomsky hierarchy

Remarks - properties (cont'd)

- A language is of type n if there exists a grammar of type n which defines it.
- We have type 3 $\subset$ type 2 $\subset$ type 1 $\subset$ type 0

Chapter 5: Regular grammars

Outline

Regular grammars (definition)

Definition (Type 3: Regular grammars)

Class of grammars composed of the following 2 subclasses :

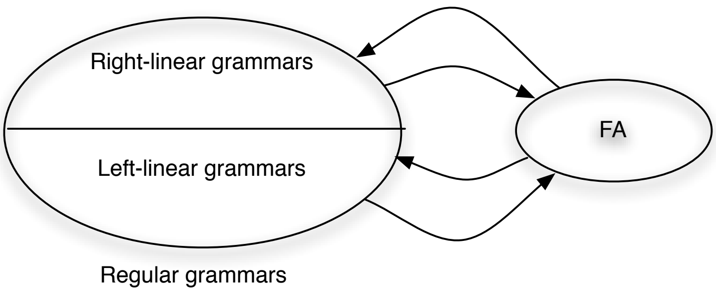
1. *the right-linear grammars, where all the rules have the form :*
   $A \rightarrow wB$
   $A \rightarrow w$ with $A, B \in V \wedge w \in T^*$
2. *the left-linear grammars, where all the rules have the form :*
   $A \rightarrow Bw$
   $A \rightarrow w$ with $A, B \in V \wedge w \in T^*$

Outline

Equivalence between regular grammars and regular languages

One can show that

1. Every language generated by a right-linear grammar is regular
2. Every language generated by a left-linear grammar is regular
3. Every regular language is generated by a right-linear grammar
4. Every regular language is generated by a left-linear grammar



Which implies that the class of languages generated by a right-linear grammar is the same that the one generated by a left-linear grammar i.e. the class of regular languages

Chapter 6: Context-free grammars

Outline

1. Reminder and definitions
2. Derivation tree
3. Cleaning and simplification of context-free grammars

Type 2 : context-free grammar

Definition (Context-free grammar (CFG))

Grammar where the rules have the form:

- $A \to \alpha$ *with* $\alpha \in (T \cup V)^*$

Definition (Context-free language (CFL))

L is a CFL if $L = L(G)$ for a CFG G

Examples of context-free grammars

Example (The palindrome language on the alphabet $\{0,1\}$)

$$G = \langle \{P\}, \{0,1\}, A, P\rangle$$

with $A = \{P \to \epsilon \mid 0 \mid 1 \mid 0P0 \mid 1P1\}$

Example (Language of arithmetic expressions)

$G = \langle \{E, T, F\}, \{i, c, +, *, (,)\}, P, E\rangle$ with P :

- $E \to T \mid E + T$
- $T \to F \mid T * F$
- $F \to i \mid c \mid (E)$

Derivations

Given $G = \langle \{E, T, F\}, \{i, c, +, *, (,)\}, P, E\rangle$ and

- $E \to T \mid E + T$
- $T \to F \mid T * F$
- $F \to i \mid c \mid (E)$

We have

$$E \overset{*}{\Rightarrow} i + c * i$$

Several derivations are possible, such as:

1. $E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow i + T$
   $\Rightarrow i + T * F \Rightarrow i + F * F \Rightarrow i + c * F \Rightarrow i + c * i$
2. $E \Rightarrow E + T \Rightarrow E + T * F \Rightarrow E + T * i$
   $\Rightarrow E + F * i \Rightarrow E + c * i \Rightarrow T + c * i \Rightarrow F + c * i \Rightarrow i + c * i$
3. $E \Rightarrow E + T \Rightarrow T + T \Rightarrow T + T * F \Rightarrow T + F * F$
   $\Rightarrow T + c * F \Rightarrow F + c * F \Rightarrow F + c * i \Rightarrow i + c * i$
4. ...

Definition (Left-most (resp. right-most) derivation)

Derivation of the grammar G which always first rewrites the left-most (resp. right-most) variable of the sentential form.

- *derivation 1. (of the example) is left-most (one writes* $S\ {}_G\!\overset{*}{\Rightarrow} \alpha$*)*
- *derivation 2. is right-most (one writes* $S \overset{*}{\Rightarrow}_G \alpha$*)*

Outline

Derivation tree

For a context-free grammar G, one can show that $w \in L(G)$ using a derivation tree (or parse tree).

Example (derivation tree)

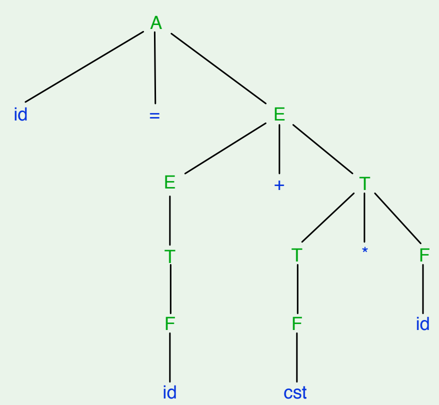
The grammar with the rules:

- A = **“id” “=”** E
- E = T | E **“+”** T
- T = F | T **“*”** F
- F = **“id”** | **“cst”** | **“(”** E **“)”**

can give:

- **id = id + cst * id**

with the derivation tree:



Construction of a derivation tree

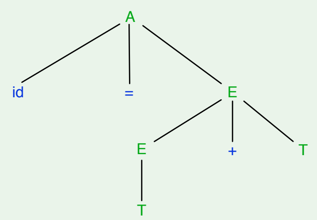
Definition (Derivation tree)

Given a CFG $G = \langle V, T, P, S \rangle$. A derivation tree for G is such that:

1. *each internal node is labeled by a variable*
2. *each leaf is labeled by a terminal, a variable or ϵ.*
   Each leaf ϵ is the only son of its father
3. *If an internal node is labeled A and its sons (from left to right) are labeled $X_1, X_2, \ldots, X_k$ then $A \rightarrow X_1 X_2 \ldots X_k \in P$*

Example (derivation tree)

For the grammar on slide 156



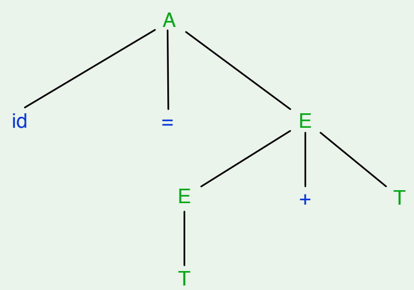
Yield of a derivation tree

Definition (Yield of a derivation tree)

String formed by the concatenation of the labels of the leaves in the left-right order (corresponds to the derived sentential form).

Example (of yield of a derivation tree)

The yield of the following derivation tree:



is

$$id = T + T$$

Complete derivation tree and A-derivation tree

Definition (*Complete* derivation tree)

*Given a CFG $G = \langle V, T, P, S \rangle$, a **complete** derivation tree for G is a derivation tree such that:*

1. *The root is labeled for the start symbol*
2. *Each leaf is labeled by a terminal or ϵ (not a variable).*

Example (of complete tree)

See slide 156

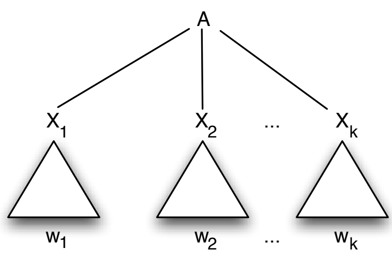
Definition (A-derivation tree)

Derivation tree whose root is labeled by a variable A

Reminder and definitions
Derivation tree
Cleaning and simplification of context-free grammars

Why these grammars are called **context-free**

$A \Rightarrow X_1 X_2 \ldots X_k \overset{*}{\Rightarrow} w$ corresponds to a derivation tree of the form :



- Each X_i is derived independently of the other X_j.
- Therefore $X_i \overset{*}{\Rightarrow} w_i$
- Note that left-most and right-most derivations handle each variable one at a time

Outline

Ambiguous context-free grammars

Ambiguous context-free grammar

- For a CFG G every string w of $L(G)$ has at least a derivation tree for G.
- $w \in L(G)$ can have several derivation trees for G: in that case the grammar is ambiguous.
- Ideally, to allow proper parsing, a grammar must not be ambiguous. Indeed, the derivation tree determines the code generated by the compiler.
- In that case, we try to modify the grammar to suppress the ambiguities.
- There is no algorithm to suppress the ambiguities of a context-free grammar.
- Some context-free languages are *inherently ambiguous*!

Ambiguous context-free grammar

Example (of inherently ambiguous context free language)

$$L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{ a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$$

Example of CFG for L

$$\begin{array}{l} S \longrightarrow AB \mid C \\ A \longrightarrow aAb \mid ab \\ B \longrightarrow cBd \mid cd \\ C \longrightarrow aCd \mid aDd \\ D \longrightarrow bDc \mid bc \end{array}$$

With G, for all $i \geq 0$ $a^i b^i c^i d^i$ has 2 derivation trees. One can prove that any other CFG for L is ambiguous

Removal of ambiguities

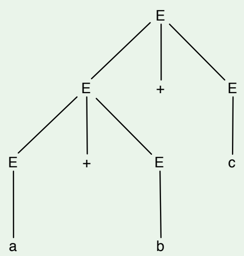
Priority and associativity

When the language defines strings composed of instructions and operations, the syntactic tree (which will determine the code produced by the compiler) must reflect

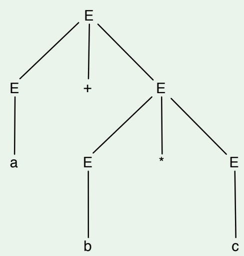
- the priorities and
- associativity

Example (of trees associated to expressions)

$a + b + c$



$a + b * c$



Ambiguity removal

Priority and associativity

To respect the left hand-side associativity, one does not write

$$E \rightarrow E + E \mid T$$

but

$$E \rightarrow E + T \mid T$$

To respect priorities, we define several levels of variables / rules (the start symbol has level 0): the operators with lowest priority are defined at a smallest level (closer of the start symbol) than the one with more priority.
We should not write

$$E \rightarrow T + E \mid T * E \mid T$$
$$T \rightarrow id \mid (E)$$

but use 2 levels instead:

$$E \rightarrow T + E \mid T$$
$$T \rightarrow F * T \mid F$$
$$F \rightarrow id \mid (E)$$

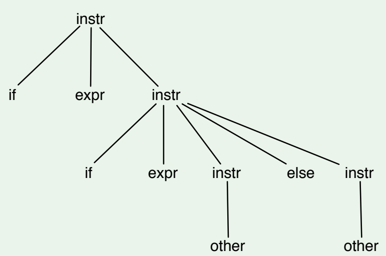
Ambiguity removal

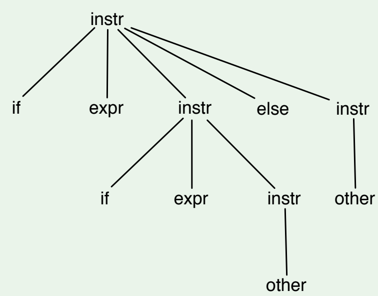
Example (Associativity of the “if” instruction)

The grammar:

$$instr \rightarrow if\ expr\ instr \mid if\ expr\ instr\ else\ instr \mid other$$

is ambiguous





In usual imperative languages, the left tree is the adequate one.

Ambiguity removal

Example (Associativity of the "if" instruction)

One can, e.g., transform the grammar into:

instr → *open* | *close*
close → *if expr close else close* | *other*
open → *if expr instr*
open → *if expr close else open*

Removal of useless symbols

Definition (useful / useless symbols)

For a grammar $G = \langle V, T, P, S \rangle$,

- *a symbol X is* **useful** *if there exists a derivation*

$$S \overset{*}{\underset{G}{\Rightarrow}} \alpha X \beta \overset{*}{\underset{G}{\Rightarrow}} w$$

  for strings α, β *and a string of terminals w*
  Otherwise X is useless.
- *a symbol X* **produces something** *if* $X \overset{*}{\underset{G}{\Rightarrow}} w$ *for a string w of terminals.*
- *a symbol X is* **accessible** *if* $S \overset{*}{\underset{G}{\Rightarrow}} \alpha X \beta$ *for some strings* α, β

Theorem (For CFGs: useful symbols = (accessible + produce something))

In a CFG,

every symbol is accessible and produces something
$\Longleftrightarrow$
every symbol is useful

Removal of useless symbols

Theorem (Removal of useless symbols of a CFG)

Let $G = \langle V, T, P, S\rangle$ be a CFG.
If $G' = \langle V', T', P', S\rangle$ is the grammar provided after the 2 following steps:

1. *Removing the symbols that produce nothing and the rules where they appear in G (we obtain $G_I = \langle V_I, T_I, P_I, S\rangle$),*
2. *Removing inaccessible symbols and productions where they appear in G_I.*

then G' is equivalent to G and has no useless symbols.

Algorithm to compute the set of symbols which produce something

Algorithm to compute the set $g(G)$ of symbols which produce something in $G = \langle V, T, P, S\rangle$

- Basis: $g(G) \leftarrow T$
- Induction: If $\alpha \in (g(G))^*$ et $X \rightarrow \alpha \in P$ then $g(G) \stackrel{\cup}{\leftarrow} \{X\}$

Example (of computation of $g(G)$)

Given G with the rules $S \rightarrow AB \mid a, A \rightarrow b$

1. Initially $g(G) \leftarrow \{a, b\}$
2. $S \rightarrow a$ then $g(G) \stackrel{\cup}{\leftarrow} \{S\}$
3. $A \rightarrow b$ then $g(G) \stackrel{\cup}{\leftarrow} \{A\}$
4. Finally $g(G) = \{S, A, a, b\}$

Theorem (At saturation, $g(G)$ contains the set of all symbols which produce something)

Algorithm to compute the set of accessible symbols

Algorithm to compute the set $r(G)$ of accessible symbols of $G = \langle V, T, P, S\rangle$

- Base: $r(G) \leftarrow \{S\}$
- Induction: If $A \in r(G)$ and $A \rightarrow \alpha \in P$ then $r(G) \stackrel{\cup}{\leftarrow} \{X \mid \exists \alpha_1, \alpha_2 : \alpha = \alpha_1 X \alpha_2\}$

Example (of computation of $r(G)$)

Given G with the rules $S \rightarrow AB \mid a, A \rightarrow b$

1. Initially $r(G) \leftarrow \{S\}$
2. $S \rightarrow AB$ $r(G) \stackrel{\cup}{\leftarrow} \{A, B\}$
3. $S \rightarrow a$ then $r(G) \stackrel{\cup}{\leftarrow} \{a\}$
4. $A \rightarrow b$ then $r(G) \stackrel{\cup}{\leftarrow} \{b\}$
5. Finally $r(G) = \{S, A, B, a, b\}$

Theorem (At saturation, $r(G)$ contains the set of accessible symbols)

Removal of left-recursion

Definition (Left-recursion)

A CFG G is left-recursive if there exists a derivation $A \stackrel{*}{\underset{G}{\Rightarrow}} A\alpha$

Note:

We will see, in the chapter on parsing, that left-recursion is a problem for *top-down parsing*. In this case, one replaces left-recursion by another kind of recursion.

Removal of left-recursion

Algorithm to remove left-recursion

Let

$$A \rightarrow A\alpha_1 \mid \cdots \mid A\alpha_r$$

be the set of directly left-recursive A-productions and

$$A \rightarrow \beta_1 \mid \cdots \mid \beta_s$$

the other A-productions.

All these productions are replaced by:

$$A \rightarrow \beta_1 A' \mid \cdots \mid \beta_s A'$$
$$A' \rightarrow \alpha_1 A' \mid \cdots \mid \alpha_r A' \mid \epsilon$$

where A' is a new variable

Removal of left-recursion

General algorithm to remove left-recursion

With $V = \{A_1, A_2, \ldots, A_n\}$.

```
For i = 1 to n do
   For j = 1 to i − 1 do
      For each production of the form Ai → Aj α do
         Remove Ai → Aj α from the grammar
         For each production of the form Aj → β do
            Add Ai → βα to the grammar
         od
      od
   od
   Remove direct left-recursion of the Ai-productions
od
```

Removal of left-recursion

Example (of removal of left-recursion)

Given G with the rules:

$$\begin{array}{lcl} A & \rightarrow & Ab \mid a \mid Cf \\ B & \rightarrow & Ac \mid d \\ C & \rightarrow & Bg \mid Ae \mid Cc \end{array}$$

Treatment of A:

$$\begin{array}{lcl} A & \rightarrow & aA' \mid CfA' \\ A' & \rightarrow & bA' \mid \epsilon \\ B & \rightarrow & Ac \mid d \\ C & \rightarrow & Bg \mid Ae \mid Cc \end{array}$$

Treatment of B:

$$\begin{array}{lcl} A & \rightarrow & aA' \mid CfA' \\ A' & \rightarrow & bA' \mid \epsilon \\ B & \rightarrow & aA'c \mid CfA'c \mid d \\ C & \rightarrow & Bg \mid Ae \mid Cc \end{array}$$

Treatment of C :

$$\begin{array}{lcl} A & \rightarrow & aA' \mid CfA' \\ A' & \rightarrow & bA' \mid \epsilon \\ B & \rightarrow & aA'c \mid CfA'c \mid d \\ C & \rightarrow & aA'cg \mid dg \mid aA'e \mid \\ & & CfA'cg \mid CfA'e \mid Cc \end{array}$$

Treatment of C (direct recursion):

$$\begin{array}{lcl} A & \rightarrow & aA' \mid CfA' \\ A' & \rightarrow & bA' \mid \epsilon \\ B & \rightarrow & aA'c \mid CfA'c \mid d \\ C & \rightarrow & aA'cgC' \mid dgC' \mid aA'eC' \\ C' & \rightarrow & fA'cgC' \mid fA'eC' \mid cC' \mid \epsilon \end{array}$$

Left-factoring

Definition (Rules that can be factored)

In a CFG G, various productions can be left-factored if they have the form

$$A \rightarrow \alpha\beta_1 \mid \cdots \mid \alpha\beta_n$$

with a common prefix $\alpha \neq \epsilon$
Remark: G can have other A-productions.

Note:

For top-down parsers, we will see that rules *must* be left-factored.

Algorithm for left-factoring

Replace:

$$A \rightarrow \alpha\beta_1 \mid \cdots \mid \alpha\beta_n$$

by

$$\begin{array}{l} A \rightarrow \alpha A' \\ A' \rightarrow \beta_1 \mid \cdots \mid \beta_n \end{array}$$

Definition (Unit production)

A unit production has the form :

$$A \rightarrow B$$

with $B \in V$

Note:

Unit productions are seen as rules that make no “progress” in the derivation: it is hence better to remove them

Algorithm to remove unit productions

For all $A \stackrel{*}{\Rightarrow} B$ using only unit productions, and $B \rightarrow \alpha$ a non unit production
Add:

$$A \rightarrow \alpha$$

At the end remove all unit productions.

Chapter 7: Pushdown automata and properties of context-free languages

Outline

1. Pushdown automata (PDA)
2. Equivalence between PDA and CFG
3. Properties of context-free languages

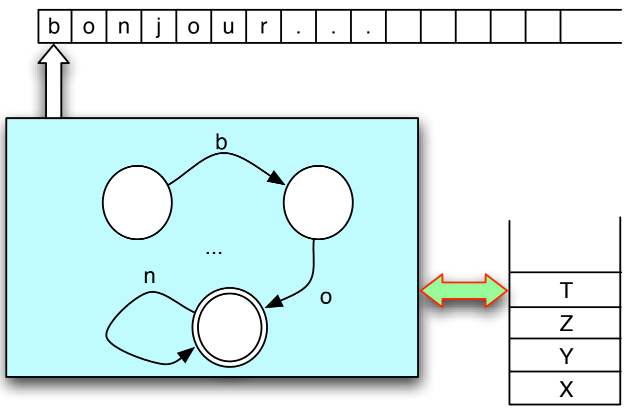
Introduction

Informal presentation

A pushdown automaton (PDA) is, in short, an ϵ-NFA with a stack.

During a transition, the PDA

1. Consumes an input symbol (or not if it is an ϵ-transition)
2. Changes its control state
3. Replaces the symbol T on top of the stack by a string (which, in particular can be ϵ (pop), "T" (no change), "AT" (push a symbol A))

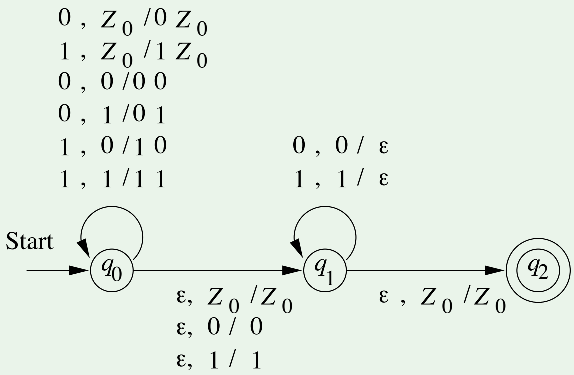


Example

Example (PDA for $L_{wwr} = \{ww^R \mid w \in \{0,1\}^*\}$)

Corresponds to the "grammar" $P \rightarrow 0P0 \mid 1P1 \mid \epsilon$.
One can build an equivalent PDA equivalent with 3 states, which works as follows:

- In q_0 It can guess we are in w: push the symbol on the stack
- In q_0 It can guess we are in the middle (at the end of w): it goes to state q_1
- In q_1 It compares what is read and what is on the stack: if both symbols are identical, the comparison is correct, it pops the top of the stack and continues (otherwise it is stuck)
- In q_1 If it meets the initial symbol on the stack, it goes to state q_2 (accepting).



PDA: formal definition

Definition

A PDA is a 7-tuple:

$$P = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$$

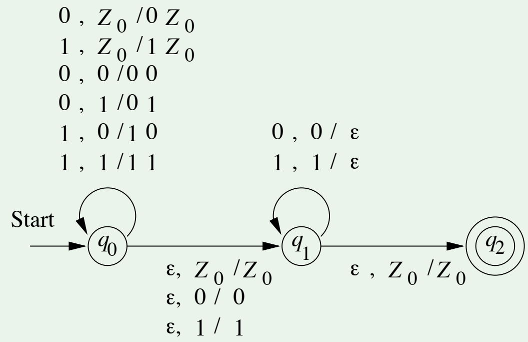
where

- Q *is a finite set of states*
- Σ *is the input alphabet*
- Γ *is the stack alphabet*
- $\delta : Q \times (\Sigma \cup \{\epsilon\} \times \Gamma) \rightarrow 2^{Q \times \Gamma^*}$ *is the transition function*
- $q_0 \in Q$ *is the initial state*
- $Z_0 \in \Gamma$ *is the initial symbol on the stack*
- $F \subseteq Q$ *is the set of accepting states*

PUB Cours-Librairie, av. P. Héger 42, B-1000 Bruxelles / INFO-F-403_A

Example of PDA

Example (of formal definition of the following PDA P)



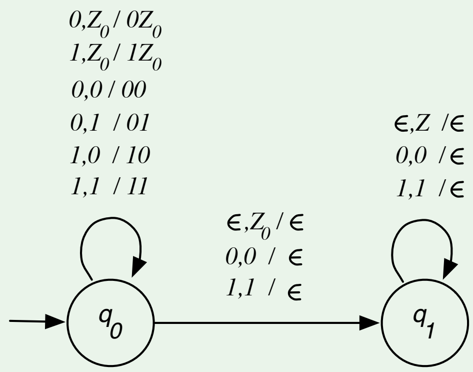
$$P = \langle \{q_0, q_1, q_2\}, \{0,1\}, \{0,1,Z_0\}, \delta, q_0, Z_0, \{q_2\} \rangle$$

δ	$(0, Z_0)$	$(1, Z_0)$	$(0, 0)$	$(0, 1)$	$(1, 0)$	$(1, 1)$
$\rightarrow q_0$	$\{(q_0, 0Z_0)\}$	$\{(q_0, 1Z_0)\}$	$\{(q_0, 00)\}$	$\{(q_0, 01)\}$	$\{(q_0, 10)\}$	$\{(q_0, 11)\}$
q_1	$\emptyset$	$\emptyset$	$\{(q_1, \epsilon)\}$	$\emptyset$	$\emptyset$	$\{(q_1, \epsilon)\}$
$*q_2$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

δ	(ϵ, Z_0)	$(\epsilon, 0)$	$(\epsilon, 1)$
$\rightarrow q_0$	$\{(q_1, Z_0)\}$	$\{(q_1, 0)\}$	$\{(q_1, 1)\}$
q_1	$\{(q_2, Z_0)\}$	$\emptyset$	$\emptyset$
$*q_2$	$\emptyset$	$\emptyset$	$\emptyset$

Other example of PDA

Example (of formal definition of the following PDA P')



$$P = \langle \{q_0, q_1\}, \{0,1\}, \{0,1,Z_0\}, \delta, q_0, Z_0, \emptyset \rangle$$

δ	$(0, Z_0)$	$(1, Z_0)$	$(0, 0)$	$(0, 1)$
$\rightarrow q_0$	$\{(q_0, 0Z_0)\}$	$\{(q_0, 1Z_0)\}$	$\{(q_0, 00), (q_1, \epsilon)\}$	$\{(q_0, 01)\}$
q_1	$\emptyset$	$\emptyset$	$\{(q_1, \epsilon)\}$	$\emptyset$

δ	$(1, 0)$	$(1, 1)$	(ϵ, Z_0)
$\rightarrow q_0$	$\{(q_0, 10)\}$	$\{(q_0, 11), (q_1, \epsilon)\}$	$\{(q_1, \epsilon)\}$
q_1	$\emptyset$	$\{(q_1, \epsilon)\}$	$\{(q_1, \epsilon)\}$

Accepting condition

A string w is accepted:

- By empty stack: the string is completely read and the stack is empty.
- By final state: the string is completely read and the PDA is in an accepting state.

Remark

- For a PDA P, 2 (a priori different) languages are defined:
  - $N(P)$ (acceptation by empty stack) and
  - $L(P)$ (acceptation by final state)
- $N(P)$ does not use F and is therefore not modified if one defines $F = \emptyset$

Configuration and accepted language

Definition (Configuration of a PDA $P = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$)

Triple $\langle q, w, \gamma \rangle \in Q \times \Sigma^* \times \Gamma^*$

- *Initial configuration:* $\langle q_0, w, Z_0 \rangle$ *where* w *is the string to accept*
- *Final configuration using empty stack acceptation:* $\langle q, \epsilon, \epsilon \rangle$ *(with any* q*)*
- *Final configuration using final state acceptation:* $\langle q, \epsilon, \gamma \rangle$ *with* $q \in F$ *(with any* $\gamma \in \Gamma^*$*)*

Definition (Configuration change)

$$\langle q, aw, X\beta \rangle \underset{P}{\vdash} \langle q', w, \alpha\beta \rangle$$

iff

$$\langle q', \alpha \rangle \in \delta(q, a, X)$$

Languages of P: $L(P)$ and $N(P)$

Definition ($L(P)$ and $N(P)$)

$$L(P) = \{w \mid w \in \Sigma^* \ \wedge \ \exists q \in F, \gamma \in \Gamma^* : \langle q_0, w, Z_0 \rangle \overset{*}{\underset{P}{\vdash}} \langle q, \epsilon, \gamma \rangle\}$$

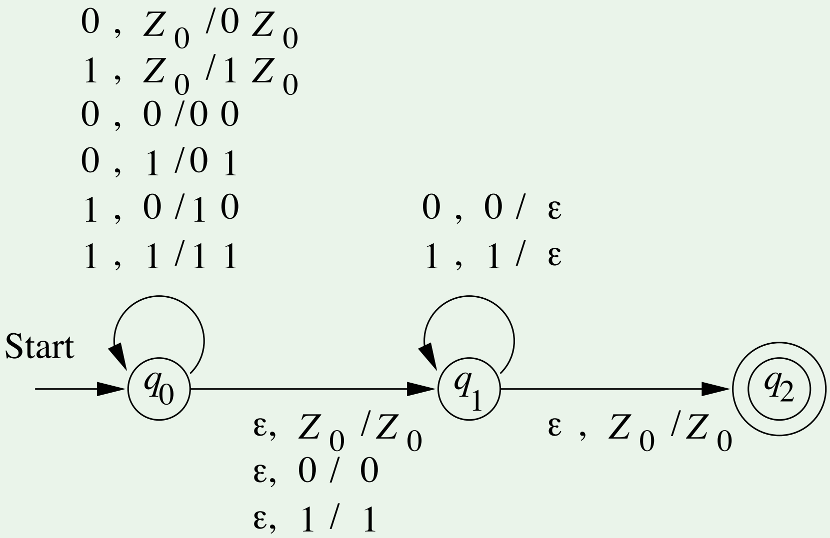
$$N(P) = \{w \mid w \in \Sigma^* \ \wedge \ \exists q \in Q : \langle q_0, w, Z_0 \rangle \overset{*}{\underset{P}{\vdash}} \langle q, \epsilon, \epsilon \rangle\}$$

where

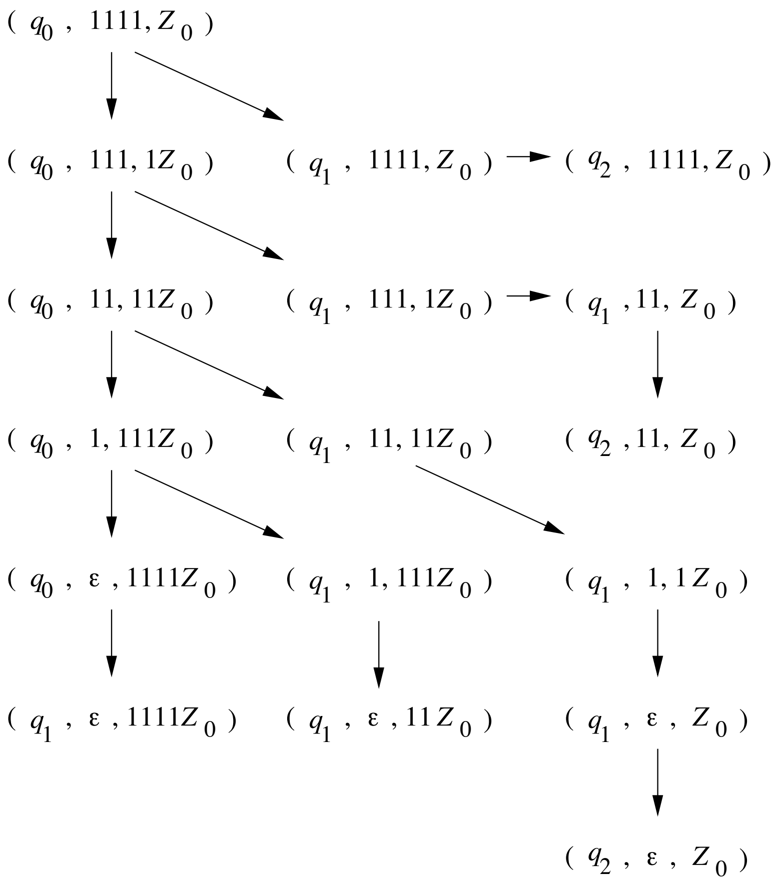
$\overset{*}{\underset{P}{\vdash}}$ *is the reflexo-transitive closure of* $\underset{P}{\vdash}$

Example of “runs”

Example (PDA P with $L(P) = L_{wwr} = \{ww^R \mid w \in \{0,1\}^*\}$ and possible runs for the string 1111)



sequences (see next slide)



Example of languages accepted by a PDA

Example (The PDA P on slide 183)

$$L(P) = \{ww^R \mid w \in \{0,1\}^*\} \quad N(P) = \emptyset$$

Example (PDA P' on slide 184)

$$L(P') = \emptyset \quad N(P') = \{ww^R \mid w \in \{0,1\}^*\}$$

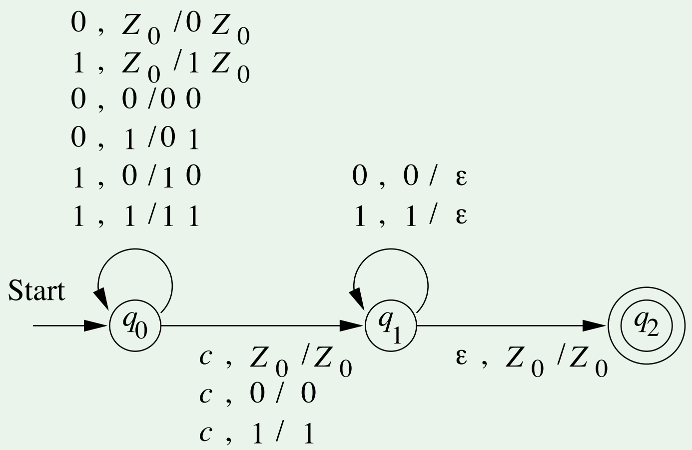
Deterministic PDA (DPDA)

Definition (deterministic PDA (DPDA))

A PDA $P = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$ is *deterministic* iff

1. $\delta(q, a, X)$ is always either empty or a singleton (for $a \in \Sigma \cup \{\epsilon\}$)
2. If $\delta(q, a, X)$ is not empty then $\delta(q, \epsilon, X)$ is empty

Example (Deterministic PDA P with $L(P) = \{wcw^R \mid w \in \{0,1\}^*\}$)



Deterministic PDA

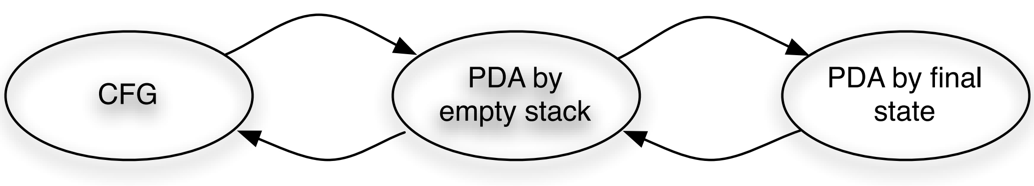
Theorem (The class of languages defined by a deterministic PDA is strictly included in the class of languages defined by a general PDA)

Proof sketch:
One can show that the language L_{wwr} defined on slide 183 cannot be defined by a deterministic PDA

Outline

PDA-CFG equivalence

One can show the following inclusions (each arrow corresponds to an inclusion)



which proves that:

Theorem (The following three classes of languages are equivalent)

- *The languages defined by CFGs (i.e. CFLs)*
- *The languages defined by a PDA with acceptation by empty stack*
- *The languages defined by a PDA with acceptation by final state*

Outline

Questions that one can wonder on languages L, L_1, L_2

- Is L context-free?
- For which operators are context-free languages closed?
- $w \in L$?
- Is L empty, finite, infinite?
- $L_1 \subseteq L_2$, $L_1 = L_2$?

For which operators are context-free languages closed?

Theorem (If L and M are context-free, then the following languages are context-free)

- *Union:* $L \cup M$
- *Concatenation:* $L.M$
- *Kleene closure:* L^*
- *Mirror image:* L^R

Theorem (If L and M are context-free, then the following languages may not be context-free)

- *Complement:* $\overline{L}$
- *Intersection:* $L \cap M$
- *Difference:* $L \setminus M$

Pushdown automata (PDA)
Equivalence between PDA and CFG
Properties of context-free languages

Undecidable problems for CFL

The following problems on CFL are undecidable (there is no algorithm to solve them in a general way)

- Is the CFG G ambiguous?
- Is the CFL L inherently ambiguous?
- Is the intersection of 2 CFLs empty?
- Is the CFL $L = \Sigma^*$?
- $L_1 \subseteq L_2$?
- $L_1 = L_2$?
- Is $\overline{L(G)}$ a CFL?
- Is $L(G)$ deterministic?
- Is $L(G)$ regular?

Chapter 8: Syntactic analysis (parsing)

1. Roles and place of parsing
2. Top-down parsing
3. Bottom-up parsing

Outline

1. Roles and place of parsing
2. Top-down parsing
3. Bottom-up parsing

Roles and place of the parser

Main role of the parser

- Verify that the structure of the string of tokens provided by the scanner (typically the program) belongs to the language (generally defined by a context-free grammar)
- Build the syntactic tree corresponding to that string of tokens
- Play the role of conductor (main program) of the compiler (syntax-oriented compiler)

Place of the parser

Between the scanner and the semantic analyzer:

- It calls the scanner to ask for tokens and
- It calls the semantic analyzer and then the code generator to finish the analysis and generate the corresponding code

Token = terminal of the grammar

Note:

In the following, each token is symbolized by a terminal of the grammar.

$ in the end of the string

In the grammar which defines the syntax, one will add systematically a new start symbol S', a new terminal $ (which symbolizes the end of the string)[a] and the production rule $S' \rightarrow S\$$ where S is the old start symbol.

[a]We suppose that neither S' nor $ are used anywhere else

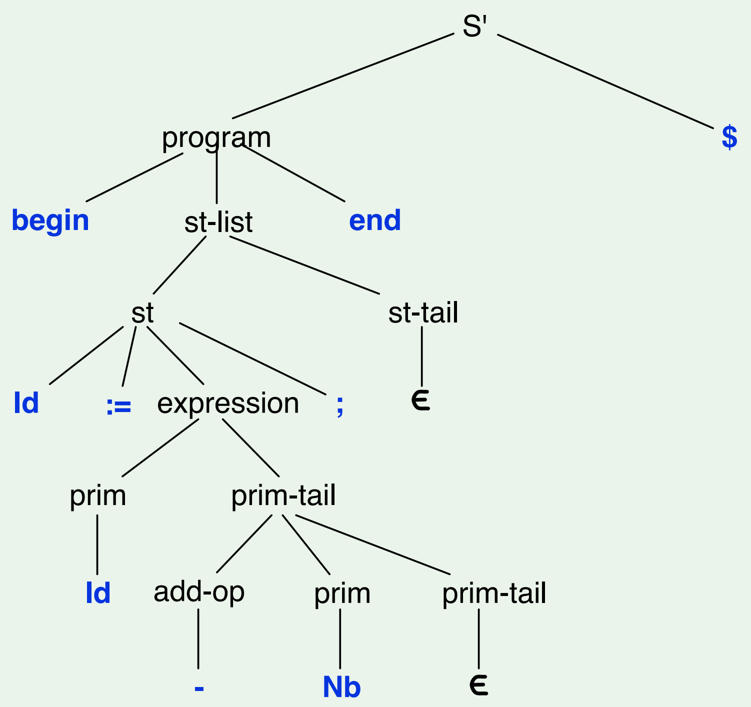
PUB Cours-Librairie, av. P. Héger 42, B-1000 Bruxelles / INFO-F-403_A

Example (Grammar of a very simple language)

Rules	Production rules		
0	*S'*	$\rightarrow$	*program* **$**
1	*program*	$\rightarrow$	**begin** *st-list* **end**
2	*st-list*	$\rightarrow$	*st st-tail*
3	*st-tail*	$\rightarrow$	*st st-tail*
4	*st-tail*	$\rightarrow$	ϵ
5	*st*	$\rightarrow$	**Id :=** *expression* **;**
6	*st*	$\rightarrow$	**read (** *id-list* **) ;**
7	*st*	$\rightarrow$	**write(** *expr-list* **);**
8	*id-list*	$\rightarrow$	**Id** *id-tail*
9	*id-tail*	$\rightarrow$	**, Id** *id-tail*
10	*id-tail*	$\rightarrow$	ϵ
11	*expr-list*	$\rightarrow$	*expression expr-tail*
12	*expr-tail*	$\rightarrow$	**,** *expression expr-tail*
13	*expr-tail*	$\rightarrow$	ϵ
14	*expression*	$\rightarrow$	*prim prim-tail*
15	*prim-tail*	$\rightarrow$	*add-op prim prim-tail*
16	*prim-tail*	$\rightarrow$	ϵ
17	*prim*	$\rightarrow$	**(** *expression* **)**
18	*prim*	$\rightarrow$	**Id**
19	*prim*	$\rightarrow$	**Nb**
20 | 21	*add-op*	$\rightarrow$	**+** | **-**

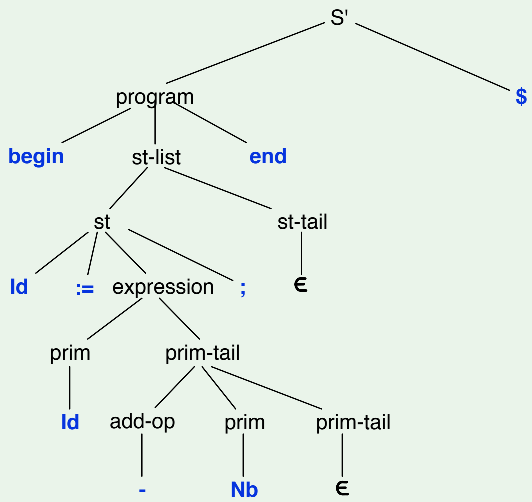
Example of syntactic tree

Example (Syntactic tree corresponding to **begin Id := Id - Nb ; end** $)



Example of syntactic tree and of left-most / right-most derivation

Example (Left-most / right-most derivation corresponding to the syntactic tree)



Left-most derivation $S'_G \overset{*}{\Rightarrow}$ **begin Id := Id - Nb ; end** $:

0 1 2 5 14 18 15 21 19 16 4

Right-most derivation $S' \overset{*}{\Rightarrow}_G$ **begin Id := Id - Nb ; end** $:

0 1 2 4 5 14 15 16 19 21 18

Left-most derivation

Example (Complete corresponding left-most derivation)

Rule	longest prefix $\in T^*$	last part of the sentential form	
		S'	$\Rightarrow$
0		program $	$\Rightarrow$
1	**begin**	st-list **end**$	$\Rightarrow$
2	**begin**	st st-tail **end**$	$\Rightarrow$
5	**begin Id :=**	expression **;** st-tail **end**$	$\Rightarrow$
14	**begin Id :=**	prim prim-tail **;** st-tail **end**$	$\Rightarrow$
18	**begin Id := Id**	prim-tail **;** st-tail **end**$	$\Rightarrow$
15	**begin Id := Id**	add-op prim prim-tail **;** st-tail **end**$	$\Rightarrow$
21	**begin Id := Id -**	prim prim-tail **;** st-tail **end**$	$\Rightarrow$
19	**begin Id := Id - Nb**	prim-tail **;** st-tail **end**$	$\Rightarrow$
16	**begin Id := Id - Nb ;**	st-tail **end**$	$\Rightarrow$
4	**begin Id := Id - Nb ; end** $		

Right-most derivation

Example (Complete corresponding right-most derivation)

Rule	sentential form	
	S'	⇒
0	program $	⇒
1	**begin** st-list **end** $	⇒
2	**begin** st st-tail **end** $	⇒
4	**begin** st **end** $	⇒
5	**begin Id :=** expression **; end** $	⇒
14	**begin Id :=** prim prim-tail **; end** $	⇒
15	**begin Id :=** prim add-op prim prim-tail **; end** $	⇒
16	**begin Id :=** prim add-op prim **; end** $	⇒
19	**begin Id :=** prim add-op **Nb ; end** $	⇒
21	**begin Id :=** prim **- Nb ; end** $	⇒
18	**begin Id := Id - Nb ; end** $	

General structure of a parser

Parser = Algorithm to recognize the structure of the program and build the corresponding syntactic tree

Since the language to recognize is context-free, a parser will work as a Pushdown automaton (PDA)
We will see two big classes of parsers:

- The top-down parser
- The bottom-up parser

General structure of a parser

A parser must recognize the string and produce an output, which can be:

- the corresponding syntactic tree
- calls to the semantic analyzer and code generator
- ...

Remark:

In what follows, the output = the sequence of production rules used in the derivation.
If we also know that it is a left-most (resp. right-most) derivation, this sequence identifies the derivation and the corresponding tree (and allows to easily build it).

Outline

Reminder: building a PDA equivalent to a given CFG

From a CFG G one can build a PDA M with $L(G) = N(M)$

Principle of the construction: With the CFG $G = \langle V, T, P, S\rangle$, one builds a PDA M with one state which simulates the left-most derivations of G

$$\begin{array}{l} P = \langle \{q\}, T, V \cup T, \delta, q, S, \emptyset\rangle \text{ with} \\ \forall A \rightarrow X_1X_2\ldots X_k \in P : \langle q, X_1X_2\ldots X_k\rangle \in \delta(q, \epsilon, A) \\ \forall a \in T : \delta(q, a, a) = \{\langle q, \epsilon\rangle\} \end{array}$$

- Initially the start symbol S is on the stack
- Every variable A on top of the stack with $A \rightarrow X_1X_2\ldots X_k \in P$ can be replaced by its right part $X_1X_2\ldots X_k$ with X_1 on top of the stack
- Each terminal on top of the stack which is equal to the next symbol of the input is matched with the input (the input is read and the symbol is popped from the stack)
- At the end, if the stack is empty, the string is accepted

In the construction, the rule $S' \rightarrow S\$$ was not added

Outline of a top-down parser

Outline of a top-down parser: PDA with one state and with output

Initially S' is on the stack
The PDA can do 4 types of actions:

- Produce: the variable A on top of the stack is replaced by the right part of one of its rules (numbered i) and the number i is written on the output
- Match: the terminal a on top of the stack corresponds to the next input terminal; this terminal is popped and we go to the next input
- Accept: Corresponds to a Match of the terminal \$: the terminal on the top of the stack is \$ and corresponds to the next input terminal; the analysis terminates with success
- Error: If no Match nor Produce is possible

Left-most derivation

Example (Complete corresponding left-most derivation)

On the stack	Remaining input	Action	Output
S'⊣	**begin Id := Id - Nb ; end** $	P0	ϵ
program $⊣	**begin Id := Id - Nb ; end** $	P1	0
begin st-list **end**$⊣	**begin Id := Id - Nb ; end** $	M	0 1
st-list **end**$⊣	**Id := Id - Nb ; end** $	P2	0 1
st st-tail **end**$⊣	**Id := Id - Nb ; end** $	P5	0 1 2
Id := expression **;** st-tail **end**$⊣	**Id := Id - Nb ; end** $	M	0 1 2 5
:= expression **;** st-tail **end**$⊣	**:= Id - Nb ; end** $	M	0 1 2 5
expression **;** st-tail **end**$⊣	**Id - Nb ; end** $	P14	0 1 2 5
prim prim-tail **;** st-tail **end**$⊣	**Id - Nb ; end** $	P18	0 1 2 5 14
Id prim-tail **;** st-tail **end**$⊣	**Id - Nb ; end** $	M	0 1 2 5 14 18
prim-tail **;** st-tail **end**$⊣	**- Nb ; end** $	P15	0 1 2 5 14 18
add-op prim prim-tail **;** st-tail **end**$⊣	**- Nb ; end** $	P21	0 1 2 5 14 18 15
- prim prim-tail **;** st-tail **end**$⊣	**- Nb ; end** $	M	0 1 2 5 14 18 15 21
prim prim-tail **;** st-tail **end**$⊣	**Nb ; end** $	P19	0 1 2 5 14 18 15 21
Nb prim-tail **;** st-tail **end**$⊣	**Nb ; end** $	M	0 1 2 5 14 18 15 21 19
prim-tail **;** st-tail **end**$⊣	**; end** $	P16	0 1 2 5 14 18 15 21 19
; st-tail **end**$⊣	**; end** $	M	0 1 2 5 14 18 15 21 19 16
st-tail **end**$⊣	**end** $	P4	0 1 2 5 14 18 15 21 19 16
end $⊣	**end** $	M	0 1 2 5 14 18 15 21 19 16 4
$⊣	$	A	0 1 2 5 14 18 15 21 19 16 4

where:

- Pi : Produce with rule *i*
- M : Match
- A : Accept (corresponds to a Match of the $ symbol)
- E : Error (or blocking which requests a backtracking) (not in this example)

Points to improve in the outline of the top-down parser

Criticism of the top-down parser outline

- As such, this parser is extremely inefficient since it must do backtracking to explore all the possibilities.
- In this kind of parser, a choice must be made when several “Produces” are possible.
- If several choices are possible and no criteria in the method allow to select the good one, one will talk about Produce/Produce conflicts.
- Without guide when the choice must be done, one possibly has to explore all the possible Produces: the parser could therefore take an exponential time (typically in the length of the input) which is unacceptable!
- We will see efficient top-down parsing techniques in the next chapter.

Outline

Outline of a bottom-up parser

Outline of a bottom-up parser

PDA with one state and with output.
We start from the input string and build the tree bottom-up. In order to do so, two actions are available:

1. "Shift": shift the input symbols on the stack until identification of a right-hand part α (handle) of the rule $A \rightarrow \alpha$
2. "Reduction": replacement of α by A [a]

Initially the stack is empty.
The PDA can do 4 kinds of actions :

- Shift: reading of an input symbol and push of this symbol on the stack
- Reduce: the top of the stack α corresponding to the handle (the right part of a rule number i : $A \rightarrow \alpha$), is replaced by A on the stack and the number i of the used rule is written on the output
- Accept: corresponds to a Reduce of the rule $S' \rightarrow S\$$ (which shows that the input has been completely read and analyzed); the analysis is completed successfully
- Error: If no Shift nor Reduce is possible

[a] Formally corresponds to $|\alpha|$ pops followed by a push of A

Outline of a bottom-up parser

Remark:

- One can see that the analysis corresponds to a reverse order right-most analysis: one starts from the string and goes up in the derivation back to the start symbol. Analysis is done in reverse order since the input is read from left to right.
- The output will be built in reverse order (each new output is put before all what has been produced before) to obtain this right-most derivation.

Right-most derivation

Example (Corresponding complete right-most derivation)

On the stack	Remaining input	Act	Output
⊢	**begin Id := Id - Nb ; end** $	S	ϵ
⊢ **begin**	**Id := Id - Nb ; end** $	S	ϵ
⊢ **begin Id**	**:= Id - Nb ; end** $	S	ϵ
⊢ **begin Id :=**	**Id - Nb ; end** $	S	ϵ
⊢ **begin Id := Id**	**- Nb ; end** $	R18	ϵ
⊢ **begin Id :=** prim	**- Nb ; end** $	S	18
⊢ **begin Id :=** prim **-**	**Nb ; end** $	S	18
⊢ **begin Id :=** prim **-**	**Nb ; end** $	R21	18
⊢ **begin Id :=** prim add-op	**Nb ; end** $	S	21 18
⊢ **begin Id :=** prim add-op **Nb**	**; end** $	R19	21 18
⊢ **begin Id :=** prim add-op prim	**; end** $	R16	19 21 18
⊢ **begin Id :=** prim add-op prim prim-tail	**; end** $	R15	16 19 21 18
⊢ **begin Id :=** prim prim-tail	**; end** $	R14	15 16 19 21 18
⊢ **begin Id :=** expression	**; end** $	S	14 15 16 19 21 18
⊢ **begin Id :=** expression **;**	**end** $	R5	14 15 16 19 21 18
⊢ **begin** st	**end** $	R4	5 14 15 16 19 21 18
⊢ **begin** st st-tail	**end** $	R2	4 5 14 15 16 19 21 18
⊢ **begin** st-list	**end** $	S	2 4 5 14 15 16 19 21 18
⊢ **begin** st-list **end**	$	R1	2 4 5 14 15 16 19 21 18
⊢ program	$	S	1 2 4 5 14 15 16 19 21 18
⊢ program $	ϵ	A	1 2 4 5 14 15 16 19 21 18
⊢ S'	ϵ		0 1 2 4 5 14 15 16 19 21 18

where:

S : Shift

Ri : Reduce with the rule *i*

A : Accept (corresponds to a Reduce with the rule 0)

E : Error (or blocking which requests a backtracking)

Points to improve in the outline of the bottom-up parser

Criticism of the outline of the bottom-up parser

- As such, this parser is extremely inefficient since it must backtrack to explore all the possibilities.
- In this kind of parser, a choice must occur when both a “Reduce” and “Shift” can be done, or when several “Reduces” are possible.
- If several choices are possible and no criteria in the method allow to choose, one can talk of Shift/Reduce or Reduce/Reduce conflicts.
- Without guide when the choice must be done, possibly every possible Shift and Reduce must be tried: the parser could therefore take an exponential time (typically in the length of the input) which is unacceptable!
- We will show efficient bottom-up parsing techniques in a later chapter.

Chapter 9: LL(k) parsers

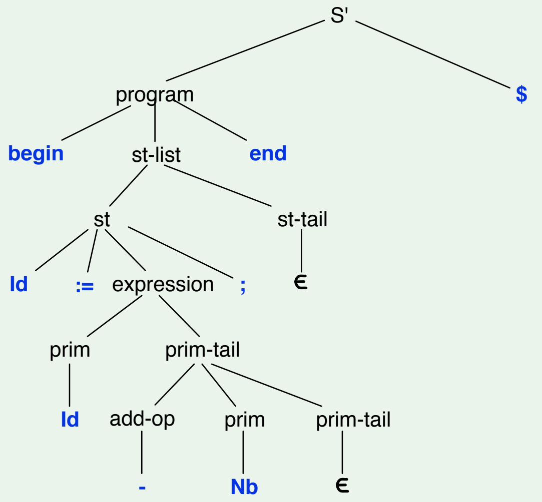
Outline

Example (Grammar of a very simple language)

Rules	*Production rules*		
0	*S'*	→	*program* **$**
1	*program*	→	**begin** *st-list* **end**
2	*st-list*	→	*st st-tail*
3	*st-tail*	→	*st st-tail*
4	*st-tail*	→	ϵ
5	*st*	→	**Id :=** *expression* **;**
6	*st*	→	**read (** *id-list* **) ;**
7	*st*	→	**write(** *expr-list* **);**
8	*id-list*	→	**Id** *id-tail*
9	*id-tail*	→	**, Id** *id-tail*
10	*id-tail*	→	ϵ
11	*expr-list*	→	*expression expr-tail*
12	*expr-tail*	→	**,** *expression expr-tail*
13	*expr-tail*	→	ϵ
14	*expression*	→	*prim prim-tail*
15	*prim-tail*	→	*add-op prim prim-tail*
16	*prim-tail*	→	ϵ
17	*prim*	→	**(** *expression* **)**
18	*prim*	→	**Id**
19	*prim*	→	**Nb**
20 | 21	*add-op*	→	**+ | -**

Example of syntactic tree and of left-most derivation

Example (Left-most derivation corresponding to the syntactic tree)



Left-most derivation $S'_G \stackrel{*}{\Rightarrow}$ **begin Id := Id - Nb ; end** $:

0 1 2 5 14 18 15 21 19 16 4

Left-most derivation

Example (Complete corresponding left-most derivation)

Rule	longest prefix $\in T^*$	last part of the sentential form	
		S'	$\Rightarrow$
0		program $	$\Rightarrow$
1	**begin**	st-list **end**$	$\Rightarrow$
2	**begin**	st st-tail **end**$	$\Rightarrow$
5	**begin Id :=**	expression **;** st-tail **end**$	$\Rightarrow$
14	**begin Id :=**	prim prim-tail **;** st-tail **end**$	$\Rightarrow$
18	**begin Id := Id**	prim-tail **;** st-tail **end**$	$\Rightarrow$
15	**begin Id := Id**	add-op prim prim-tail **;** st-tail **end**$	$\Rightarrow$
21	**begin Id := Id -**	prim prim-tail **;** st-tail **end**$	$\Rightarrow$
19	**begin Id := Id - Nb**	prim-tail **;** st-tail **end**$	$\Rightarrow$
16	**begin Id := Id - Nb ;**	st-tail **end**$	$\Rightarrow$
4	**begin Id := Id - Nb ; end** $		

PUB Cours-Librairie, av. P. Héger 42, B-1000 Bruxelles / INFO-F-403_A

Outline of a top-down parser

Outline of a top-down parser: PDA with one state and with output

Initially S' is on the stack
The PDA can do 4 types of actions:

- Produce: the variable A on top of the stack is replaced by the right part of one of its rules (numbered i) and the number i is written on the output
- Match: the terminal a on top of the stack corresponds to the next input terminal; this terminal is popped and we go to the next input
- Accept: Corresponds to a Match of the terminal $: the terminal on the top of the stack is $ and corresponds to the next input terminal; the analysis terminates with success
- Error: If no Match nor Produce is possible

Left-most derivation

Example (Complete corresponding left-most derivation)

On the stack	Remaining input	Action	Output
S'⊣	**begin Id := Id - Nb ; end** $	P0	ϵ
program $⊣	**begin Id := Id - Nb ; end** $	P1	0
begin st-list **end**$⊣	**begin Id := Id - Nb ; end** $	M	0 1
st-list **end**$⊣	**Id := Id - Nb ; end** $	P2	0 1
st st-tail **end**$⊣	**Id := Id - Nb ; end** $	P5	0 1 2
Id := expression **;** st-tail **end**$⊣	**Id := Id - Nb ; end** $	M	0 1 2 5
:= expression **;** st-tail **end**$⊣	**:= Id - Nb ; end** $	M	0 1 2 5
expression **;** st-tail **end**$⊣	**Id - Nb ; end** $	P14	0 1 2 5
prim prim-tail **;** st-tail **end**$⊣	**Id - Nb ; end** $	P18	0 1 2 5 14
Id prim-tail **;** st-tail **end**$⊣	**Id - Nb ; end** $	M	0 1 2 5 14 18
prim-tail **;** st-tail **end**$⊣	**- Nb ; end** $	P15	0 1 2 5 14 18
add-op prim prim-tail **;** st-tail **end**$⊣	**- Nb ; end** $	P21	0 1 2 5 14 18 15
- prim prim-tail **;** st-tail **end**$⊣	**- Nb ; end** $	M	0 1 2 5 14 18 15 21
prim prim-tail **;** st-tail **end**$⊣	**Nb ; end** $	P19	0 1 2 5 14 18 15 21
Nb prim-tail **;** st-tail **end**$⊣	**Nb ; end** $	M	0 1 2 5 14 18 15 21 19
prim-tail **;** st-tail **end**$⊣	**; end** $	P16	0 1 2 5 14 18 15 21 19
; st-tail **end**$⊣	**; end** $	M	0 1 2 5 14 18 15 21 19 16
st-tail **end**$⊣	**end** $	P4	0 1 2 5 14 18 15 21 19 16
end $⊣	**end** $	M	0 1 2 5 14 18 15 21 19 16 4
$⊣	$	A	0 1 2 5 14 18 15 21 19 16 4

where:

Pi : Produce with rule i

M : Match

A : Accept (corresponds to a Match of the $ symbol)

E : Error (or blocking which requests a backtracking) (not in the example)

Points to improve in the outline of the top-down parser

Criticism of the top-down parser outline

- As such, this parser is extremely inefficient since it must do backtracking to explore all the possibilities.
- In this kind of parser, a choice must be made when several "Produces" are possible.
- If several choices are possible and no criteria in the method allow to select the good one, one will talk about Produce/Produce conflicts.
- Without guide when the choice must be done, one possibly has to explore all the possible Produces: the parser could therefore take an exponential time (typically in the length of the input) which is unacceptable!
- We will see efficient top-down parsing techniques in this chapter.

Outline

Introduction to predictive parsers

Motivation

- During top-down parsing, the choices the parser must make occur when the action to achieve is a Produce and the concerned variable (on top of the stack) has several rules.
- In that case, the remaining (not yet matched) input can be used as a "guide".
- In the example, if a produce must be done with the variable *st*, depending on the fact the remaining input is **Id**, **read** or **write**, it is clear that the parser must make a Produce 5, 6 or 7

Introduction to predictive parsers

Predictive parser

The $LL(k)$ parsers are predictive and have k look-ahead symbols (k is a natural number): when a variable is on top of the stack, the produce done will depend on:

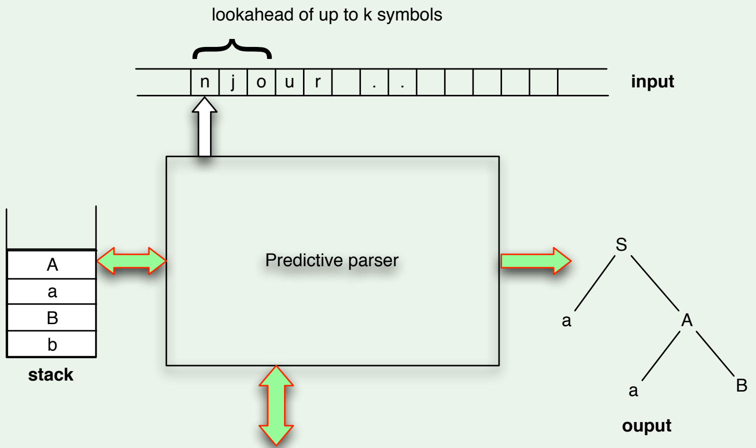
1. the variable
2. the (at most) k first input symbols

Example (Action table M)

An $LL(k)$ parser has a 2-dimensional table M where $M[A, u]$ determines the production rule to use when A is the variable to develop (on top of stack) and u is the look-ahead.

- For an $LL(1)$ parser the look-ahead is a terminal symbol a.
- For an $LL(k)$ parser, with k an integer bigger than 1, we will have $M[A, u]$, with u a string of size up to k symbols (limited by the final \$)

Example ($LL(k)$ predictive parsers)



M

	a	b	o	n	j	...
S	1		4			
A	2			6		
C						
...						

Action table M

Remark: Every unexpected combination in **M** gives an error during parsing

Algorithm : outline of an $LL(k)$ predictive parser for $G = \langle V, T, P, S' \rangle$ and rules of the form $A \rightarrow \alpha_i$

The table M is assumed to have already been built

Parser-LL-k():=

Initially: $Push(S')$

While (no Error nor Accept)

$X \leftarrow Top()$

$u \leftarrow Look\text{-}ahead$

If ($X = A \in V$ and $M[A, u] = i$) : Produce(i);

Else if ($X = a \neq \$ \in T$) and $u = av$ ($v \in T^*$) : Match();

Else if ($X = u = \$$) : Accept();

Else : /* not expected */ Error();

FProc

Produce(i) := Pop(); Push(α_i); Endproc

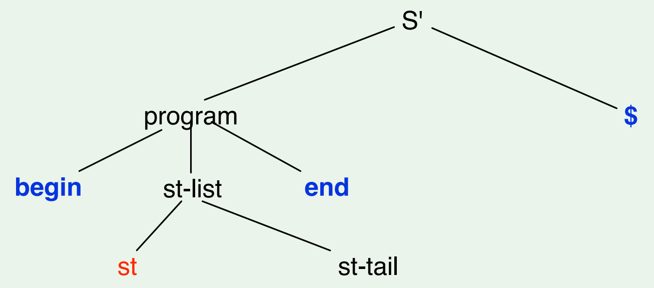
Match() := Pop(); Shifts to the next input; Endproc

Accept() := Informs of the success of parsing; Endproc

Error() := Informs of an error during parsing; Endproc

How can we predict (i.e. fill M)?

Example (1: a step of top-down parsing for $S'_G \stackrel{*}{\Rightarrow}$ begin Id := Id - Nb ; end $)

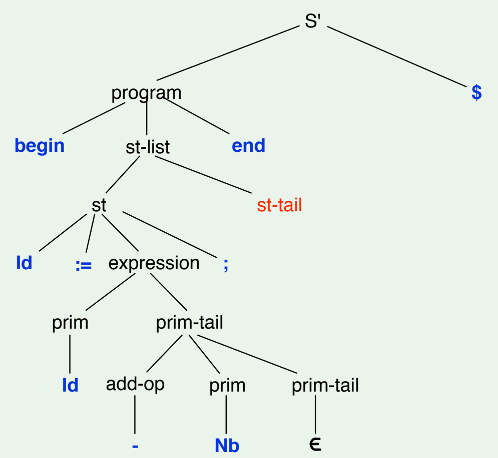


- Left-most derivation already achieved $S'_G \stackrel{*}{\Rightarrow}$ **begin** *st st-tail* **end** $: **0 1 2**
- Remaining input (not yet matched): **Id := Id - Nb ; end** $

$\Rightarrow$ The Produce *st* $\rightarrow$ Id := *expression* ; must be done which starts with Id and corresponds to the input

How can we predict (i.e. fill M)?

Example (2: a step of top-down parsing for $S'_G \stackrel{*}{\Rightarrow}$ begin Id := Id - Nb ; end $)



- Left-most derivation already achieved $S'_G \stackrel{*}{\Rightarrow}$ **begin Id := Id - Nb ;** *st-tail* **end** $: **0 1 2 5 14 18 15 21 19 16**
- Remaining input (not yet matched): **end** $

$\Rightarrow$ The Produce *st-tail* $\rightarrow \epsilon$ must be done since what follows *st-tail* starts with end which corresponds to the input

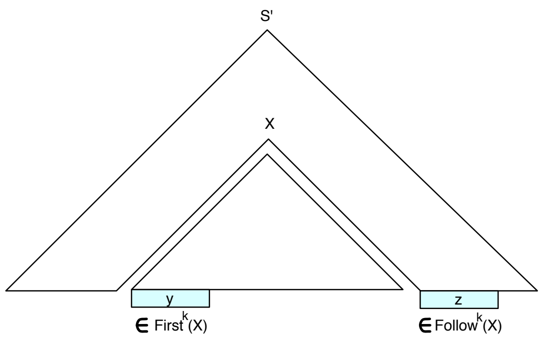
$First^k(X) - Follow(^k(X)$

For a symbol X, we have to know :

- $First^k(X)$: the set of strings of terminals of maximum length but limited to k symbols which can start a string generated from X
- $Follow^k(X)$: the set of strings of terminals of maximum length but limited to k symbols which can follow a string generated from X

In the following figure we have:

- $y \in First^k(X)$
- $z \in Follow^k(X)$:



$First^k$ - $Follow^k$

Construction of the action table M uses the $First^k$ and $Follow^k$ functions defined for a given CFG $G = \langle V, T, P, S' \rangle$:

Definition ($First^k(\alpha)$. For the CFG G, a positive integer k and $\alpha \in (T \cup V)^*$)

$First^k(\alpha)$ is the set of terminal strings of maximum length but limited to k symbols which can start a string generated from α

Mathematically:

$$First^k(\alpha) = \Big\{ w \in T^{\leq k} \mid \exists x \in T^* : \alpha \overset{*}{\Rightarrow} wx \wedge \Big((|w| = k) \vee (|w| < k \wedge x = \epsilon) \Big) \Big\}$$

where : $T^{\leq k} = \bigcup_{i=0}^{k} T^i$

$First^1(\alpha)$ (or $First(\alpha)$)

Definition ($First^1(\alpha)$ (or $First(\alpha)$))

$First^1(\alpha)$, also simply denoted $First(\alpha)$, is the set of terminal symbols which can start a string generated from α, union ϵ if α can generate ϵ

Mathematically:

$$First(\alpha) = \{a \in T \mid \exists x \in T^* : \alpha \stackrel{*}{\Rightarrow} ax\} \cup \{\epsilon \mid \alpha \stackrel{*}{\Rightarrow} \epsilon\}$$

Computation of $First^k(\alpha)$

$First^k(\alpha)$ with $\alpha = X_1 X_2 \ldots X_n$

$$First^k(\alpha) = First^k(X_1) \oplus^k First^k(X_2) \oplus^k \cdots \oplus^k First^k(X_n)$$

with

$$L_1 \oplus^k L_2 = \Big\{ w \in T^{\leq k} \mid \exists x \in T^*, y \in L_1, z \in L_2 : wx = yz \wedge \big((|w| = k) \vee (|w| < k \wedge x = \epsilon)\big) \Big\}$$

Computation of $First^k(X)$

Computation of $First^k(X)$ with $X \in (T \cup V)$

Greedy algorithm: one increases the sets $First^k(X)$ until stabilisation.

Basis:

$$\forall a \in T : First^k(a) = \{a\}$$
$$\forall A \in V : First^k(A) = \emptyset$$

Induction: loop until stabilisation:

$$\forall A \in V : First^k(A) \stackrel{\cup}{\Leftarrow} \{x \in T^* \mid A \rightarrow Y_1 Y_2 \ldots Y_n \wedge x \in First^k(Y_1) \oplus^k First^k(Y_2) \oplus^k \cdots \oplus^k First^k(Y_n)\}$$

Computation of $First(X)$

Example (of computation of $First(A)$ ($\forall A \in V$) with $G = \langle V, T, P, S' \rangle$)

where P contains:

- $S' \rightarrow E\$$
- $E \rightarrow TE'$
- $E' \rightarrow +TE' \mid \epsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT' \mid \epsilon$
- $F \rightarrow (E) \mid id$

Initially

1. $\forall A \in V : First(A) = \emptyset$

Step 1:

1. $First(E') \stackrel{\cup}{\Leftarrow} \{\epsilon\}$
2. $First(T') \stackrel{\cup}{\Leftarrow} \{\epsilon\}$
3. $First(F) \stackrel{\cup}{\Leftarrow} \{id\}$

Step 2:

1. $First(T) \stackrel{\cup}{\Leftarrow} \{id\}$
2. $First(T') \stackrel{\cup}{\Leftarrow} \{*\}$

Step 3:

1. $First(E) \stackrel{\cup}{\Leftarrow} \{id\}$
2. $First(E') \stackrel{\cup}{\Leftarrow} \{+\}$

Step 4:

1. $First(S') \stackrel{\cup}{\Leftarrow} \{id\}$
2. $First(F) \stackrel{\cup}{\Leftarrow} \{(\}$

Step 5:

1. $First(T) \stackrel{\cup}{\Leftarrow} \{(\}$

Step 6:

1. $First(E) \stackrel{\cup}{\Leftarrow} \{(\}$

Step 7:

1. $First(S') \stackrel{\cup}{\Leftarrow} \{(\}$

Step 8: stabilisation

Example (of computation of $First(A)$ ($\forall A \in V$) with $G = \langle V, T, P, S' \rangle$)

where P contains:

- $S' \rightarrow E\$$
- $E \rightarrow TE'$
- $E' \rightarrow +TE' \mid \epsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT' \mid \epsilon$
- $F \rightarrow (E) \mid id$

- $First(S') = \{id, (\ \}$
- $First(E) = \{id, (\ \}$
- $First(E') = \{+, \epsilon\}$
- $First(T) = \{id, (\ \}$
- $First(T') = \{*, \epsilon\}$
- $First(F) = \{id, (\ \}$

Example (of computation of $First^2(A)$ ($\forall A \in V$) with the same grammar G)

Initially

1. $\forall A \in V : First^2(A) = \emptyset$

Step 1:

1. $First^2(E') \stackrel{\cup}{\Leftarrow} \{\epsilon\}$
2. $First^2(T') \stackrel{\cup}{\Leftarrow} \{\epsilon\}$
3. $First^2(F) \stackrel{\cup}{\Leftarrow} \{id\}$

Step 2:

1. $First^2(T) \stackrel{\cup}{\Leftarrow} \{id\}$
2. $First^2(T') \stackrel{\cup}{\Leftarrow} \{*id\}$

Step 3:

1. $First^2(E) \stackrel{\cup}{\Leftarrow} \{id\}$
2. $First^2(E') \stackrel{\cup}{\Leftarrow} \{+id\}$
3. $First^2(T) \stackrel{\cup}{\Leftarrow} \{id*\}$

Step 4:

1. $First^2(S') \stackrel{\cup}{\Leftarrow} \{id\$\}$
2. $First^2(E) \stackrel{\cup}{\Leftarrow} \{id+, id*\}$
3. $First^2(F) \stackrel{\cup}{\Leftarrow} \{(id\}$

Step 5:

1. $First^2(S') \stackrel{\cup}{\Leftarrow} \{id+, id*\}$
2. $First^2(T) \stackrel{\cup}{\Leftarrow} \{(id\}$
3. $First^2(T') \stackrel{\cup}{\Leftarrow} \{*(\}$

Step 6:

1. $First^2(E) \stackrel{\cup}{\Leftarrow} \{(id\}$
2. $First^2(E') \stackrel{\cup}{\Leftarrow} \{+(\}$

Step 7:

1. $First^2(S') \stackrel{\cup}{\Leftarrow} \{(id\}$
2. $First^2(F) \stackrel{\cup}{\Leftarrow} \{((\}$

Step 8:

1. $First^2(T) \stackrel{\cup}{\Leftarrow} \{((\}$

Step 9:

1. $First^2(E) \stackrel{\cup}{\Leftarrow} \{((\}$

Step 10:

1. $First^2(S') \stackrel{\cup}{\Leftarrow} \{((\}$

Step 11: stabilisation

Computation of $First^2(X)$

Example (of computation of $First^2(A)$ ($\forall A \in V$) with $G = \langle V, T, P, S' \rangle$)

with the rules:

- $S' \rightarrow E\$$
- $E \rightarrow TE'$
- $E' \rightarrow +TE' \mid \epsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT' \mid \epsilon$
- $F \rightarrow (E) \mid id$

- $First^2(S') = \{id\$, id+, id*, (id, ((\ \}$
- $First^2(E) = \{id, id+, id*, (id, ((\}$
- $First^2(E') = \{\epsilon, +id, +(\}$
- $First^2(T) = \{id, id*, (id, ((\}$
- $First^2(T') = \{\epsilon, *id, *(\}$
- $First^2(F) = \{id, (id, ((\}$

$Follow^k$

Definition ($Follow^k(\beta)$ for the CFG G, a positive integer k and $\beta \in (T \cup V)^*$)

$Follow^k(\beta)$ is the set of terminal strings of maximum length but limited to k symbols which can follow a string generated from β

Mathematically:

$$Follow^k(\beta) = \{w \in T^{\leq k} \mid \exists \alpha, \gamma \in (T \cup V)^* : S' \stackrel{*}{\Rightarrow} \alpha\beta\gamma \wedge w \in First^k(\gamma)\}$$

Computation of $Follow^k(X)$

We only need to compute the Follow for variables.

Computation of $Follow^k(A)$ with $A \in (T \cup V)$

Greedy algorithm: we increase the sets $Follow^k(B)$; initially empty, until stabilisation.

Basis:

$$\forall A \in V : Follow^k(A) = \emptyset$$

Induction:

$$\forall A \in V, A \rightarrow \alpha B \beta \in P \ (B \in V; \alpha, \beta \in (V \cup T)^*) :$$
$$Follow^k(B) \stackrel{\cup}{\Leftarrow} First^k(\beta) \oplus^k Follow^k(A)$$

Until stabilisation.

Principles of top-down parsing
Predictive parsers - $First^k$ - $Follow^k$
$LL(k)$ CFGs
$LL(1)$ parsers
Strongly $LL(k)$ parsers ($k > 1$)
$LL(k)$ parsers ($k > 1$)
Error handling and resynchronization
Recursive $LL(k)$ parsers

Computation of $Follow(A)$

Example (of computation of $Follow(A)$ ($\forall A \in V$) with $G = \langle V, T, P, S' \rangle$)

with rules:

- $S' \rightarrow E\$$
- $E \rightarrow TE'$
- $E' \rightarrow +TE' \mid \epsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT' \mid \epsilon$
- $F \rightarrow (E) \mid id$

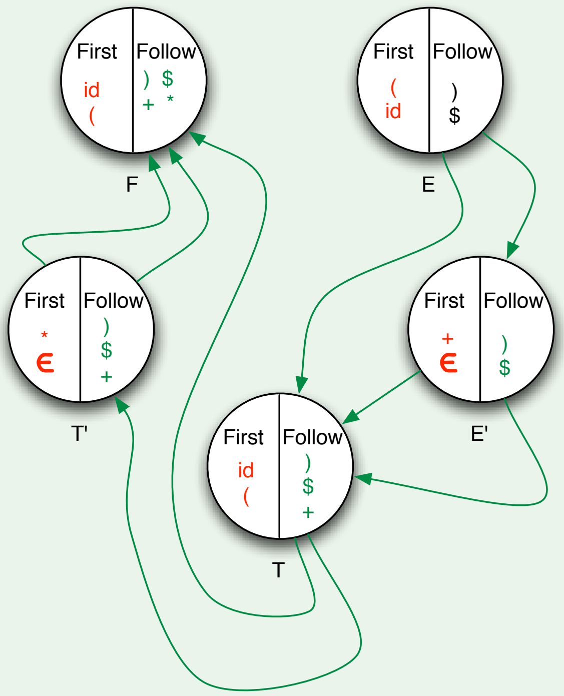
We obtain, using the algorithm (see figure):

- $Follow(S') = \{\epsilon\}$
- $Follow(E) = \{), \$\}$
- $Follow(E') = \{), \$\}$
- $Follow(T) = \{), \$, +\}$
- $Follow(T') = \{), \$, +\}$
- $Follow(F) = \{), \$, +, *\}$

PUB Cours-Librairie, av. P. Héger 42, B-1000 Bruxelles / INFO-F-403_A

Computation of $Follow(A)$

Example (of computation of $Follow(A)$ ($\forall A \in V$) with $G = \langle V, T, P, S' \rangle$)



Outline

$LL(k)$ CFGs

$LL(k)$ means

- Left scanning
- Leftmost derivation
- k look-ahead symbols

Definition (The CFG $G = \langle V, T, P, S' \rangle$ is $LL(k)$ (k a fixed natural number) if)

$\forall w, x_1, x_2 \in T^*$:

- $S' \; _G\overset{*}{\Rightarrow} wA\gamma \; _G\Rightarrow w\alpha_1\gamma \; _G\overset{*}{\Rightarrow} wx_1$
- $S' \; _G\overset{*}{\Rightarrow} wA\gamma \; _G\Rightarrow w\alpha_2\gamma \; _G\overset{*}{\Rightarrow} wx_2$
- $First^k(x_1) = First^k(x_2)$

$\Rightarrow \alpha_1 = \alpha_2$

Which implies that for the sentential form $wA\gamma$, if one knows $First^k(x_1)$ (what remains to be analyzed after w), one can determine with the look-ahead of k symbols, the rule $A \rightarrow \alpha_i$ to apply

Problem

In theory, to determine if G (if we suppose it generated an infinite language) is $LL(k)$, one must verify an infinite number of conditions

Property 1 on $LL(k)$ CFGs

Theorem (1 on $LL(k)$ CFGs)

$$\text{A CFG } G \text{ is } LL(k)$$
$$\Longleftrightarrow$$
$$\forall A \in V : S' \overset{*}{\Rightarrow} wA\gamma, A \rightarrow \alpha_i \in P \; (i = 1, 2 : \alpha_1 \neq \alpha_2) :$$
$$First^k(\alpha_1\gamma) \cap First^k(\alpha_2\gamma) = \emptyset$$

Problem

In theory, to determine if the property is satisfied on G (if we suppose the produced language is infinite), an infinite number of conditions must be verified: since only the k first symbols interest us, one can find an algorithm which does that in a finite time (see below))

PUB Cours-Librairie, av. P. Héger 42, B-1000 Bruxelles / INFO-F-403_A

Property 1 on $LL(k)$ CFGs

Theorem (1 on $LL(k)$ CFGs)

A CFG G is LL(k)

$$\Longleftrightarrow$$

$$\forall S' \stackrel{*}{\Rightarrow} wA\gamma, A \rightarrow \alpha_i \in P\ (i = 1, 2 : \alpha_1 \neq \alpha_2) :$$
$$First^k(\alpha_1\gamma) \cap First^k(\alpha_2\gamma) = \emptyset$$

Proof: *By contradiction*

$\Rightarrow$ *: One supposes G is $LL(k)$ and the property is not verified.*

- $\exists S' \stackrel{*}{\Rightarrow} wA\gamma, A \rightarrow \alpha_i \in P\ (i = 1, 2 : \alpha_1 \neq \alpha_2) :$ $First^k(\alpha_1\gamma) \cap First^k(\alpha_2\gamma) \neq \emptyset$
- *Then* $\exists x_1, x_2$:
- $S'\ {}_G\!\stackrel{*}{\Rightarrow} wA\gamma\ {}_G\!\Rightarrow w\alpha_1\gamma\ {}_G\!\stackrel{*}{\Rightarrow} wx_1$
- $S'\ {}_G\!\stackrel{*}{\Rightarrow} wA\gamma\ {}_G\!\Rightarrow w\alpha_2\gamma\ {}_G\!\stackrel{*}{\Rightarrow} wx_2$
- $First^k(x_1) = First^k(x_2) \wedge \alpha_1 \neq \alpha_2$

Which contradicts that G is $LL(k)$

Property 1 on $LL(k)$ CFGs

Theorem (1 on $LL(k)$ CFGs)

A CFG G is LL(k)

$$\Longleftrightarrow$$

$$\forall S' \stackrel{*}{\Rightarrow} wA\gamma, A \rightarrow \alpha_i \in P\ (i = 1, 2 : \alpha_1 \neq \alpha_2) :$$
$$First^k(\alpha_1\gamma) \cap First^k(\alpha_2\gamma) = \emptyset$$

Proof (cont'd): *By contradiction*

$\Leftarrow$ *: One supposes the property is verified and G is not $LL(k)$*

- $\forall S' \stackrel{*}{\Rightarrow} wA\gamma, A \rightarrow \alpha_i \in P\ (i = 1, 2 : \alpha_1 \neq \alpha_2) :$ $First^k(\alpha_1\gamma) \cap First^k(\alpha_2\gamma) = \emptyset \wedge G$ *is not* $LL(k)$
- *Then since G is not $LL(k)$:* $\exists w, x_1, x_2 \in T^*$:

$$\begin{array}{l} S'\ {}_G\!\stackrel{*}{\Rightarrow} wA\gamma\ {}_G\!\Rightarrow w\alpha_1\gamma\ {}_G\!\stackrel{*}{\Rightarrow} wx_1 \wedge \\ S'\ {}_G\!\stackrel{*}{\Rightarrow} wA\gamma\ {}_G\!\Rightarrow w\alpha_2\gamma\ {}_G\!\stackrel{*}{\Rightarrow} wx_2 \wedge \\ First^k(x_1) = First^k(x_2) \wedge \\ \alpha_1 \neq \alpha_2 \end{array}$$

But

- $First^k(x_1) \in First^k(\alpha_1\gamma)$
- $First^k(x_2) \in First^k(\alpha_2\gamma)$

Which contradicts the property.

Property 2 on $LL(k)$ CFGs

Theorem (on $LL(1)$ CFGs)

$$\text{A CFG } G \text{ is } LL(1)$$
$$\Longleftrightarrow$$
$$\forall A \rightarrow \alpha_i \in P \; (i = 1, 2 : \alpha_1 \neq \alpha_2) :$$
$$First(\alpha_1 Follow(A)) \cap First(\alpha_2 Follow(A)) = \emptyset$$

Advantage

To determine if G is $LL(1)$, it is sufficient to verify a finite number of conditions

Property 3 on $LL(k)$ CFGs

Theorem (2 on $LL(k)$ CFGs)

$$\text{A CFG } G \text{ is } LL(k)$$
$$\Leftarrow$$
$$\forall A \rightarrow \alpha_i \in P \; (i = 1, 2 : \alpha_1 \neq \alpha_2) :$$
$$First^k(\alpha_1 Follow^k(A)) \cap First^k(\alpha_2 Follow^k(A)) = \emptyset$$

Definition (Strongly $LL(k)$ CFG)

A CFG which satisfies the property above is strongly $LL(k)$

$LL(1)$ = strongly $LL(1)$

From the properties on $LL(1)$ languages and the definition of strongly $LL(k)$ languages, one can deduce that every $LL(1)$ language is strongly $LL(1)$
For $k > 1$ one has: strong $LL(k) \Rightarrow LL(k)$

Advantage

To determine if G is strongly $LL(k)$, it is sufficient to verify a finite number of conditions

From $k = 2$: "strongly $LL(k)$" $\subset LL(k)$

Example (of CFG G that is $LL(2)$ but not strongly $LL(2)$)

with rules :

- $S' \rightarrow S\$$
- $S \rightarrow aAa$
- $S \rightarrow bABa$
- $A \rightarrow b$
- $A \rightarrow \epsilon$
- $B \rightarrow b$
- $B \rightarrow c$

G is $LL(2)$

- $S' \stackrel{1}{\Rightarrow} S\$: First^2(aAa\$) \cap First^2(bABa\$) = \emptyset$
- $S' \stackrel{2}{\Rightarrow} aAa\$: First^2(ba\$) \cap First^2(a\$) = \emptyset$
- $S' \stackrel{2}{\Rightarrow} bABa\$: First^2(bBa\$) \cap First^2(Ba\$) = \emptyset$
- $S' \stackrel{3}{\Rightarrow} bbBa\$: First^2(ba\$) \cap First^2(ca\$) = \emptyset$
- $S' \stackrel{3}{\Rightarrow} bBa\$: First^2(ba\$) \cap First^2(ca\$) = \emptyset$

G is not strongly $LL(2)$

- For S : $First^2(aAa\dots) \cap First^2(bABa\dots) = \emptyset$
- For B : $First^2(b\dots) \cap First^2(c\dots) = \emptyset$
- But for A: $First^2(bFollow^2(A)) \cap First^2(\epsilon Follow^2(A)) \neq \emptyset$
  $(\{ba, bb, bc\} \cap \{a\$, ba, ca\} \neq \emptyset)$

Non $LL(k)$ grammar

Every ambiguous CFG G is not $LL(k)$ (for any k)

Proof: straightforward using definitions of ambiguous and $LL(k)$ CFG

Every left-recursive CFG G (where all symbols are useful), is not $LL(1)$

Proof:
If G is left-recursive, on has for some variable A of G (useful by hypothesis),

- $A \rightarrow \alpha \mid \beta$ (hence $First(\beta) \subseteq First(A)$)
- $A \Rightarrow \alpha \stackrel{*}{\Rightarrow} A\gamma$ (hence $First(A) \subseteq First(\alpha)$)

$\Rightarrow First(\beta) \subseteq First(\alpha)$

Hence G is not $LL(1)$

Non $LL(k)$ grammars

Remark:

Generally a left-recursive CFG G is not $LL(k)$ for any k

Every CFG G with 2 rules $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \in P$ et $\alpha \stackrel{*}{\Rightarrow} x \wedge |x| \geq k$ is not $LL(k)$

Cleaning of G

The bigger k is, the more complex the $LL(k)$ parser will be. One tries to have $LL(k)$ CFGs with the smallest possible k (1 if possible). For that, one:

- Suppresses the possible ambiguities in G
- Suppresses the left recursions
- Left factorizes

Outline

PUB Cours-Librairie, av. P. Héger 42, B-1000 Bruxelles / INFO-F-403_A

Construction of an $LL(1)$ Parser

Algorithm to build the Actions table $M[A, a]$

Initialisation : $\forall A, a : M[A, a] = \emptyset$

$\forall A \rightarrow \alpha \in P$ (rule number i) :

$\forall a \in First(\alpha Follow(A))$

$M[A, a] \overset{\cup}{\Leftarrow} i$

Note:

The grammar is $LL(1)$ if and only if each entry in the table M has at most one value.

Else, it means that Produce/Produce conflicts are unresolved

For instance for the CFG G with rules (1) and (2): $S \rightarrow aS \mid a$
$M[A, a] = \{1, 2\}$

Construction of the action table for an $LL(1)$ parser

Example (Construction of M for G)

with rules :

$S' \rightarrow E\$$ (0)
$E \rightarrow TE'$ (1)
$E' \rightarrow +TE'$ (2)
$E' \rightarrow \epsilon$ (3)
$T \rightarrow FT'$ (4)
$T' \rightarrow *FT'$ (5)
$T' \rightarrow \epsilon$ (6)
$F \rightarrow (E)$ (7)
$F \rightarrow id$ (8)

$First(E\$) = \{id, (\ \}$
$First(TE') = \{id, (\ \}$
$First(+TE') = \{+\}$
$First(FT') = \{id, (\ \}$
$First(*FT') = \{*\}$
$First((E)) = \{(\ \}$
$First(id) = \{id\}$

$Follow(S') = \{\epsilon\}$
$Follow(E) = \{), \$\}$
$Follow(E') = \{), \$\}$
$Follow(T) = \{), \$, +\}$
$Follow(T') = \{), \$, +\}$
$Follow(F) = \{), \$, +, *\}$

M	*id*	+	*	(	)	$
S'	0			0		
E	1			1		
E'		2			3	3
T	4			4		
T'		6	5		6	6
F	8			7		

Analysis of a string with an $LL(1)$ parser

Example (Analysis of $a * (b + c)\$$)

On the stack	Remaining input	Action	Output
S'⊣	$a * (b + c)\$$	P0	ϵ
E $⊣	$a * (b + c)\$$	P1	0
TE' $⊣	$a * (b + c)\$$	P4	0 1
FT'E'$⊣	$a * (b + c)\$$	P8	0 1 4
idT'E'$⊣	$*(b + c)\$$	M	0 1 4 8
T'E'$⊣	$*(b + c)\$$	P5	0 1 4 8
*FT'E'$⊣	$*(b + c)\$$	M	0 1 4 8 5
FT'E'$⊣	$(b + c)\$$	P7	0 1 4 8 5
(E)T'E'$⊣	$(b + c)\$$	M	0 1 4 8 5 7
E)T'E'$⊣	$b + c)\$$	P1	0 1 4 8 5 7
TE')T'E'$⊣	$b + c)\$$	P4	0 1 4 8 5 7 1
FT'E')T'E'$⊣	$b + c)\$$	P8	0 1 4 8 5 7 1 4
idT'E')T'E'$⊣	$b + c)\$$	M	0 1 4 8 5 7 1 4 8
T'E')T'E'$⊣	$+c)\$$	P6	0 1 4 8 5 7 1 4 8
E')T'E'$⊣	$+c)\$$	P2	0 1 4 8 5 7 1 4 8 6
+TE')T'E'$⊣	$+c)\$$	M	0 1 4 8 5 7 1 4 8 6 2
TE')T'E'$⊣	$c)\$$	P4	0 1 4 8 5 7 1 4 8 6 2
FT'E')T'E'$⊣	$c)\$$	P8	0 1 4 8 5 7 1 4 8 6 2 4
idT'E')T'E'$⊣	$c)\$$	M	0 1 4 8 5 7 1 4 8 6 2 4 8
T'E')T'E'$⊣	)$	P6	0 1 4 8 5 7 1 4 8 6 2 4 8
E')T'E'$⊣	)$	P3	0 1 4 8 5 7 1 4 8 6 2 4 8 6
)T'E'$⊣	)$	M	0 1 4 8 5 7 1 4 8 6 2 4 8 6 3
T'E'$⊣	$	P6	0 1 4 8 5 7 1 4 8 6 2 4 8 6 3
E'$⊣	$	P3	0 1 4 8 5 7 1 4 8 6 2 4 8 6 3 6
$⊣	$	A	0 1 4 8 5 7 1 4 8 6 2 4 8 6 3 6 3

Outline

Construction of a strongly $LL(k)$ parser

Building of the Actions table $M[A, u] (u \in T^{\leq k})$

Initialisation : $\forall A, u : M[A, u] = \text{Error}$

$\forall A \rightarrow \alpha \in P$ (rule number i) :

$\forall u \in First^k(\alpha Follow^k(A))$

$M[A, u] \stackrel{\cup}{\Leftarrow} i$

Notes:

- The grammar is strongly $LL(k)$ if and only if each entry of the table M has at most one value.
  Otherwise, it means that Produce/Produce conflicts are unresolved.
- In practice M is stored in a compact way

Outline

$LL(k)$ parsers for non strongly $LL(k)$ CFGs

Explanation

It means that for some $A \to \alpha_1 \mid \alpha_2$

$$First^k(\alpha_1 Follow^k(A)) \cap First^k(\alpha_2 Follow^k(A)) \neq \emptyset$$

- unlike "strongly $LL(k)$" CFGs where global look-aheads are sufficient
- for $LL(k)$ CFGs which are not strongly $LL(k)$, one must use local look-aheads

Said differently, in practice, during the analysis, the top of the stack and the look-ahead are not sufficient to determine the Produce to do; we must also "record" in which local context we are.

Let us find out the problem

Let us look again at the definition:
The CFG G is $LL(k)$ $\iff \forall w, x_1, x_2 \in T^*$:

- $S' \ {}_G\!\overset{*}{\Rightarrow} wA\gamma \ {}_G\!\Rightarrow w\alpha_1\gamma \ {}_G\!\overset{*}{\Rightarrow} wx_1$
- $S' \ {}_G\!\overset{*}{\Rightarrow} wA\gamma \ {}_G\!\Rightarrow w\alpha_2\gamma \ {}_G\!\overset{*}{\Rightarrow} wx_2$
- $First^k(x_1) = First^k(x_2)$

$\Rightarrow \alpha_1 = \alpha_2$

Which means that in the $wA\gamma$ context, a look-ahead of k symbols allows to determine exactly what production to use next.

PUB Cours-Librairie, av. P. Héger 42, B-1000 Bruxelles / INFO-F-403_A

Let us find out the problem

Let us take the $LL(2)$ grammar which is not strongly $LL(2)$ with rules:

- $S' \longrightarrow S\$$
- $S \longrightarrow aAa$
- $S \longrightarrow bABa$
- $A \longrightarrow b$
- $A \longrightarrow \epsilon$
- $B \longrightarrow b$
- $B \longrightarrow c$

Depending on whether $S \longrightarrow aAa$ or $S \longrightarrow bABa$ has been done, a look-ahead "ba" means that we must do a produce $A \longrightarrow b$ or $A \longrightarrow \epsilon$

In general, we must compute the "local follow" of each variable for each possible context.

Transformation of $LL(k)$ CFG into strongly $LL(k)$ CFG

Since there are only a finite number of variables and look-aheads of length up to k, one can compute the set of possible local follows.

One can transform G into G' where each variable A is replaced by a couple $[A, L]$

- the name A of the variable in G
- the local follow, i.e. the set L of possible local look-aheads

In the previous example, the variable A transforms into $[A, \{a\$\}]$ and $[A, \{ba, ca\}]$

Property

If G is $LL(k)$ then G' is strongly $LL(k)$

Transforming an $LL(k)$ CFG into a strongly $LL(k)$ CFG

Algorithm to transform an $LL(k)$ CFG G into a strongly $LL(k)$ G'

With $G = \langle V, T, P, S' \rangle$ one builds $G' = \langle V', T, P', S'' \rangle$ as follows:

Initially:

$V' \Leftarrow \{[S', \{\epsilon\}]\}$

$S'' = [S', \{\epsilon\}]$

$P' = \emptyset$

Repeat until every new variable has its rules:

With $[A, L] \in V' \wedge A \rightarrow \alpha \equiv x_0 B_1 x_1 B_2 \ldots B_m x_m \in P$ ($x_i \in T^*, B_i \in V$)

$P' \overset{\cup}{\Leftarrow} [A, L] \rightarrow T(\alpha)$ with

$T(\alpha) = x_0[B_1, L_1]x_1[B_2, L_2] \ldots [B_m, L_m]x_m$

$L_i = First^k(x_i B_{i+1} \ldots B_m x_m.L)$

$\forall 1 \leq i \leq m : V' \overset{\cup}{\Leftarrow} [B_i, L_i]$

Transforming an $LL(k)$ CFG into a strongly $LL(k)$ CFG

Example (of transformation of $LL(2)$ G into strongly $LL(2)$ G')

$G = \langle V, T, P, S' \rangle$ with rules:

$S' \rightarrow S\$$	(0)
$S \rightarrow aAa$	(1)
$S \rightarrow bABa$	(2)
$A \rightarrow b$	(3)
$A \rightarrow \epsilon$	(4)
$B \rightarrow b$	(5)
$B \rightarrow c$	(6)

is transformed into: $G' = \langle V', T, P', S'' \rangle$ with :

Rule	number in G'	number in G
$[S', \{\epsilon\}] \rightarrow [S, \{\$\}]\$$	(0′)	(0)
$[S, \{\$\}] \rightarrow a[A, \{a\$\}]a$	(1′)	(1)
$[S, \{\$\}] \rightarrow b[A, \{ba, ca\}][B, \{a\$\}]a$	(2′)	(2)
$[A, \{a\$\}] \rightarrow b$	(3′)	(3)
$[A, \{a\$\}] \rightarrow \epsilon$	(4′)	(4)
$[A, \{ba, ca\}] \rightarrow b$	(5′)	(3)
$[A, \{ba, ca\}] \rightarrow \epsilon$	(6′)	(4)
$[B, \{a\$\}] \rightarrow b$	(7′)	(5)
$[B, \{a\$\}] \rightarrow c$	(8′)	(6)

Transforming an $LL(k)$ CFG into a strongly $LL(k)$ CFG

Example (of transformation of $LL(2)$ G into strongly $LL(2)$ G')

$G' = \langle V', T, P', S'' \rangle$ with :

Rule	number in G'	number in G
$[S', \{\epsilon\}] \rightarrow [S, \{\$\}]\$$	$(0')$	(0)
$[S, \{\$\}] \rightarrow a[A, \{a\$\}]a$	$(1')$	(1)
$[S, \{\$\}] \rightarrow b[A, \{ba, ca\}][B, \{a\$\}]a$	$(2')$	(2)
$[A, \{a\$\}] \rightarrow b$	$(3')$	(3)
$[A, \{a\$\}] \rightarrow \epsilon$	$(4')$	(4)
$[A, \{ba, ca\}] \rightarrow b$	$(5')$	(3)
$[A, \{ba, ca\}] \rightarrow \epsilon$	$(6')$	(4)
$[B, \{a\$\}] \rightarrow b$	$(7')$	(5)
$[B, \{a\$\}] \rightarrow c$	$(8')$	(6)

M	ab	aa	bb	bc	ba	ca	$a\$$
$[S', \epsilon]$	$0'$	$0'$	$0'$	$0'$			
$[S, \{\$\}]$	$1'$	$1'$	$2'$	$2'$			
$[A, \{a\$\}]$					$3'$		$4'$
$[A, \{ba, ca\}]$			$5'$	$5'$	$6'$	$6'$	
$[B, \{a\$\}]$					$7'$	$8'$	

Analysis with the built strongly $LL(k)$ CFG

Example (Analysis of $bba\$$)

M	ab	aa	bb	bc	ba	ca	$a\$$
$[S', \epsilon]$	$0'$	$0'$	$0'$	$0'$			
$[S, \{\$\}]$	$1'$	$1'$	$2'$	$2'$			
$[A, \{a\$\}]$					$3'$		$4'$
$[A, \{ba, ca\}]$			$5'$	$5'$	$6'$	$6'$	
$[B, \{a\$\}]$					$7'$	$8'$	

During the analysis, we can output the corresponding rules of G that are used

On the stack	Input	Action	Out. of G'	Out. of G
$[S', \epsilon] \dashv$	$bba\$$	P0'	ϵ	ϵ
$[S, \$]\ \$ \dashv$	$bba\$$	P2'	0'	0
$b\ [A, \{ba, ca\}][B, \{a\$\}]\ a\$ \dashv$	$bba\$$	M	0' 2'	0 2
$[A, \{ba, ca\}][B, \{a\$\}]a\$\dashv$	$ba\$$	P6'	0' 2'	0 2
$[B, \{a\$\}]a\$\dashv$	$ba\$$	P7'	0' 2' 6'	0 2 4
$ba\$\dashv$	$ba\$$	M	0' 2' 6' 7'	0 2 4 5
$a\$\dashv$	$ba\$$	M	0' 2' 6' 7'	0 2 4 5
$\$\dashv$	$ba\$$	A	0' 2' 6' 7'	0 2 4 5

Outline

Error handling and synchronizations in predictive analysis

When an error occurs, the parser can decide

- to inform of the error and stop.
- to try to continue (without code production) to detect possible further errors (resynchronization).

One detects an error when

- the terminal on top of the stack does not correspond to the next symbol on input
- the variable on top of the stack has not, as look-ahead, the next input(s)

In these cases, the parser can try to resynchronize, by modifying

- the stack
- the input

Example: resynchronization in panic mode

Error handling in **panic mode**

When an error in found:

- Pop the terminal on top of stack until the first variable
- Skip the input symbols which do not correspond to a look-ahead or a resynchronization symbol associated to the variable
- If the symbol met is a synchronisation symbol, Pop the variable from the stack
- Continue the analysis (hoping that synchronization has been achieved successfully)

Error handling in **panic mode** for an $LL(1)$ parser

Synchronization symbols

If the entry is not already used in M, we add in M, for each variable A, the synchronization symbols of A; for instance:

- the $Follow(A)$ symbols
- other well chosen symbols

Example (Action table for the $LL(1)$ parser of $G = \langle V, T, P, S' \rangle$)

with rules:

$$S' \rightarrow E \quad (0) \qquad T \rightarrow FT' \quad (4)$$
$$E \rightarrow TE' \quad (1) \qquad T' \rightarrow *FT' \mid \epsilon \quad (5) \mid (6)$$
$$E' \rightarrow +TE' \mid \epsilon \quad (2) \mid (3) \qquad F \rightarrow (E) \mid id \quad (7) \mid (8)$$

M	*id*	+	*	(	)	$
S'	0			0		
E	1			1	*sync*	*sync*
E'		2			3	3
T	4	*sync*		4	*sync*	*sync*
T'		6	5		6	6
F	8	*sync*	*sync*	7	*sync*	*sync*

Analysis of a wrong string with an $LL(1)$ parser

Example (Analysis of $+a*+b\$$)

On the stack	Input	Action	Output
S'⊣	$+a*+b\$$	Error : skips **+**	ϵ
S'⊣	$a*+b\$$	P0	*
E $⊣	$a*+b\$$	P1	* 0
TE' $⊣	$a*+b\$$	P4	* 0 1
FT'E' $⊣	$a*+b\$$	P8	* 0 1 4
idT'E' $⊣	$a*+b\$$	M	* 0 1 4 8
T'E' $⊣	$*+b\$$	P5	* 0 1 4 8
*FT'E' $⊣	$*+b\$$	M	* 0 1 4 8 5
FT'E' $⊣	$+b\$$	Error: + is sync: pop F	* 0 1 4 8 5
T'E' $⊣	$+b\$$	P6	* 0 1 4 8 5 *
E' $⊣	$+b\$$	P2	* 0 1 4 8 5 * 6
+TE' $⊣	$+b\$$	M	* 0 1 4 8 5 * 6 2
TE'$⊣	$b\$$	P4	* 0 1 4 8 5 * 6 2
FT'E'$⊣	$b\$$	P8	* 0 1 4 8 5 * 6 2 4
idT'E'$⊣	$b\$$	M	* 0 1 4 8 5 * 6 2 4 8
T'E'$⊣	$b\$$	P6	* 0 1 4 8 5 * 6 2 4 8
E'$⊣	$	P3	* 0 1 4 8 5 * 6 2 4 8 6
$⊣	$	A (with errors)	* 0 1 4 8 5 * 6 2 4 8 6 3

Outline

Example of recursive $LL(1)$ parser

An $LL(k)$ parser can easily be encoded as a recursive program where the parsing of each variable of G is achieved by a recursive procedure.

The following code gives a recursive $LL(1)$ parser for G whose production rules are:

$$E \rightarrow TE' \quad (1)$$
$$E' \rightarrow +TE' \mid \epsilon \quad (2) \mid (3)$$

$$T \rightarrow FT' \quad (4)$$
$$T' \rightarrow *FT' \mid \epsilon \quad (5) \mid (6)$$
$$F \rightarrow (E) \mid id \quad (7) \mid (8)$$

Recursive code of an $LL(1)$ parser

```
/* main.c */
/* E -> TE'; E' -> +TE' | e ; T -> FT' ;
   T' -> *FT' | e;  F -> (E) | id  */

#define NOTOK 0
#define OK 1
char Phrase[100];
int CurToken;
int Res;

int Match(char t)
{
  if (Phrase[CurToken]==t)
  {
    CurToken++;
    return OK;
  }
  else
  {
    return NOTOK;
  }
}

void Send_output(int no)
{
  printf("%d  ",no);
}
```

Recursive code of an $LL(1)$ parser

```
int E(void)
{
  Send_output(1);
  if(T()==OK)
  if(E2()==OK)
    return(OK);
  return(NOTOK);

}
```

Recursive code of an $LL(1)$ parser

```
int E2(void)
{
  switch (Phrase[CurToken])
  {
  case '+':
    Send_output(2);
    Match('+');
    if(T()==OK)
    if(E2()==OK)
      return(OK);
    break;
  case '$':
  case ')':
    Send_output(3);
    return(OK);
  }
  return(NOTOK);
}
```

Recursive code of an $LL(1)$ parser

```
int T(void)
{
  Send_output(4);
  if(F()==OK)
  if(T2()==OK)
    return(OK);
  return(NOTOK);
 }
```

Recursive code of an $LL(1)$ parser

```
int T2(void)
{
  switch (Phrase[CurToken])
  {
  case '*':
    Send_output(5);
    Match('*');
    if(F()==OK)
    if (T2()==OK)
      return(OK);
    break;
  case '+':
  case '$':
  case ')':
    Send_output(6);
    return(OK);
  }
  return(NOTOK);
}
```

Recursive code of an *LL*(1) parser

```
int F(void)
{
  switch (Phrase[CurToken])
  {
  case '(':
    Send_output(7);
    Match('(');
    if (E()==OK)
    if (Match(')')==OK)
      return(OK);
    break;
  case 'n':
    Send_output(8);
    Match('n');
    return (OK);
    break;
  }
  return(NOTOK);
}
```

Recursive code of an *LL*(1) parser

```
int main()
{
  scanf("%s", Phrase);
  if (E()!= OK)
  {
    printf("error1\n");
  }
  else
  {
    if (Match('$')==OK)
    {
      printf("\n");
    }
    else
    {
      printf("error2\n");
    }
  }
}
```

Chapter 10: LR(k) parsers

Outline

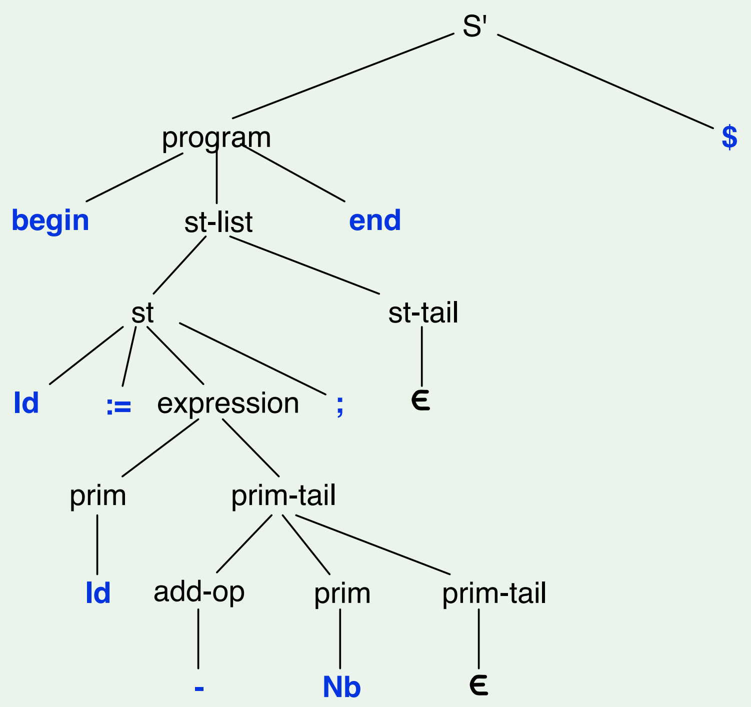
PUB Cours-Librairie, av. P. Héger 42, B-1000 Bruxelles / INFO-F-403_A

Example (Grammar of a very simple language)

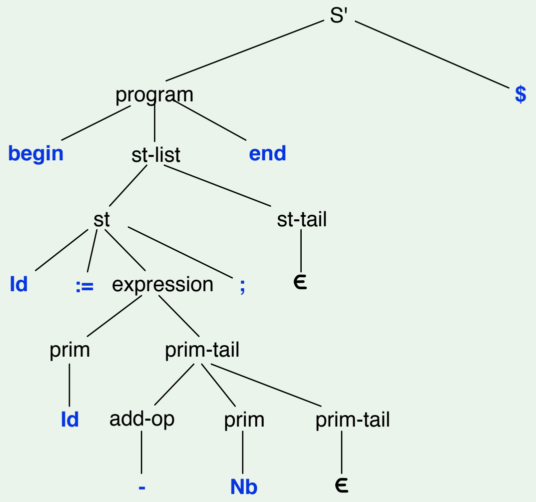
Rules	*Production rules*		
0	*S'*	→	*program* **$**
1	*program*	→	**begin** *st-list* **end**
2	*st-list*	→	*st st-tail*
3	*st-tail*	→	*st st-tail*
4	*st-tail*	→	ϵ
5	*st*	→	**Id :=** *expression* **;**
6	*st*	→	**read (** *id-list* **) ;**
7	*st*	→	**write(** *expr-list* **);**
8	*id-list*	→	**Id** *id-tail*
9	*id-tail*	→	**, Id** *id-tail*
10	*id-tail*	→	ϵ
11	*expr-list*	→	*expression expr-tail*
12	*expr-tail*	→	**,** *expression expr-tail*
13	*expr-tail*	→	ϵ
14	*expression*	→	*prim prim-tail*
15	*prim-tail*	→	*add-op prim prim-tail*
16	*prim-tail*	→	ϵ
17	*prim*	→	**(** *expression* **)**
18	*prim*	→	**Id**
19	*prim*	→	**Nb**
20 | *21*	*add-op*	→	**+** | **-**

Example of syntactic tree

Example (Syntactic tree corresponding to **begin Id := Id - Nb ; end** $)



Example (right-most derivation corresponding to the syntactic tree)



Right-most derivation $S' \stackrel{*}{\Rightarrow}_G$ **begin Id := Id - Nb ; end** $:

0 1 2 4 5 14 15 16 19 21 18

Example (Complete corresponding right-most derivation)

Rule	sentential form	
	S'	⇒
0	program $	⇒
1	**begin** st-list **end** $	⇒
2	**begin** st st-tail **end** $	⇒
4	**begin** st **end** $	⇒
5	**begin Id :=** expression **; end** $	⇒
14	**begin Id :=** prim prim-tail **; end** $	⇒
15	**begin Id :=** prim add-op prim prim-tail **; end** $	⇒
16	**begin Id :=** prim add-op prim **; end** $	⇒
19	**begin Id :=** prim add-op **Nb ; end** $	⇒
21	**begin Id :=** prim **- Nb ; end** $	⇒
18	**begin Id := Id - Nb ; end** $	

Outline of a bottom-up parser

PDA with one state and with output.
We start from the input string and build the tree bottom-up. In order to do so, two actions are available:

1. "Shift": shift the input symbols on the stack until identification of a right-hand part α (handle) of the rule $A \to \alpha$
2. "Reduction": replacement of α by A [a]

Initially the stack is empty.
The PDA can do 4 kinds of actions :

- Shift: reading of an input symbol and push of this symbol on the stack
- Reduce: the top of the stack α corresponding to the handle (the right part of a rule number i : $A \to \alpha$), is replaced by A on the stack and the number i of the used rule is written on the output
- Accept: corresponds to a Reduce of the rule $S' \to S\$$ (which shows that the input has been completely read and analyzed); the analysis is completed successfully
- Error: If no Shift nor Reduce is possible

[a] Formally corresponds to $|\alpha|$ pops followed by a push of A

Remark:

- One can see that the analysis corresponds to a reverse order right-most analysis: one starts from the string and goes up in the derivation back to the start symbol. Analysis is done in reverse order since the input is read from left to right.
- The output will be built in reverse order (each new output is put before all what has been produced before) to obtain this right-most derivation.

Right-most derivation

Example (Corresponding complete right-most derivation)

On the stack	Remaining input	Act	Output
⊢	**begin Id := Id - Nb ; end** $	S	ϵ
⊢ **begin**	**Id := Id - Nb ; end** $	S	ϵ
⊢ **begin Id**	**:= Id - Nb ; end** $	S	ϵ
⊢ **begin Id :=**	**Id - Nb ; end** $	S	ϵ
⊢ **begin Id := Id**	**- Nb ; end** $	R18	ϵ
⊢ **begin Id :=** prim	**- Nb ; end** $	S	18
⊢ **begin Id :=** prim **-**	**Nb ; end** $	S	18
⊢ **begin Id :=** prim **-**	**Nb ; end** $	R21	18
⊢ **begin Id :=** prim add-op	**Nb ; end** $	S	21 18
⊢ **begin Id :=** prim add-op **Nb**	**; end** $	R19	21 18
⊢ **begin Id :=** prim add-op prim	**; end** $	R16	19 21 18
⊢ **begin Id :=** prim add-op prim prim-tail	**; end** $	R15	16 19 21 18
⊢ **begin Id :=** prim prim-tail	**; end** $	R14	15 16 19 21 18
⊢ **begin Id :=** expression	**; end** $	S	14 15 16 19 21 18
⊢ **begin Id :=** expression **;**	**end** $	R5	14 15 16 19 21 18
⊢ **begin** st	**end** $	R4	5 14 15 16 19 21 18
⊢ **begin** st st-tail	**end** $	R2	4 5 14 15 16 19 21 18
⊢ **begin** st-list	**end** $	S	2 4 5 14 15 16 19 21 18
⊢ **begin** st-list **end**	$	R1	2 4 5 14 15 16 19 21 18
⊢ program	$	S	1 2 4 5 14 15 16 19 21 18
⊢ program $	ϵ	A	1 2 4 5 14 15 16 19 21 18
⊢ S'	ϵ		0 1 2 4 5 14 15 16 19 21 18

where:

- S : Shift
- Ri : Reduce with the rule *i*
- A : Accept (corresponds to a Reduce with the rule 0)
- E : Error (or blocking which requests a backtracking)

Points to improve in the outline of the bottom-up parser

Criticism of the outline of the bottom-up parser

- As such, this parser is extremely inefficient since it must backtrack to explore all the possibilities.
- In this kind of parser, a choice must occur when both a “Reduce” and “Shift” can be done, or when several “Reduces” are possible.
- If several choices are possible and no criteria in the method allow to choose, one can talk of Shift/Reduce or Reduce/Reduce conflicts.
- Without guide when the choice must be done, possibly every possible Shift and Reduce must be tried: the parser could therefore take an exponential time (typically in the length of the input) which is unacceptable!
- We will show efficient bottom-up parsing techniques in this chapter.

Outline

$LR(k)$ CFG

$LR(k)$ CFG

$LR(k)$ means

- **L**eft scanning
- **R**ightmost derivation
- **k** lookahead symbols

Definition (The CFG $G' = \langle V, T, P, S' \rangle$ is $LR(k)$ (k a fixed natural number) if)

- $S' \stackrel{*}{\Rightarrow}_G \gamma A x \Rightarrow_G \gamma\alpha x$
- $S' \stackrel{*}{\Rightarrow}_G \delta B y \Rightarrow_G \delta\beta y = \gamma\alpha x'$
- $First^k(x) = First^k(x')$

$\Rightarrow$

$\gamma A x' = \delta B y$: i.e.
$\gamma = \delta$,
$A = B$,
$x' = y$

Intuitively it means that if we look at $First^k(x)$ we can determine uniquely the handle $A \rightarrow \alpha$ in $\gamma\alpha x$

Non $LR(k)$ grammars

Example (of grammar which is neither $LR(0)$ nor $LR(1)$)

The following grammar G' is not $LR(0)$ nor $LR(1)$

$$\begin{array}{lcl} S' & \rightarrow & S\$ \\ S & \rightarrow & Sa \mid a \mid \epsilon \end{array}$$

- $S' \stackrel{2}{\Rightarrow}_G Sa\$ \Rightarrow_G a\$$
- $S' \stackrel{2}{\Rightarrow}_G Sa\$ \Rightarrow_G aa\$$
- $First(a\$) = First(aa\$)$

$$\begin{array}{ccc} A = S & \gamma = \epsilon & \alpha = \epsilon \\ x = a\$ & x' = aa\$ & B = S \\ \delta = \epsilon & \beta = a & y = a\$ \end{array}$$

But $\delta By = Sa\$ \neq Saa\$ = \gamma Ax'$

Note that G' is also ambiguous.

Theorem (Every ambiguous CFG G' is not $LR(k)$ for any k)

Types of $LR(k)$ parsers studied

Types of bottom-up parsers studied here

We will study 3 types of $LR(k)$ parsers

- "Canonical *LR*" parsers: most powerful but expensive
- "Simple *LR*" (SLR) parsers: less expensive but less powerful
- "*LALR*" parsers: more powerful than *SLR* (a little less powerful but less expensive than *LR*)

Operation of a bottom-up parser

All 3 types of bottom-up parsers use

- a stack
- an *Action* table which, depending on the top of the stack and the look-ahead determines if the parser must do a Shift or a Reduce i where i is the number of the rule to use
- a *Successor* table which determines what must be put on the stack (see below)

Outline

$LR(0)$ parsing

Principle of construction of the parser

- The principle of an LR parser is to determine a handle and to achieve the reduction
- We construct a deterministic finite automaton which recognizes all the *viable prefixes* and determines when we reach the handle

Notion of $LR(0)$ item

- We use the notion of $LR(0)$-item for this construction
- A $LR(0)$-item is a production rule with a $\bullet$ somewhere is the right part of the rule
- For the rule $A \rightarrow \alpha$: $A \rightarrow \alpha_1 \bullet \alpha_2$ with $\alpha = \alpha_1\alpha_2$ means that it is possible we are
  - analysing a rule $A \rightarrow \alpha$,
  - after the analysis of α_1 (α_1 is on the stack)
  - before the analysis of α_2

$LR(0)$ parsing

Remark on $A \rightarrow \alpha_1 \bullet \alpha_2$

They are grouped possibilities. One could e.g. have the 2 following grouped possibilities:

- $S \rightarrow a \bullet AC$
- $S \rightarrow a \bullet b$

2 types of $LR(0)$-items are particular:

- $A \rightarrow \bullet\alpha$ which predicts that we can start an analysis with the rule $A \rightarrow \alpha$
- $A \rightarrow \alpha\bullet$ which recognizes the end of the analysis of a rule $A \rightarrow \alpha$ (and determines, if no conflict exists, that the corresponding Reduce can be done)

Construction of the $LR(0)$ characteristic Finite State Machine (CFSM)

$LR(0)$ Characteristic Finite State Machine (CFSM)

- A deterministic finite automaton is built. Depending on the input characters read and on the variables already recognized, it determines the possible handle.
- When the automaton reaches an "accepting" state, i.e. which contains an $LR(0)$-item $A \rightarrow \alpha\bullet$ (where a handle is complete), $A \rightarrow \alpha$ is recognised, we can achieve the reduce, i.e. replace α by A
- **The language of this automaton is the set of viable prefixes of G'**

Construction of the $LR(0)$ characteristic finite state machine (CFSM)

Construction of the $LR(0)$ characteristic finite state machine (CFSM)

- Initially mark that S' must be analyzed : $S' \rightarrow \bullet S\$$
- An $LR(0)$-item $A \rightarrow \gamma_1 \bullet B\gamma_2$ where the $\bullet$ is just before a variable B: it means that we "predict" that B must be analyzed just after; i.e., a right part of a rule $B \rightarrow \beta_j$: for all B-productions $B \rightarrow \beta_j$: $B \rightarrow \bullet\beta_j$ must therefore be added
- Add, using the same principle, all $LR(0)$-items $B \rightarrow \bullet\beta$ until stabilisation: this is called the closure operation
- The closure operation allows to obtain the set of possibilities on that state in the analysis of G' and forms a state of the $LR(0)$-CFSM
- The transitions $s \xrightarrow{X} s'$ of this finite automaton express that the analysis of the (terminal or variable) symbol X is completed

Example of $LR(0)$-CFSM

Example (Of construction of $LR(0)$-CFSM)

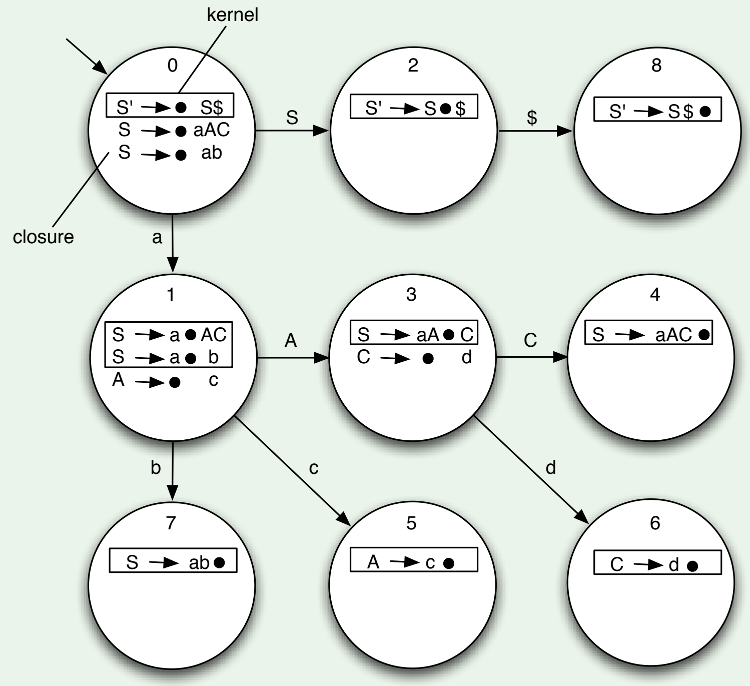
For G' with the following rules:

$$
\begin{array}{llll}
S' & \rightarrow & S\$ & (0) \\
S & \rightarrow & aAC & (1) \\
S & \rightarrow & ab & (2) \\
A & \rightarrow & c & (3) \\
C & \rightarrow & d & (4)
\end{array}
$$

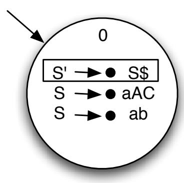
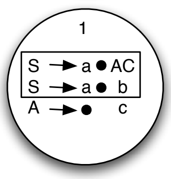
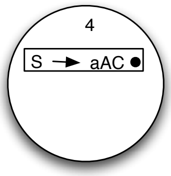
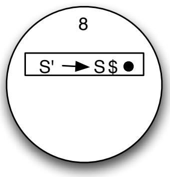
The $LR(0)$-CFSM is represented by the following figure :

Example of $LR(0)$-CFSM

Example (Of construction of an $LR(0)$-CFSM)

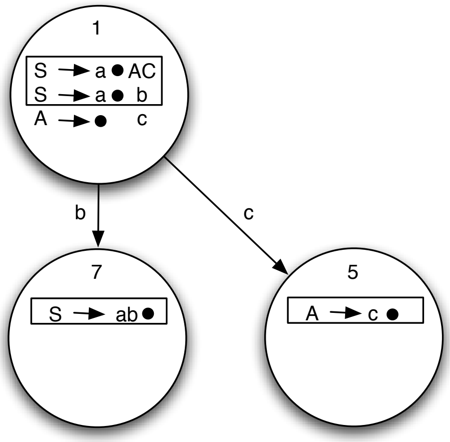


Intuitive meaning of the $LR(0)$-CFSM

State of the CFSM:	Status of the analysis:
	The beginning of the analysis (initial state), *S*$ must be analyzed, i.e. either *aAC* or *ab* must be analyzed
	*a* has already been analyzed and either *AC* or *b* remains to be analyzed. To analyze *A*, *c* must be analyzed
	*aAC* has been analyzed; therefore the analysis of a *S* is completed
	*S*$ has been analyzed; therefore the analysis of S' is terminated (there is only one $S' \rightarrow S\$$ - since it has been added to *G*); the analysis is therefore completed successfully.

Interpretation of the $LR(0)$-CFSM

State and transitions in the CFSM:

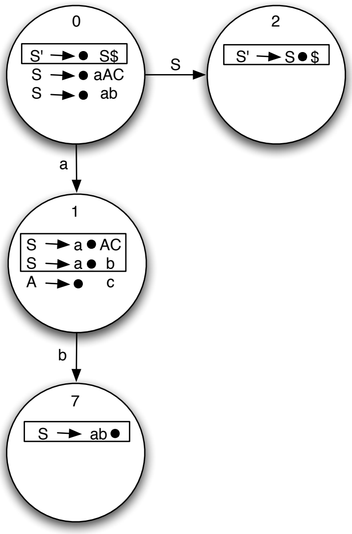


Status of the analysis:

- In the state 1 (among others) a *b* or a *c* can be analyzed
- A Shift will be done
- If the shifted input is a *b*: go to state 7
- If the shifted input is a *c*: go to state 5
- If the input is any other character, it is an error (the analyzed string does not belong to the language)

Interpretation of the $LR(0)$-CFSM

State and transitions in the CFSM:



Status of the analysis:

- In the state 7 an analysis of *ab* is completed ...
- this corresponds to the analysis of an *S*, started 2 states before in the path taken by the analysis, i.e. in the state 0
- then the analysis of *S*, started in state 0, is now completed
- go to state 2

This corresponds to a Reduce of $S \rightarrow ab$

Note

One sees that a *Reduce* corresponds to a *Shift* of a variable whose analysis is completed

LR parsing

LR parsing

- This analysis of a string corresponds to the identification of handles
- A stack is used to keep the states of the path taken in the $LR(0)$-CFSM
- The analysis starts at state 0 which includes $S' \rightarrow \bullet S\$$
- Implicitly, an error state $\emptyset$ exists; any transition which is not explicitly expected goes to that error state

Construction of the *Action* and *Successor* tables

The *Action* and *Successor* tables synthesize the information to keep from the $LR(0)$-CFSM.

Construction of the *Action* table

The *Action* table gives for each state of the $LR(0)$-CFSM, the action to do among:

- Shift
- Reduce i where i is the number of the rule of G' to use for the reduce
- Accept which corresponds to Reduce $S' \rightarrow S\$$
- Error if in the state no reduce is possible and the input is not expected

Construction of the *Successor* table

The *Successor* table contains the transition function of the $LR(0)$-CFSM. It is used to determine

- For a Shift action
- For a Reduce action

which state will be the next one (i.e. which state is Pushed on the stack), possibly (in case of Reduce) after having removed $|\alpha|$ symbols)

Construction algorithms of an $LR(0)$ parser

Closure algorithm for a set s of $LR(0)$-items

Closure(s) :=
  Closure $\Leftarrow s$
  Repeat
    Closure' $\Leftarrow$ Closure
    if $[B \rightarrow \delta \bullet A\rho] \in$ Closure
      $\forall\ A \rightarrow \gamma \in P$ (A-production of G')
        Closure $\stackrel{\cup}{\Leftarrow} \{[A \rightarrow \bullet\gamma]\}$
      f$\forall$
    fi
  Until : Closure = Closure'
  Return : Closure
Endproc

Principles of bottom-up parsing
LR(k) CFGs
LR(0) parsers
LR(1) parsers
SLR(1) parsers
LALR(1) parsers
LL vs LR classes
The Yacc (Bison) tool

Construction algorithms of an $LR(0)$ parser

Algorithm to compute the next state of a state s for a symbol X

Transition(s,X) :=
  Transition $\Leftarrow$ Closure($\{[B \rightarrow \delta X \bullet \rho] \mid [B \rightarrow \delta \bullet X\rho] \in s\}$)
  Return : Transition
Endproc

Construction algorithms of an $LR(0)$ parser

Construction algorithm of the set of states $\mathcal{C}$ of the $LR(0)$-CFSM

Note: A state of $\mathcal{C}$ is a set of $LR(0)$-items

Construction-LR(0)-CFSM :=

$\mathcal{C} \Leftarrow$ Closure$(\{[S' \rightarrow \bullet S\$]\}) \cup \{\emptyset\}$ /* where $\emptyset$ is the error state */

Repeat

Given $s \in \mathcal{C}$ not processed yet

$\forall X \in V \cup T$

$Successor[s, X] \Leftarrow$Transition$(s, X)$

$\mathcal{C} \overset{\cup}{\Leftarrow} Successor[s, X]$

f$\forall$

Until every s has been processed

/* The empty entries in *Successor* point to the error state $\emptyset$ */

Endproc

Construction algorithms of an $LR(0)$ parser

Construction algorithm of the *Action* table

Construction-Action-table() :=

$\forall s \in \mathcal{C}$: $Action[s] \Leftarrow \emptyset$

$\forall s \in \mathcal{C}$

if $[A \rightarrow \alpha \bullet] \in s$: $Action[s] \overset{\cup}{\Leftarrow} Reduce\ i$

/* where $A \rightarrow \alpha$ is the rule i */

if $[A \rightarrow \alpha \bullet a\beta] \in s$: $Action[s] \overset{\cup}{\Leftarrow} Shift$

if $[S' \rightarrow S\$\bullet] \in s$: $Action[s] \overset{\cup}{\Leftarrow} Accept$

Endproc

Language recognized by the $LR(0)$-CFSM

Theorem (The CFG G' is $LR(0)$ if and only if the *Action* table has at most one action in each entry)

Intuitively, Action[s] summarises the state s

Theorem (The language recognized by the $LR(0)$-CFSM of a CFG G' $LR(0)$, when all states are accepting, is the set of its viable prefixes)

Remark

The states of the automaton are generally named with a natural integer identifier (0 for the initial state)

Construction of the tables for G'

Example (*Action* and *Successor* tables for G' of the slide 306)

Action

	Action
0	S
1	S
2	S
3	S
4	R1
5	R3
6	R4
7	R2
8	A

Successor

	a	b	c	d	A	C	S	$
0	1						2	
1		7	5		3			
2								8
3				6		4		
4								
5								
6								
7								
8								

Reminder: Implicitly the empty entries of the table *Successor* refer to an error state

Tables of an $LR(0)$ CFG

Notes

- Few grammars are $LR(0)$ (but there are more than $LL(0)$ grammars)
- If G' is not $LR(0)$, one must consider a look-ahead and build an $LR(k)$ parser with $k \geq 1$.
- Obviously here also, the bigger the k, the more complex the parser.
- We try to limit ourselves to $k = 1$.

Predictive $LR(k)$ parsers

Remark on the algorithm

- The general $LR(k)$ parsing algorithm for $k \geq 0$ is given here
- If $k = 0$, the *Action* table is a vector
- If $k \geq 1$, the *Action* table has, as second parameter, a look-ahead u of size $\leq k$
- The construction of the *Action* and *Successor* tables for $k > 0$ will be given in the next sections.

PUB Cours-Librairie, av. P. Héger 42, B-1000 Bruxelles / INFO-F-403_A

Predictive $LR(k)$ parsers

General algorithm for the way an $LR(k)$ parser works

We assume the *Action* and *Successor* tables are already built

Parser-LR-k():=

Initially: *Push*(0) /* Initial state */

Loop

$s \leftarrow Top()$

if $Action[s, \mathbf{u}] = Shift$: $Shift(s)$ /* $|u| \leq k$ and *Action* is a */

if $Action[s, \mathbf{u}] = Reduce\ i$: $Reduce(i)$ /* vector (no u parameter */

if $Action[s, \mathbf{u}] = \emptyset$: $Error()$ /* if the parser is $LR(0)$) */

if $Action[s, \mathbf{u}] = Accept$: $Accept()$

Endloop

Endproc

- Shift(s) := $X \leftarrow$ next input; $Push(Successor[s, X])$ Endproc
- Reduce(i) := (the rule $i \equiv A \rightarrow \alpha$) **For** $j = 1$ to $|\alpha|$ $Pop()$ **endfor**;
  $s \leftarrow Top()$; $Push(Successor[s, A])$; Endproc
- Error() := Informs of an error in the analysis; Endproc
- Accept() := Informs of the success of the analysis; Endproc

$LR(0)$ analysis

Example (Analysis of the string $acd\$$ for G')

with the following rules:

$$
\begin{array}{llll}
S' & \rightarrow & S\$ & (0) \\
S & \rightarrow & aAC & (1) \\
S & \rightarrow & ab & (2) \\
A & \rightarrow & c & (3) \\
C & \rightarrow & d & (4)
\end{array}
$$

On the stack	Remaining Input	Act	Output
$\vdash$ 0	$acd\$$	S	ϵ
$\vdash$ 01	$cd\$$	S	ϵ
$\vdash$ 015	$d\$$	R3	ϵ
$\vdash$ 013	$d\$$	S	3
$\vdash$ 0136	\$	R4	3
$\vdash$ 0134	\$	R1	43
$\vdash$ 02	\$	S	143
$\vdash$ 028	ϵ	A	143

Correspondence between the analysis such as presented at the beginning of the chapter and LR analysis

Although it is useless, one can (to help the reader's understanding) explicitly *Push* the *Shifted* symbols (terminals or variables during the Reduce) on the stack

On the stack	Remaining Input	Act	Output
$\vdash 0$	$acd\$$	S	ϵ
$\vdash 0a1$	$cd\$$	S	ϵ
$\vdash 0a1c5$	$d\$$	R3	ϵ
$\vdash 0a1A3$	$d\$$	S	3
$\vdash 0a1A3d6$	\$	R4	3
$\vdash 0a1A3C4$	\$	R1	43
$\vdash 0S2$	\$	S	143
$\vdash 0S2\$8$	ϵ	A	143

We have :

- the stack where the numbers of states have been abstracted, concatenated with the remaining input is at each moment a sentential form of a right-most derivation.
- The stack, where the number of states are abstracted, is always a viable prefix.

Example 2 of $LR(0)$-CFSM

Example (Of construction of $LR(0)$-CFSM)

For G_2' with the following rules:

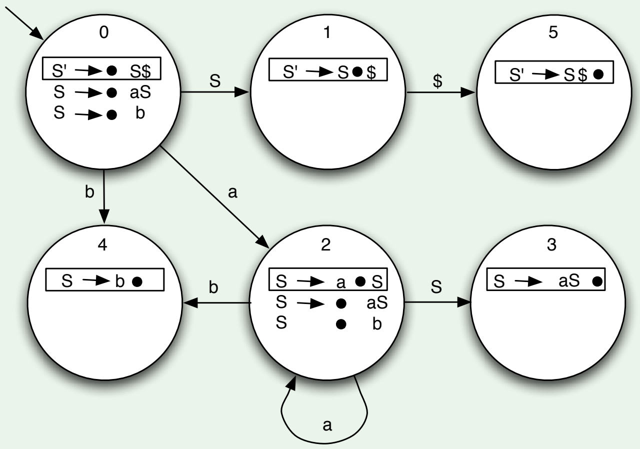
$$\begin{array}{ll} S' \rightarrow S\$ & (0) \\ S \rightarrow aS & (1) \\ S \rightarrow b & (2) \end{array}$$

We get the following $LR(0)$-CFSM and *Action* and *Successor* tables:

PUB Cours-Librairie, av. P. Héger 42, B-1000 Bruxelles / INFO-F-403_A

$LR(0)$-CFSM and *Action* and *Successor* tables

Example ($LR(0)$-CFSM and *Action* and *Successor* tables)



	Action
0	S
1	S
2	S
3	R1
4	R2
5	A

Action

	a	b	S	$
0	2	4	1	
1				5
2	2	4	3	
3				
4				
5				

Successor

$LR(0)$ analysis

Example (Analysis of the string *aab*\$ for G_2')

with the rules :

$$
\begin{array}{ll}
S' \rightarrow S\$ & (0) \\
S \rightarrow aS & (1) \\
S \rightarrow b & (2)
\end{array}
$$

On the stack	remaining input	Act	Output
$\vdash 0$	*aab*\$	S	ϵ
$\vdash 02$	*ab*\$	S	ϵ
$\vdash 022$	*b*\$	S	ϵ
$\vdash 0224$	\$	R2	ϵ
$\vdash 0223$	\$	R1	2
$\vdash 023$	\$	R1	12
$\vdash 01$	\$	S	112
$\vdash 015$	ϵ	A	112

Note: the 0 rule is not given as output

Example 3 of *LR*(0)-CFSM

Example (Of *LR*(0)-CFSM construction)

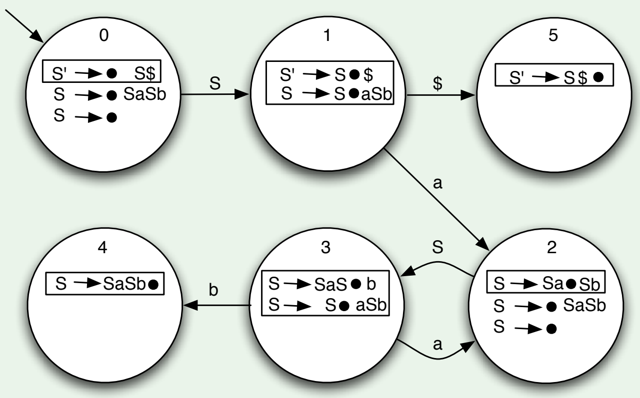
For G'_3 with rules :

$$\begin{array}{lr} S' \longrightarrow S\$ & (0) \\ S \longrightarrow SaSb & (1) \\ S \longrightarrow \epsilon & (2) \end{array}$$

We get the following *LR*(0)-CFSM and *Action* and *Successor* tables:

LR(0)-CFSM and *Action* and *Successor* tables

Example (*LR*(0)-CFSM and *Action* and *Successor* tables)



	Action
0	R2
1	S
2	R2
3	S
4	R1
5	A

Action

	a	b	S	$
0			1	
1	2			5
2			3	
3	2	4		
4				
5				

Successor

$LR(0)$ analysis

Example (Analysis of the string $aabb\$$ for G_3')

with rules :

$$\begin{array}{lr} S' \to S\$ & (0) \\ S \to SaSb & (1) \\ S \to \epsilon & (2) \end{array}$$

On the stack	Remaining input	Act	Output
$\vdash$ 0	$aabb\$$	R2	ϵ
$\vdash$ 01	$aabb\$$	S	2
$\vdash$ 012	$abb\$$	R2	2
$\vdash$ 0123	$abb\$$	S	22
$\vdash$ 01232	$bb\$$	R2	22
$\vdash$ 012323	$bb\$$	S	222
$\vdash$ 0123234	$b\$$	R1	222
$\vdash$ 0123	$b\$$	S	1222
$\vdash$ 01234	$\$$	R1	1222
$\vdash$ 01	$\$$	S	11222
$\vdash$ 015	ϵ	A	11222

Non $LR(0)$ grammar

Example (The grammar G_4' is not $LR(0)$)

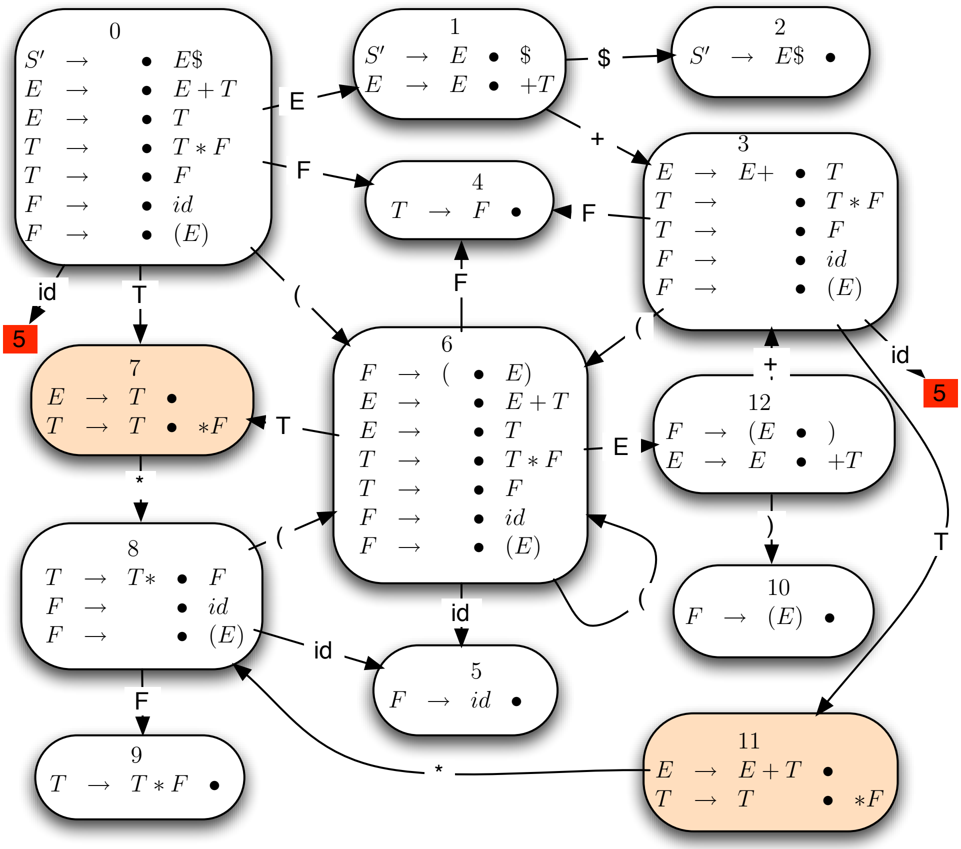
$G_4' = \langle \{S', E, T, F\}, \{+, *, id, (,), \$\}, P_4, S' \rangle$ with rules P_4 :

$$\begin{array}{lcll} S' & \to & E\$ & (0) \\ E & \to & E + T & (1) \\ E & \to & T & (2) \\ T & \to & T * F & (3) \\ T & \to & F & (4) \\ F & \to & id & (5) \\ F & \to & (E) & (6) \end{array}$$

is not $LR(0)$ as can be seen from the construction of its $LR(0)$-CFSM where

- the state 7
- the state 11

both request a Shift and Reduce action (Shift/Reduce conflict).
We will see that G_4' is a $LR(1)$ grammar.



Outline

1. Principles of bottom-up parsing
2. LR(k) CFGs
3. LR(0) parsers
4. **LR(1) parsers**
5. SLR(1) parsers
6. LALR(1) parsers
7. LL vs LR classes
8. The Yacc (Bison) tool

Construction of the $LR(1)$-CFSM

- The look-ahead is useful when a Reduce must be done
- In that case, the possible look-aheads (of length up to k, here $k = 1$) must be determined.
- This amounts to compute the *Follow* sets local to the rules (items)

Computation of the $LR(1)$-items

- $LR(1)$-items have the form $[A \rightarrow \alpha_1 \bullet \alpha_2, u]$ where u is a follow local to $A \rightarrow \alpha$ $(\alpha = \alpha_1\alpha_2)$
- The "initial" $LR(1)$-item is $[S' \rightarrow \bullet S\$, \epsilon]$
- The Closure operation increases a set s of $LR(1)$-items as follows:

- Given $[B \rightarrow \delta \bullet A\rho, \ell] \in s$
- $A \rightarrow \gamma \in P$

$\Big\} \Rightarrow \forall u \in First^k(\rho\ell) : s \overset{\cup}{\Leftarrow} [A \rightarrow \bullet\gamma, u]$

Remark

In what follows we give algorithms for $LR(k)$ even if the examples are restricted to $k = 1$.

Construction algorithms of a $LR(k)$ parser

Closure algorithm of a set s of $LR(k)$-items

Closure(s) **:=**
    Closure $\Leftarrow s$
    Repeat
        Closure' $\Leftarrow$ Closure
        if $[B \rightarrow S \bullet A\rho, \ell] \in$ Closure
            $\forall\ A \rightarrow \gamma \in P$ (A-production of G')
                $\forall u \in First^k(\rho\ell)$: Closure $\overset{\cup}{\Leftarrow} [A \rightarrow \bullet\gamma, u]$
            f$\forall$
        fi
    Until : Closure = Closure'
    Return : Closure
Endproc

Construction algorithms of an $LR(k)$ parser

Algorithm for the computation of the next state of a state s after the symbol X

Transition(s,X) :=

Transition $\Leftarrow$ Closure($\{[B \rightarrow \delta X \bullet \rho, u] \mid [B \rightarrow \delta \bullet X\rho, u] \in s\}$)

Return : Transition

Endproc

Construction algorithms of an $LR(k)$ parser

Construction algorithms of the set of states $\mathcal{C}$ of the $LR(k)$-CFSM

Construction-LR(k)-CFSM :=

$\mathcal{C} \Leftarrow$ Closure($\{[S' \rightarrow \bullet S\$, \epsilon]\}$) $\cup\ \emptyset$

/* where $\emptyset$ is an the error state */

Repeat

Given $s \in \mathcal{C}$ not processed yet

$\forall X \in V \cup T$

$Successor[s, X] \Leftarrow$Transition($s$, X)

$\mathcal{C} \stackrel{\cup}{\Leftarrow} Successor[s, X]$

f$\forall$

Until every s is processed

/* The empty entries of *Successor* point to the $\emptyset$ error state */

Endproc

Construction algorithms of an $LR(k)$ parser

Construction algorithms of the *Action* table

Construction-Action-table :=

$\forall s \in \mathcal{C}, u \in T^{\leq k}$: $Action[s, u] \Leftarrow \emptyset$

$\forall s \in \mathcal{C}$

if $[A \rightarrow \alpha\bullet, u] \in s$: $Action[s, u] \overset{\cup}{\Leftarrow} Reduce\ i$

/*where $A \rightarrow \alpha$ is the rule i */

if $[A \rightarrow \alpha \bullet a\beta, y] \in s \ \wedge\ u \in First^k(a\beta y)$: $Action[s, u] \overset{\cup}{\Leftarrow} Shift$

if $[S' \rightarrow S\$\bullet, \epsilon] \in s$: $Action[s, \epsilon] \overset{\cup}{\Leftarrow} Accept$

Endproc

Construction of the $LR(1)$ parser for G'_4

Example (Construction of the $LR(1)$ parser for G'_4)

$G'_4 = \langle\{S', E, T, F\}, \{+, *, id, (,), \$\}, P_4, S'\rangle$ with the following P_4 rules :

$$\begin{array}{lcll} S' & \rightarrow & E\$ & (0) \\ E & \rightarrow & E + T & (1) \\ E & \rightarrow & T & (2) \\ T & \rightarrow & T * F & (3) \\ T & \rightarrow & F & (4) \\ F & \rightarrow & id & (5) \\ F & \rightarrow & (E) & (6) \end{array}$$

- $LR(1)$-CFSM
- Tables

} See next slides

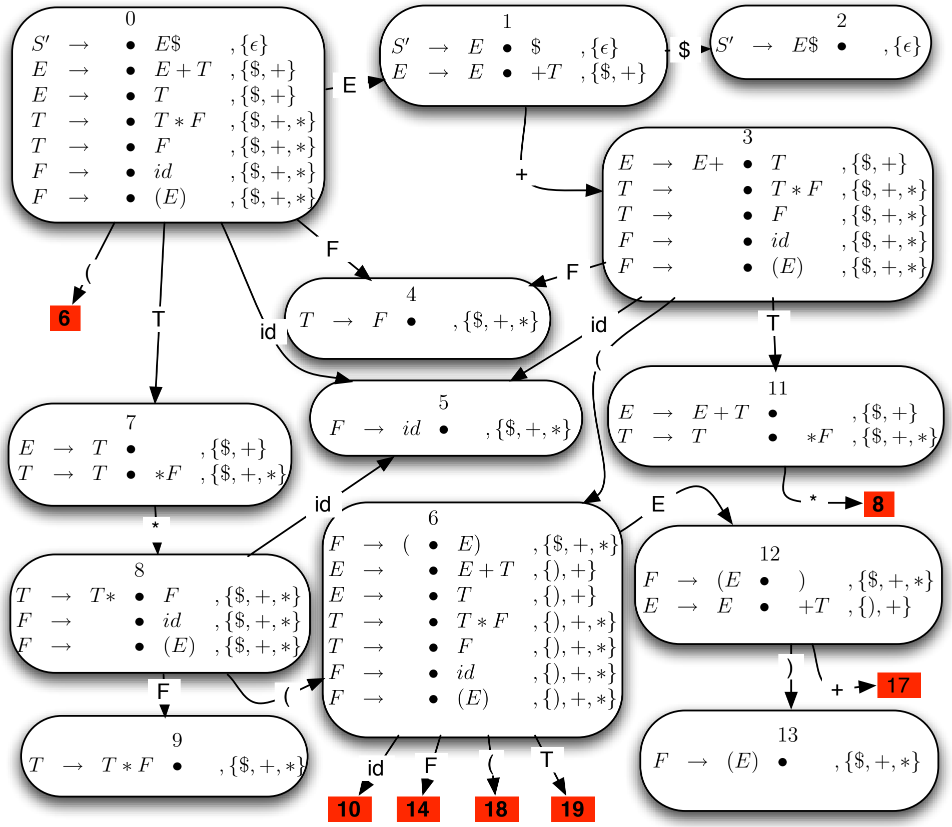
Notation: one writes $[A \rightarrow \alpha_1 \bullet \alpha_2, \{u_1, u_2, \ldots, u_n\}]$ instead of

$[A \rightarrow \alpha_1 \bullet \alpha_2, u_1]$

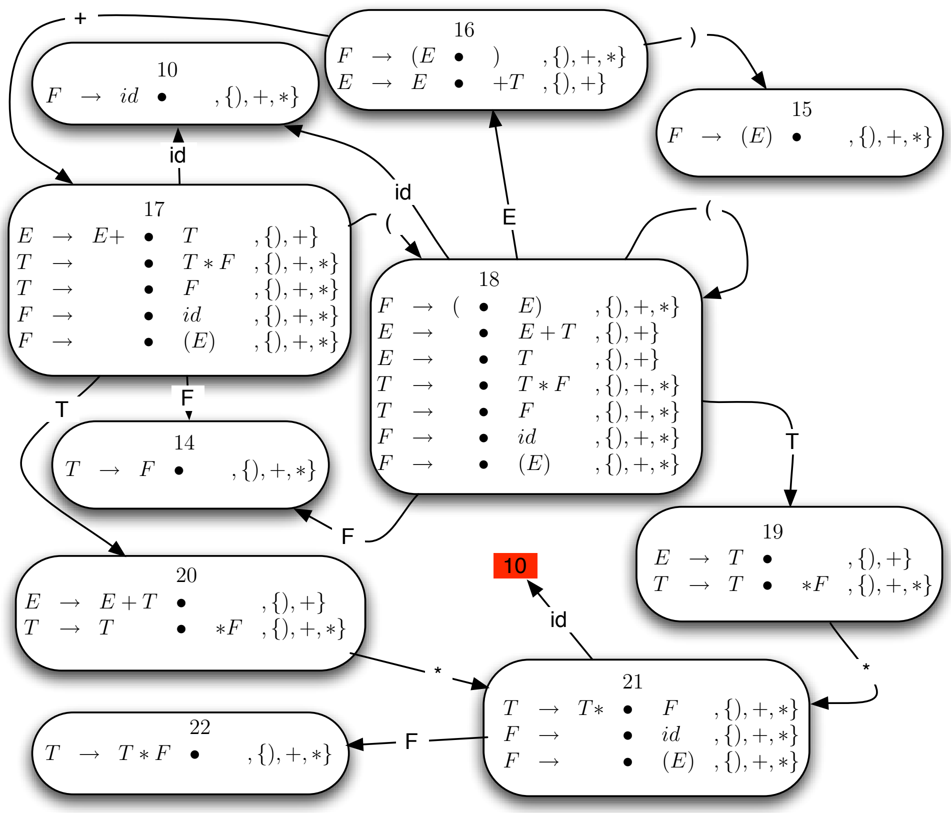
$[A \rightarrow \alpha_1 \bullet \alpha_2, u_2]$

...

$[A \rightarrow \alpha_1 \bullet \alpha_2, u_n]$



0
S' → • E$, {ε}
E → • E + T , {$, +}
E → • T , {$, +}
T → • T * F , {$, +, *}
T → • F , {$, +, *}
F → • id , {$, +, *}
F → • (E) , {$, +, *}
1
S' → E • $, {ε}
E → E • +T , {$, +}
2
S' → E$ • , {ε}
3
E → E+ • T , {$, +}
T → • T * F , {$, +, *}
T → • F , {$, +, *}
F → • id , {$, +, *}
F → • (E) , {$, +, *}
4
T → F • , {$, +, *}
5
F → id • , {$, +, *}
6
F → (• E) , {$, +, *}
E → • E + T , {), +}
E → • T , {), +}
T → • T * F , {), +, *}
T → • F , {), +, *}
F → • id , {), +, *}
F → • (E) , {), +, *}
7
E → T • , {$, +}
T → T • *F , {$, +, *}
8
T → T* • F , {$, +, *}
F → • id , {$, +, *}
F → • (E) , {$, +, *}
9
T → T * F • , {$, +, *}
11
E → E + T • , {$, +}
T → T • *F , {$, +, *}
12
F → (E •) , {$, +, *}
E → E • +T , {), +}
13
F → (E) • , {$, +, *}
E
$
+
F
(
T
id
*
)
6
8
10
14
17
18
19



10
F → id • , {), +, *}
14
T → F • , {), +, *}
15
F → (E) • , {), +, *}
16
F → (E •) , {), +, *}
E → E • +T , {), +}
17
E → E+ • T , {), +}
T → • T * F , {), +, *}
T → • F , {), +, *}
F → • id , {), +, *}
F → • (E) , {), +, *}
18
F → (• E) , {), +, *}
E → • E + T , {), +}
E → • T , {), +}
T → • T * F , {), +, *}
T → • F , {), +, *}
F → • id , {), +, *}
F → • (E) , {), +, *}
19
E → T • , {), +}
T → T • *F , {), +, *}
20
E → E + T • , {), +}
T → T • *F , {), +, *}
21
T → T* • F , {), +, *}
F → • id , {), +, *}
F → • (E) , {), +, *}
22
T → T * F • , {), +, *}
+
)
id
(
E
F
T
*
10

$LR(1)$ tables

Action

State	$+$	$*$	*id*	(	)	\$	ϵ
0			*S*	*S*			
1	*S*					*S*	
2							*A*
3			*S*	*S*			
4	*R4*	*R4*				*R4*	
5	*R5*	*R5*				*R5*	
6			*S*	*S*			
7	*R2*	*S*				*R2*	
8			*S*	*S*			
9	*R3*	*R3*				*R3*	
10	*R5*	*R5*			*R5*		
11	*R1*	*S*				*R1*	
12	*S*				*S*		
13	*R6*	*R6*				*R6*	
14	*R4*	*R4*			*R4*		
15	*R6*	*R6*			*R6*		
16	*S*				*S*		
17			*S*	*S*			
18			*S*	*S*			
19	*R2*	*S*			*R2*		
20	*R1*	*S*			*R1*		
21			*S*	*S*			
22	*R3*	*R3*			*R3*		

Successor

State	$+$	$*$	*id*	(	)	\$	*E*	*T*	*F*
0			5	6			1	7	4
1	3					2			
2									
3			5	6				11	4
4									
5									
6			10	18			12	19	14
7		8							
8			5	6					9
9									
10									
11		8							
12	17				13				
13									
14									
15									
16	17				15				
17			10	18				20	14
18			10	18			16	19	14
19		21							
20		21							
21			10						22
22									

Criticism of $LR(1)$ ($LR(k)$)

Criticism of $LR(1)$ ($LR(k)$)

- On sees that very quickly, the parser (the CFSM and the tables) becomes huge.
- Therefore alternatives have been found, more powerful than $LR(0)$ but more compact than $LR(1)$

$\Rightarrow$ The $SLR(1)$ ($SLR(k)$) and $LALR(1)$ ($LALR(k)$) parsers

Outline

Principles of construction of $SLR(k)$ parsers

Principles of construction of $SLR(1)$ parsers

The principle is very simple

1. One builds an $LR(0)$-CFSM
2. To build the *Action* table, one used the $LR(0)$-items and the global follow of the variables to determine for which look-aheads a Reduce must be done:

More precisely

- $Action[s, a]$ contains a Shift action if the state s contains $B \rightarrow \delta \bullet a\gamma$ for some variable B and strings γ and δ,
- $Action[s, a]$ contains the action Reduce i if
  - the rule i is $A \rightarrow \alpha$
  - s contains $A \rightarrow \alpha\bullet$
  - $a \in Follow(A)$

For $SLR(k)$

The same ideas can easily be extended for k symbols of look-ahead: one uses:

- $First^k(a\gamma Follow^k(B))$
- $Follow^k(A)$.

} See next slide

Construction algorithms of an $SLR(k)$ parser

Construction algorithm of the *Action* table

Construction-Action-table() :=

$\forall s \in \mathcal{C}, u \in T^{\leq k}$: $Action[s, u] \Leftarrow \emptyset$

$\forall s \in \mathcal{C}$

if $[A \rightarrow \alpha \bullet] \in s \ \wedge \ u \in Follow^k(A)$: $Action[s, u] \stackrel{\cup}{\Leftarrow} Reduce\ i$

/* where $A \rightarrow \alpha$ is the rule i */

if $[A \rightarrow \alpha \bullet a\beta] \in s \ \wedge \ u \in First^k(a\beta Follow^k(A))$:

$Action[s, u] \stackrel{\cup}{\Leftarrow} Shift$

if $[S' \rightarrow S\$\bullet] \in s$: $Action[s, \epsilon] \stackrel{\cup}{\Leftarrow} Accept$

Endproc

Principles of construction of $SLR(k)$ parsers

Example (Construction of the $SLR(1)$ parser for G'_4)

$G'_4 = \langle \{S', E, T, F\}, \{+, *, id, (,), \$\}, P_4, S' \rangle$ with rules P_4 :

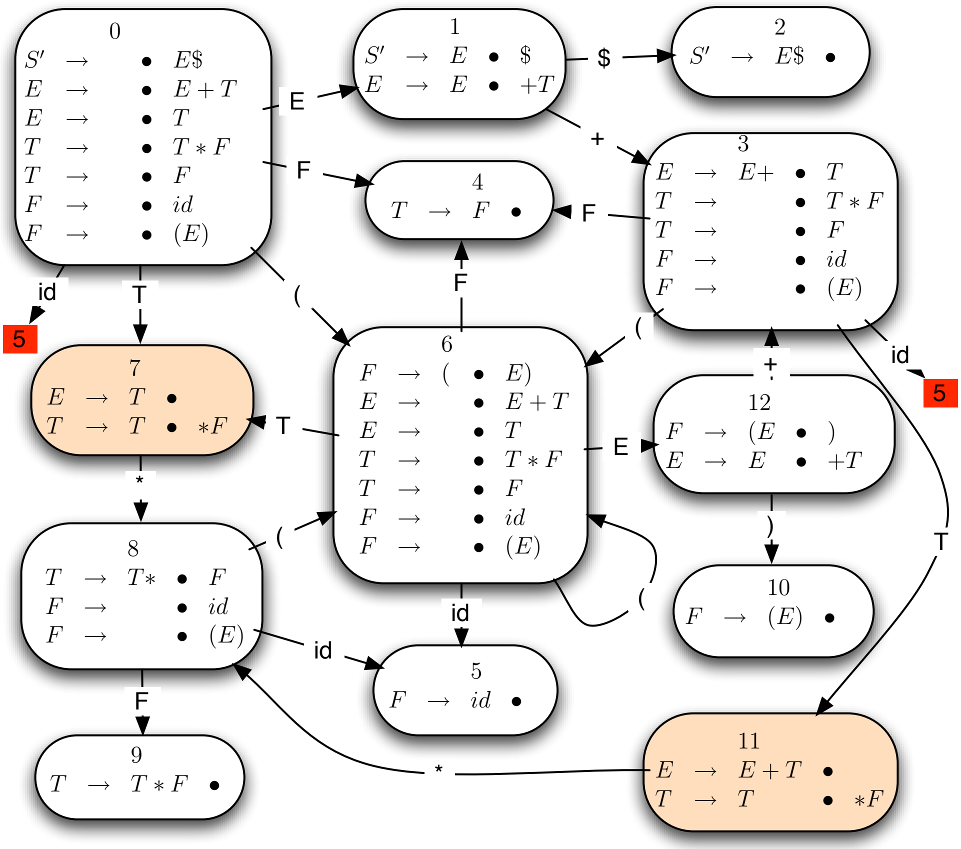
$$\begin{array}{lcll} S' & \rightarrow & E\$ & (0) \\ E & \rightarrow & E + T & (1) \\ E & \rightarrow & T & (2) \\ T & \rightarrow & T * F & (3) \\ T & \rightarrow & F & (4) \\ F & \rightarrow & id & (5) \\ F & \rightarrow & (E) & (6) \end{array}$$

We have

- $Follow(E) = \{+,), \$\}$
- $Follow(T) = \{+, *,), \$\}$
- $Follow(F) = \{+, *,), \$\}$

- $LR(0)$-CFSM
- Tables (with "global" look-aheads)

} See next slides



$SLR(1)$ tables

Action

State	$+$	$*$	*id*	(	)	\$	ϵ
0			*S*	*S*			
1	*S*					*S*	
2							*A*
3			*S*	*S*			
4	*R4*	*R4*			*R4*	*R4*	
5	*R5*	*R5*			*R5*	*R5*	
6			*S*	*S*			
7	*R2*	*S*			*R2*	*R2*	
8			*S*	*S*			
9	*R3*	*R3*			*R3*	*R3*	
10	*R6*	*R6*			*R6*	*R6*	
11	*R1*	*S*			*R1*	*R1*	
12	*S*				*S*		

Successor

State	$+$	$*$	*id*	(	)	\$	*E*	*T*	*F*
0			5	6			1	7	4
1	3					2			
2									
3			5	6				11	4
4									
5									
6			5	6			12	7	4
7		8							
8			5	6					9
9									
10									
11		8							
12	3				10				

Notice that the *Successor* table of a $SLR(k)$ parser is always equal to the one for a $LR(0)$ parser

Criticism of the $SLR(1)$ method

Limitations of the $SLR(1)$ method

The $SLR(1)$ method is simple and more powerful than the $LR(0)$ method; but, one easily can find examples where conflicts remain

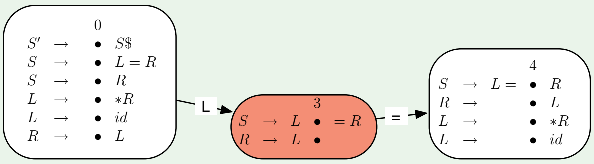
Example (Grammar which is neither $LR(0)$ nor $SLR(1)$)

$G'_5 = \langle \{S', E, T, F\}, \{+, *, id, (,), \$\}, P_5, S' \rangle$ with rules P_5 and the Follow :

S'	$\rightarrow$	$S\$$	(0)
S	$\rightarrow$	$L = R$	(1)
S	$\rightarrow$	R	(2)
L	$\rightarrow$	$*R$	(3)
L	$\rightarrow$	id	(4)
R	$\rightarrow$	L	(5)

- $Follow(S) = \{\$\}$
- $Follow(L) = \{=, \$\}$
- $Follow(R) = \{=, \$\}$

Gives the following part of CFSM :



Where one can see the conflict : $Action[3, =] = \{\text{Shift, Reduce 5}\}$

Outline

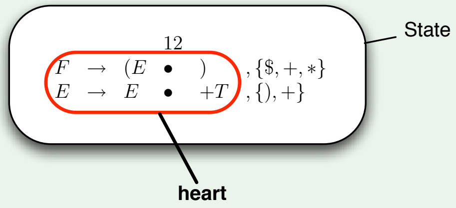
PUB Cours-Librairie, av. P. Héger 42, B-1000 Bruxelles / INFO-F-403_A

Principle of the construction of $LALR(k)$ parsers

Definition (Heart of an $LR(k)$-CFSM state)

It is the set of $LR(k)$-items of the state from which look-aheads have been removed

Example (of heart of a state s)



Principle of the construction of $LALR(k)$ parsers

The principle is very simple

1. Build the $LR(k)$-CFSM
2. merge the states with the same heart by taking the union of their $LR(k)$-items
3. The construction of the tables then keeps the algorithm for $LR(k)$

Principle of the construction of $LALR(1)$ parsers

Example (Construction of the $LALR(1)$ parser for G'_4)

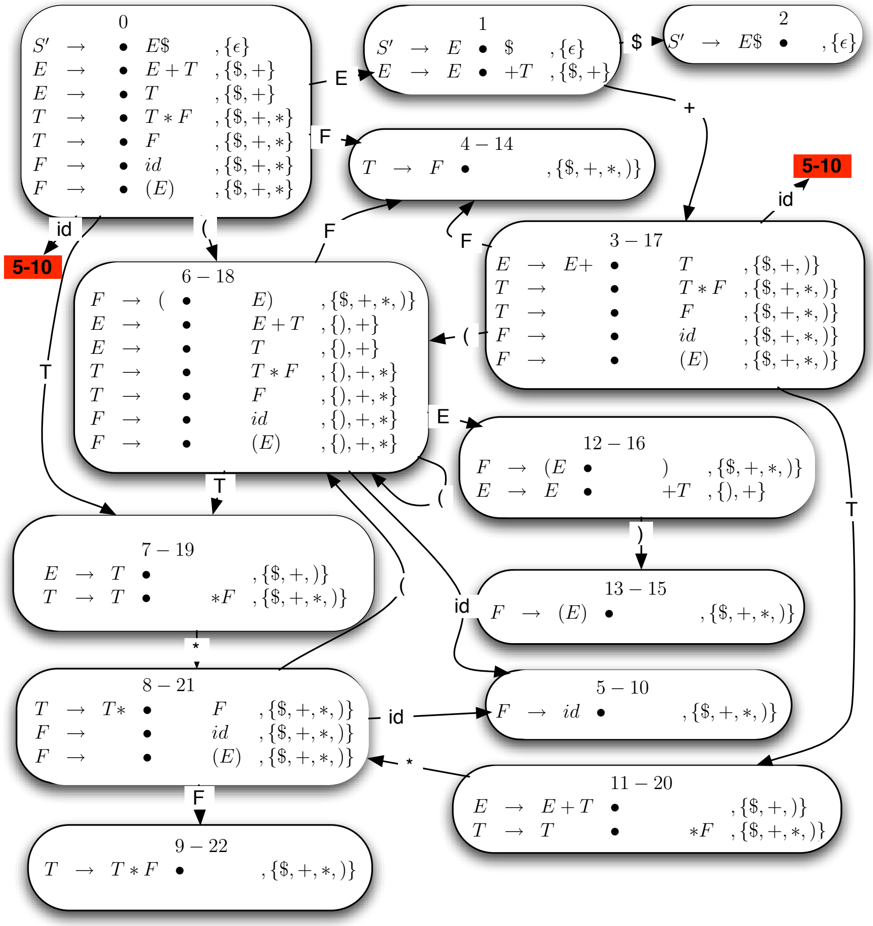
$G'_4 = \langle\{S', E, T, F\}, \{+, *, id, (,), \$\}, P_4, S'\rangle$ with rules P_4 :

$$
\begin{array}{lcll}
S' & \rightarrow & E\$ & (0)\\
E & \rightarrow & E + T & (1)\\
E & \rightarrow & T & (2)\\
T & \rightarrow & T * F & (3)\\
T & \rightarrow & F & (4)\\
F & \rightarrow & id & (5)\\
F & \rightarrow & (E) & (6)
\end{array}
$$

After the construction of the $LR(1)$-CFSM the following states have the same heart:

- 3 and 17
- 4 and 14
- 5 and 10
- 6 and 18
- 7 and 19
- 8 and 21
- 9 and 22
- 11 and 20
- 12 and 16
- 13 and 15
- $LALR(1)$-CFSM
- Tables

} See next slides



LALR(1) tables

Action

State	+	$*$	*id*	(	)	$	ϵ
0			*S*	*S*			
1	*S*					*S*	
2							*A*
3 – 17			*S*	*S*			
4 – 14	*R*4	*R*4			*R*4	*R*4	
5 – 10	*R*5	*R*5			*R*5	*R*5	
6 – 18			*S*	*S*			
7 – 19	*R*2	*S*			*R*2	*R*2	
8 – 21			*S*	*S*			
9 – 22	*R*3	*R*3			*R*3	*R*3	
13 – 15	*R*6	*R*6			*R*6	*R*6	
11 – 20	*R*1	*S*			*R*1	*R*1	
12 – 16	*S*				*S*		

$LALR(1)$ tables

Successor

State	+	*	id	(	)	$	E	T	F
0			5-10	6-18			1	7-19	4-14
1	3-17					2			
2									
3-17			5-10	6-18				11-20	4-14
4-14									
5-10									
6-18			5-10	6-18			12-16	7-19	4-14
7-19		8-21							
8-21			5-10	6-18					9-22
9-22									
13-15									
11-20		8-21							
12-16	3-17				13-15				

Features of the $LALR(k)$ method

Theorem (The merging of $LR(k)$-CFSM states is consistent)

I.e. if 2 states s_1 and s_2 must be merged in the $LALR(k)$-CFSM then $\forall X$: $Transition[s_1, X]$ and $Transition[s_2, X]$ must also be merged. Indeed, $Transition(s, X)$ only depends on the hearts of the $LR(k)$-items and not the look-aheads.

Features of the $LALR(k)$ method

Features of the $LALR(k)$ method

- For all CFGs G', if we abstract the look-aheads, the $LR(0)$-CFSM and $LALR(k)$-CFSM are the same.
- It is possible to build the $LALR(1)$-CFSM directly from the $LR(0)$-CFSM on which one directly computes the look-aheads: this method is not seen in this course.
- For the previous example, the *Actions* table of the $LALR(1)$ and $SLR(1)$ parsers are the same (modulo the states' names); it is generally not the case.
- Every $SLR(k)$ grammar is $LALR(k)$ but the inverse is not always true.
- For instance, the grammar G'_5 on slide 349 is not $SLR(1)$ but is $LR(1)$ and $LALR(1)$.

Features of the $LALR(k)$ method

An $LR(1)$ grammar is not necessarily $LALR(1)$. Indeed:

Merging states may add Reduce / Reduce conflicts

For the grammar whose rules are:

$$\begin{array}{lcll} S' & \rightarrow & S\$ & (0) \\ S & \rightarrow & aAd & (1) \\ S & \rightarrow & bBd & (2) \\ S & \rightarrow & aBe & (3) \end{array} \qquad \begin{array}{lcll} S & \rightarrow & bAe & (4) \\ A & \rightarrow & c & (5) \\ B & \rightarrow & c & (6) \end{array}$$

the following states are generated in the $LR(1)$-CFSM

- $\{[A \rightarrow c\bullet, d], [B \rightarrow c\bullet, e]\}$
- $\{[A \rightarrow c\bullet, e], [B \rightarrow c\bullet, d]\}$

whose merge produces a Reduce 5/Reduce 6 conflict

Features of the $LALR(k)$ method

Merging states cannot add Shift / Reduce conflicts

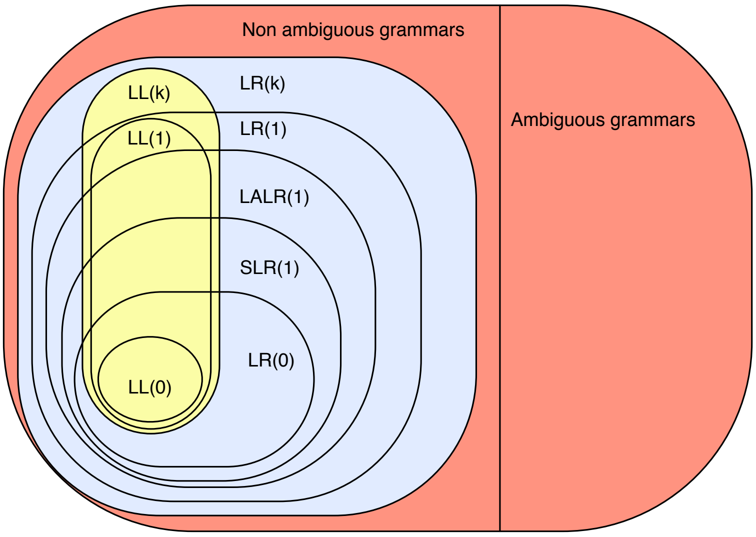
Indeed, if, in the same state of the $LALR(1)$-CFSM, one has:

- $[A \rightarrow \alpha \bullet, a]$ and
- $[B \rightarrow \beta \bullet a\gamma, b]$

  then in the state of the corresponding $LR(1)$-CFSM, which contains $[A \rightarrow \alpha \bullet, a]$
- one would have $[B \rightarrow \beta \bullet a\gamma, c]$ for some c and
- the Shift / Reduce conflict would already exist in the $LR(1)$ parser

Outline

Inclusion of the classes of grammars



Notes

- In practice, after cleaning, most grammars that we want to compile are $LALR(1)$
- One can prove that the 3 classes of languages $LR(k)$ (i.e. recognized by a $LR(k)$ grammar), $LR(1)$ and of the languages accepted by a DPDA (deterministic automaton with a stack) are the same.

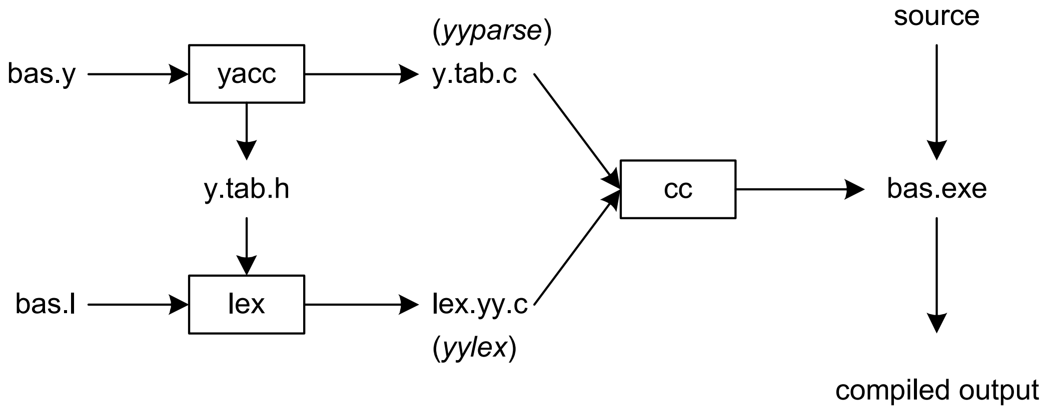
Outline

`Yacc` = **Y**et **A**nother **C**ompiler **C**ompiler (≡ Bison on GNU systems)

What does `Yacc` do?

- `Yacc` has been designed as a generator of *LALR*(1) parsers, and more generally, of parts of compilers.
- Together with the `Lex` tool, `Yacc` can build a big part of or even a full compiler.
- `Yacc` is a powerful tool to create programs which process languages whose context-free grammar is given.

General workflow for the use of `Lex` (`Flex`) and `Yacc` (`Bison`)



Compilation :

```
yacc -d bas.y                       # creates y.tab.h and y.tab.c
lex bas.l                           # creates lex.yy.c
cc lex.yy.c y.tab.c -ll -o bas.exe # compile and link
                                    # creates bas.exe
```

Yacc specification

```
declarations
%%
productions
%%
additional code
```

The resulting parser (yyparse()) tries to recognize sentences compatible with the grammar.
During the analysis by a parser generated by Yacc, *semantic actions* given with C code can be executed and *attributes* can be computed (see example and next chapter).

Example Lex and Yacc: expression evaluator

Example (Lex part of the expression evaluator)

```
/*    File ex3.l    */
%{
#include "y.tab.h"
#define yywrap() 1
extern int yylval;
%}
integer   [0-9]+
separator [\ \t]
nl        \n
%%
{integer}  { sscanf(yytext, "%d", &yylval);
             return(INTEGER);
           }
[-+*/()]   { return yytext[0]; }
quit       { return 0; }
{nl}       { return '\n'; }
{separator}        ;
.          { return yytext[0]; }
```

Example Lex and Yacc: expression evaluator

Example (Yacc part of the expression evaluator (1))

```
/*    File ex3.y      */
%{
#include <stdio.h>
%}

%token INTEGER

%%
```

Example Lex and Yacc: expression evaluator

Example (Yacc part of the expression evaluator (2))

```
lines:       /*empty*/
     |       lines line
     ;
line:        '\n'
     |       exp '\n'
             {printf(" = %d\n", $1);}

exp:         exp '+' term        {$$ = $1 + $3;}
     |       exp '-' term        {$$ = $1 - $3;}
     |       term
     ;

term:        term '*' fact       {$$ = $1 * $3;}
             term '/' fact      {$$ = $1 / $3;}
             fact
     ;

fact:        INTEGER
     |       '-' INTEGER        {$$ = - $2;}
     |       '('exp')'          {$$ = $2;}
     ;
```

Example Lex and Yacc: expression evaluator

Example (Yacc part of the expression evaluator (3))

```
%%
int yyerror()
{
     printf("syntax error\n");
     return(-1);
}
main()
{
     yyparse();
     printf("goodbye\n");

}
```


Chapter 11: Semantic analysis

PUB Cours-Librairie, av. P. Héger 42, B-1000 Bruxelles / INFO-F-403_A

Outline

1. Roles and phases of semantic analysis
2. Tools for semantic analysis
3. Construction of the AST and CFG
4. Some examples of the use of attributed grammars

Roles of semantic analysis

Definition (Role of semantic analysis)

For an imperative language, semantic analysis, also called context management, handles the non local *relations; it also addresses:*

1. *visibility control and the link between definition and uses of identifiers*
2. *type control of "objects", number and type of function parameters*
3. *control flow (verifies, for instance, that a goto is allowed - see example below)*

Example (of wrong control flow)

The following code is not allowed:

```
int main ()
{
  for(int i=1;i<10;++i)
    infor: cout << "iteration " << i << endl;
  goto infor;
}
```


First phase of the semantic analysis

Construction of the abstract syntax tree (and of the control flow graph)

Definition (AST (Abstract Syntax Tree))

Summarized form of the syntax tree which only keeps elements useful for later phases.

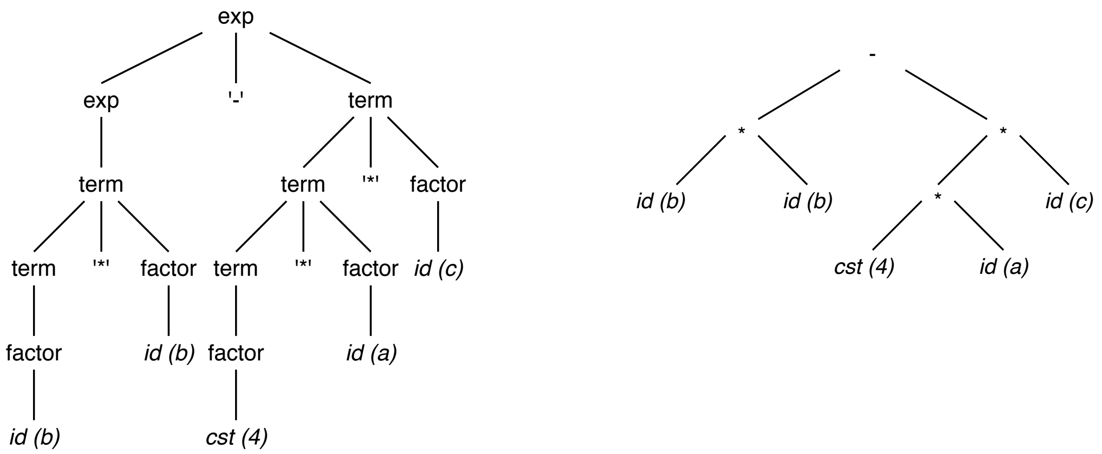
Example (of grammar, of syntax tree and of AST)

Given G with the production rules:

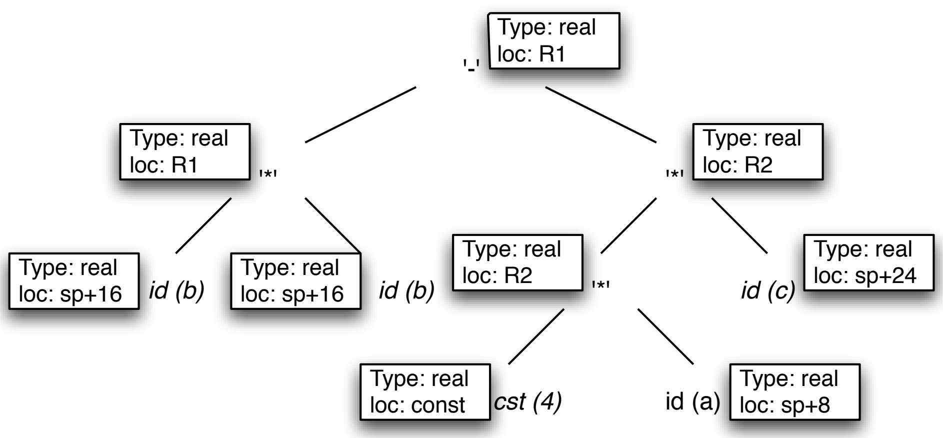
$$
\begin{array}{lcl}
exp & \rightarrow & exp + term \mid \\
 & \rightarrow & exp - term \mid \\
 & \rightarrow & term \\
term & \rightarrow & term * factor \mid \\
 & \rightarrow & term \;/\; factor \mid \\
 & \rightarrow & factor \\
factor & \rightarrow & id \mid cst \mid (\ exp\ )
\end{array}
$$

The expression b*b-4*a*c gives

the following syntax tree : the following abstract syntax tree (AST) :



This AST will be decorated during the semantic analysis and code generation phases; for instance:



This will allow to handle the context (collection of the semantic information and verification of the constraints) and afterwards to generate the code.

Second phase of the semantic analysis

Context management (semantic control)

Reminder on the role of the context management

Context management of imperative programming languages covers:

1. visibility control and the control of the link between definition and use of identifiers (through the construction and use of a symbol table)
2. type control of "objects", number and types of function parameters
3. control flow (verifies for instance that a goto is allowed)
4. the building of a completed abstract syntax tree with type information and a control flow graph to prepare the synthesis (code generation) step.

The semantic analyzer uses and computes attributes of identifiers

Attributes of an "identifier":

1. sort (constant, variable, function)
2. type
3. initial or fix value
4. scope
5. possible "localization" properties (e.g. location in memory at run-time, for use by the code generation step)

Identification of the definition which corresponds to an occurrence

Depends on the scope (each programming language has its own rules for the scope)
In Pascal / C, can be determined during the analysis with a stack of scopes (which can be merged with the symbol table).
During the analysis

- of the beginning of a block: a new scope is stacked
- of a new definition: it is put in the current scope
- of the use of an element: the corresponding definition is found by looking in the stack, from the top
- of the end of a block: the scope is popped.

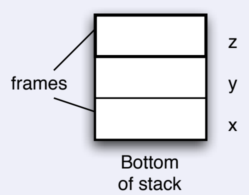
Note

This technique also allows to find the address of a variable for instance :
(number of static frames under the current frame, place in the frame)

For instance with the code:

```
{
  int x=3,y=4;
  if(x==y)
  {
    int z=8;
    y=z;
  }
}
```

`y=z` could be translated by var(1,4) ← valvar(0,0)



The notion of frame is developed in chapter 12

PUB Cours-Librairie, av. P. Héger 42, B-1000 Bruxelles / INFO-F-403_A

Other identification or control problems

Overloading

Several operators or functions with the same name. Example: the '+' can be

- +: double $\times$ double $\longrightarrow$ double
- +: int $\times$ int $\longrightarrow$ int
- +: str $\times$ str $\longrightarrow$ str
- const Matrix operator+(Matrix& m)
- ...

Roles and phases of semantic analysis
Tools for semantic analysis
Construction of the AST and CFG
Some examples of the use of attributed grammars

Polymorphism

- Several functions or methods with the same name (e.g. methods with the same name in different classes).
- "Pure" polymorphism : a function is polymorphic if it can be applied to every types (e.g. seen in functional programming languages)

Depending on the programming language, the type control and in particular the resolution of the polymorphism can be done

- statically i.e. during the semantic analysis, at compile time
- dynamically i.e. at run time, when the precise type of the "object" is known.

PUB Cours-Librairie, av. P. Héger 42, B-1000 Bruxelles / INFO-F-403_A

Note

In the context of Object Oriented programming languages, we talk about:

- overloading
  - the signature must be different to determine which method must be executed
- overriding
  - the signature must be the same
- polymorphism
  - on objects of different classes.

Other identification or control problem (cont'd)

Coercion and casting

It can happen that, during some operation, the expected type is T_1 and the the value is of type T_2.

- Either the programming language accepts to do a coercion i.e. a type conversion not explicitly asked by the programmer.
- Or the conversion must be explicitly requested using a casting operator.

Example (of casting and coercion)

```
{
  double x=3.14 y=3.14;
  int i,j;
  i = x;            // coercion: 'int -> double'
                    // a warning can be sent
  x = i;            // coercion: 'double -> int'
                    // a warning can be sent
  j = (int) y;      // casting
  y =  (double) j; // casting
}
```


Typing system

One must first:

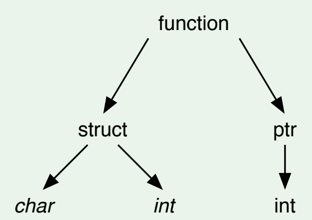
1. be able to formally define a type
2. define which types are equivalent or compatible

Definition (Type expression)

Directed graph whose nodes can be:

- *a primitive type: bool, int, double,...*
- *a constructor:*
  - *array*
  - *struct*
  - *function*
  - *pointer*
- *a type name (defined somewhere else)*
- *a type variable (a priori represents any type)*

Example (type expression)



Feature of a programming language

It can be

- statically or dynamically typed depending if it is entirely done at compile time or must be partly done at run time.
- type-safe if the only operations that can be performed on data in the language are those sanctioned by the type of the data.
- typed if the typing imposes conditions on the programs.
- strongly typed[2]
  - when it specifies one or more restrictions on how operations involving values of different data types can be intermixed or
  - [if the type information is associated with the variables and not with the values,] or
  - [if the set of types used is known at compile time and the type of the variables can be tested at run time]

[2]This term has several definitions: Programming language expert Benjamin C. Pierce has said: I spent a few weeks . . . trying to sort out the terminology of "strongly typed," "statically typed," "safe," etc., and found it amazingly difficult. . . . The usage of these terms is so various as to render them almost useless.

Type equivalence

Depending on the programming language, type equivalence can mean equivalence by name or by structure

In Pascal, C, C++

Pascal, C and C++ have equivalence by name:

- V and W are of different type from X and Y

```
struct {int i; char c;} V,W;
struct {int i; char c;} X,Y;
```

- Same here

```
typedef struct {int i; char c;} S;
S V,W;
struct {int i; char c;} X,Y;
```

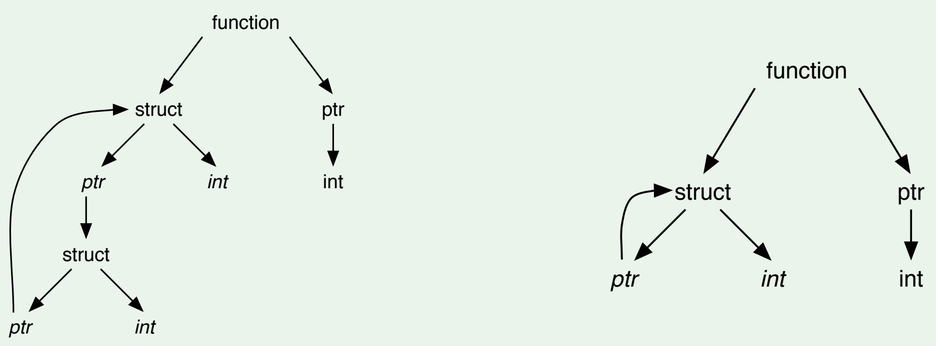
- the following variables are of equivalent types! : constant pointer to an integer

```
typedef int Veccent[100];
Veccent V,W;
int X[100],Y[10],Z[1];
```

Example of equivalence by structure

In other languages (Algol68 for instance), the equivalence is determined by the structure.

Example (Expressions structurally equivalent)



Remark:

The unification algorithm (see resolution in Prolog) allows to verify if 2 types are equivalent / compatible.

Outline

Tools for semantic analysis

Even if semantic analysis is mostly produced "by hand" (no tools to produce a complete semantic analyzer, as it is done in the lexical and syntactic analysis), two "tools" do exist.

- semantic actions
- attributed grammars

Semantic actions

Definition (Semantic actions)

Actions added to the grammar. During the parsing, analysis *of these actions corresponds to achieving the prescribed actions.*

Example (of semantic actions)

Classical example : translation of expressions in direct algebraic notation (DAN) into reverse polish notation (RPN):

$$
\begin{array}{lcl}
E & \longrightarrow & TE' \\
E' & \longrightarrow & +T\{printf('+')\}E' \\
E' & \longrightarrow & \epsilon \\
T & \longrightarrow & FT' \\
T' & \longrightarrow & *F\{printf('*')\}T' \\
T' & \longrightarrow & \epsilon \\
F & \longrightarrow & (E) \\
F & \longrightarrow & id\{printf(val(id))\}
\end{array}
$$

Attributed grammars

Definition (Attributed grammar)

It specifies specific treatments on the context-free grammar describing the syntax, which consists in the evaluation of attributes associated to the nodes of the syntax tree

Definition (Inherited and synthesized attributes)

There exists only two kind of attributes associated to a node:

- *Inherited: whose value can only depend on attributes of its parent or siblings,*
- *Synthesized: whose value can only depend on attributes of its children, or if the node is a leaf, is given by the scanner (or the parser).*

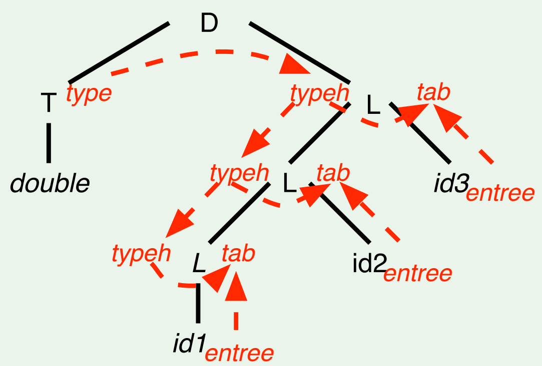
Example (of attributed grammar)

D	$\rightarrow$	$T\ L$	$L.typeh := T.type$
T	$\rightarrow$	int	$T.type := int$
T	$\rightarrow$	$double$	$T.type := double$
L	$\rightarrow$	$L_1\ id$	$L_1.typeh := L.typeh$
			$L.tab := AjouterType(L_1.tab, id.entree, L.typeh)$
L	$\rightarrow$	id	$L.tab := AjouterType(vide, id.entree, L.typeh)$

On the tree, the rules to compute the attributes give a dependency graph

- $A \dashrightarrow B$ = the value of A is used to compute B

Example:
double id1 id2 id3
Gives :



Attributed evaluation

Either the order is fixed before compilation (static order), or it is determined during compilation (dynamic order).

Naive dynamic evaluation

```
while the attributes are not all evaluated
  Evaluate (Root)

function Evaluate(S) (Rule S -> X1 X2 ... Xm) {
  1°) propagate the attributes evaluated in S
  2°) For all children Xi (i=1 -> m):     Evaluate(Xi)
  3°) for all synthetized attributes A not yet evaluated
        if the required attributes are not evaluated then
            Compute  A
}
```


Other method: **Topological sort**

Every attribute is in a predecessor list.
Each of them has its number of predecessors.
The attributes *A* without predecessors can be evaluated and the number of predecessors of the attributes which depends on *A* are decremented.

Note

Cyclic attributes are problematic for these methods

The “classical” types of attributed grammars that can be statically evaluated:

- S-attributed grammars: have only synthesized attributes (e.g. attributes computed by the parser produced by YACC)
- L-attributed grammars: computable through LR parsing (depth first search - left first). It means that an inherited attribute depends only of the left inherited attributes or the one of the parent.

Remarks: the evaluation of the attributes of these grammars can be done during an LL(1) or LALR(1) analysis.

Outline

Construction of an AST (or AS-DAG) for an expression

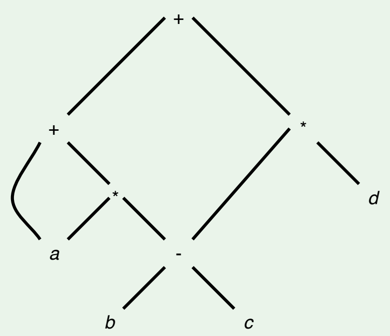
Example (of construction of an AST)

$$\begin{array}{llll} E & \rightarrow & E_1 + T & E.node := CreateNode('+', E_1.node, T.node) \\ E & \rightarrow & E_1 - T & E.node := CreateNode('-', E_1.node, T.node) \\ E & \rightarrow & T & E.node := T.node \\ T & \rightarrow & (E) & T.node := E.node \\ T & \rightarrow & id & T.node := CreateLeave(id, id.entry) \\ T & \rightarrow & nb & T.node := CreateLeave(nb, nb.entry) \end{array}$$

CreateLeave and CreateNode

- verifies if the node already exist
- returns a pointer to an existing or existing node.

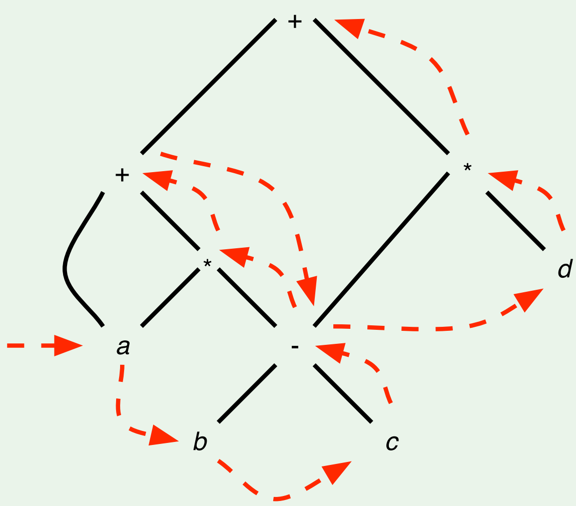
`a+a*(b-c)+(b-c)*d` gives:



Control flow graph

An extension of the attributes computation allows to obtain the Control flow graph which is the basis of several program optimisations.

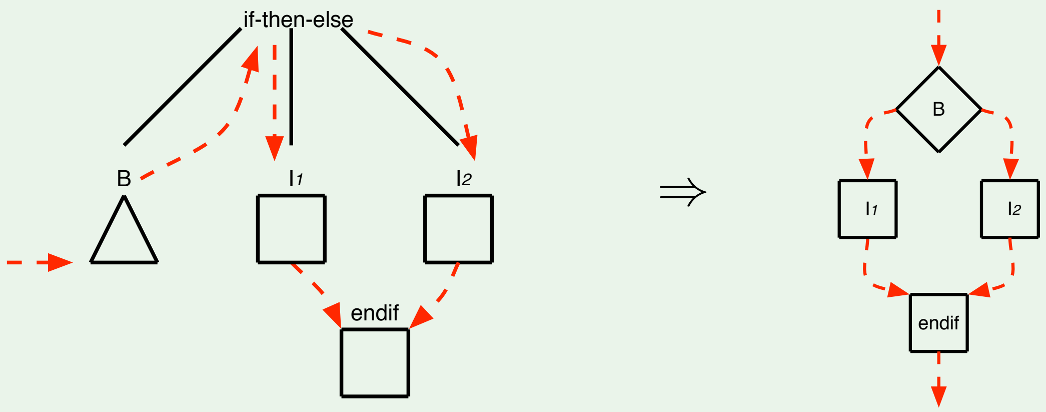
Example (of control flow graph)



Control Flow Graph

Example ((2) of control flow graph)

Control Flow Graph for if B then I_1 else I_2 endif



Control Flow Graph

Definition (Control Flow Graph)

- It is a flowchart which gives the possible flow of the instructions.
- It is composed of basic blocks.
- A basic block is a sequence of consecutive instructions without stop or connection

Outline

Calculator in Yacc

Example (Yacc)

```
S     :   E       ;
E     :   E '+' E    {$$ = $1+$3}
      | T
      ;
T     :   T '*' F    {$$ = $1*$3}
      |   F
      ;
F     :   '('E')'    {$$ = $2}
      |   nb
      ;
```

Note on Yacc

In this example,

- the default rule in YACC is `$$ = $1`
- the stack of attributes behaves like a postfix evaluation stack

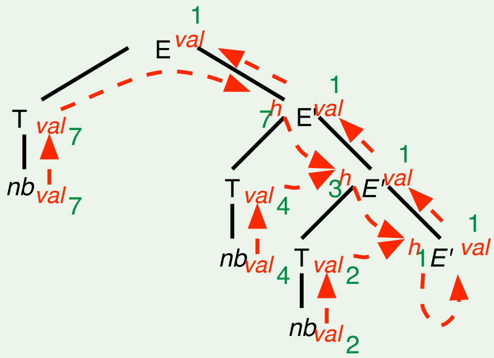
Evaluation of an expression in LL(1) analysis

Example (of attributed grammar)

With the attributed grammar :

$E \rightarrow TE'$	$E'.h := T.val$
	$E.val = E'.val$
$E' \rightarrow -TE'_1$	$E'_1.h = E'.h - T.val$
	$E'.val = E'_1.val$
$E' \rightarrow \epsilon$	$E'.val = E'.h$
$T \rightarrow nb$	$T.val = nb.val$

$7 - 4 - 2$ will be evaluated :



With a recursive (top down) LL(1) analysis, the attributes can be transmitted via input or output parameters.

Chapter 12: Code generation

Outline

Preliminary note: considered languages

Restriction

In this course, we consider that the source language is imperative and procedural

Types of languages

- Array languages
- Aspect Oriented Programming Languages
- Assembly languages
- Command Line Interface (CLI) languages (batch languages)
- Concurrent languages
- Data-oriented languages
- Dataflow languages
- Data-structured languages
- Fourth-generation languages
- Functional languages
- Declarative languages
- Logic programming languages
- Machine languages
- Macro languages
- Multi-paradigm languages
- Object-oriented languages
- Page description languages
- Procedural languages
- Rule-based languages
- Scripting languages
- Specification languages
- Syntax handling languages
- ...

Outline

Questions to ponder

1. Can functions be recursive?
2. Can a function refer to non local identifiers?
3. Can a block refer to non local identifiers?
4. Which types of parameter passing are possible?
5. Can functions be passed as parameters?
6. Can a function be returned as a result?
7. Can the program dynamically allocate memory?
8. Must the memory be freed explicitly?

To simplify we choose the following answers

1. Type(s) of parameter passing?: by value and reference
2. Recursive functions?: yes
3. Function which refers to non local id?: no
4. Block which refers to non local id?: yes
5. Functions passed as parameter?: no
6. Function returned as result?: no
7. Dynamic memory allocation?: yes
8. Memory explicitly freed?: yes

Type(s) of parameter passing?: by value and reference

Main types of parameter passing :

- By value: the formal parameter is a variable local to the function; its value is initialized with the value of the effective parameter
- By reference (or by variable): the address of the variable, passed in parameter is transmitted to the function
- By copy (in / out): the formal parameter is a variable local to the function; its value is initialized with the value of the variable given in effective parameter; at the end of the function execution, the new value is copied in the effective parameter
- By name: textual replacement of the formal parameter by the effectively transmitted parameter.

Recursive functions?: yes

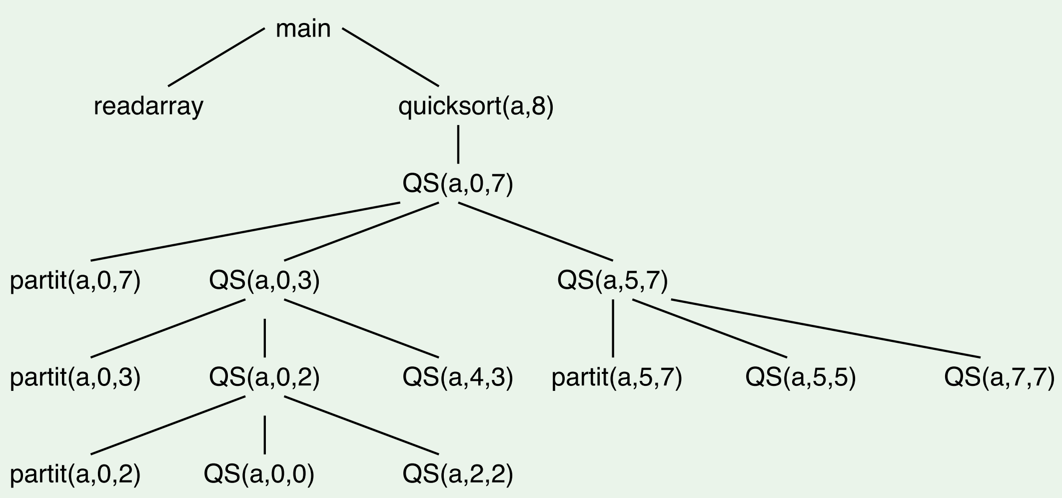
The execution of a program is sketched by its activation tree

Example (of program and of activation tree)

```
void readarray(int a[], int aSize){...}
static int partit(int a[], int first, int last) {...}
static void QS(int a[], int first, int last) {
  if (first < last) {
    int pivotIndex = partit(a, first, last);
    QS(a, first, pivotIndex - 1);
    QS(a, pivotIndex + 1, last);
  }
}
static void quicksort(int a[], int aSize) {
  QS(a, 0, aSize - 1);
}
int main() {
   ...
   readarray(a,n);
   quicksort(a,n);
}
```


Example of activation tree for quick-sort

Example (of program and of activation tree)

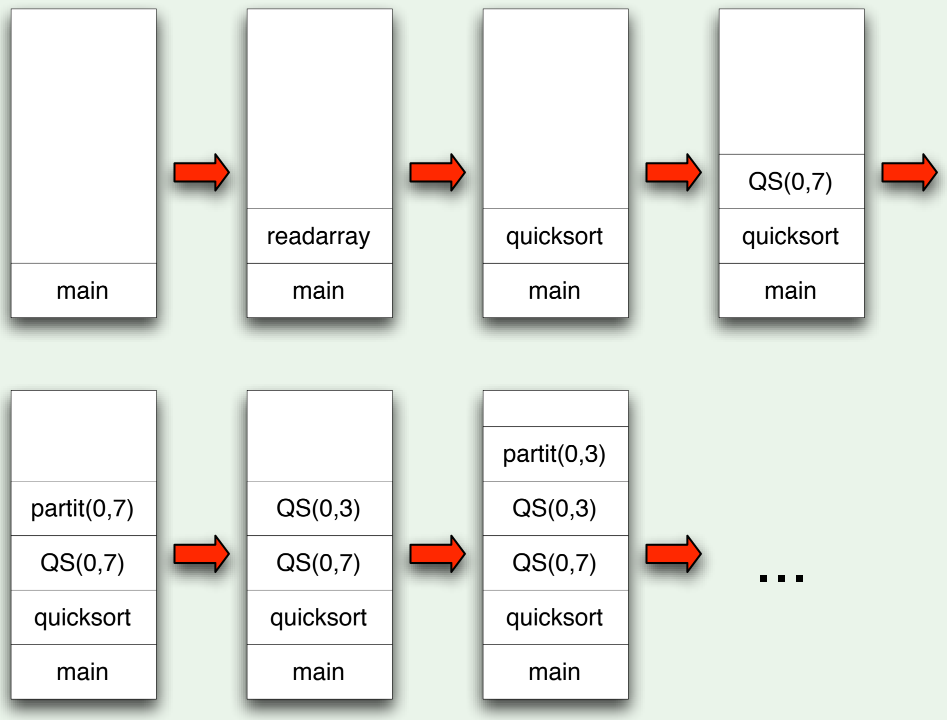


Execution = depth first search of the activation tree

the activation tree is only known at execution time
=> control stack (run-time) to record the contexts and variables local to the calling functions

Changes of the run-time stack during execution

Example (Changes of the stack)



Block refers to non local id?: yes

Activation frame

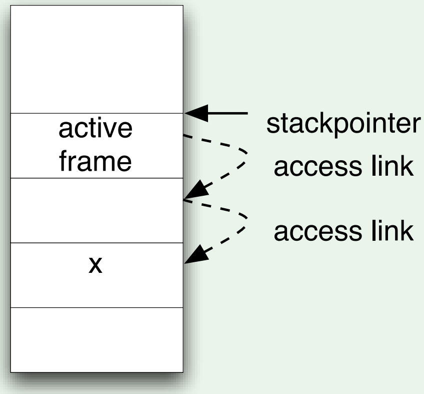
Each zone of the stack which corresponds to a block is called activation frame.

Block which refers to non local id ?: yes

- References to memory (variables for instance) will be coded by memory accesses whose addresses are relative to the begin of the frame.
- For instance, the access to a variable *x* inside the grandparent's block corresponds to an access relative to the ante-penultimate frame of the run-time stack.
- For that, each frame contains the address of the begin of the frame just below (as a simply linked list)

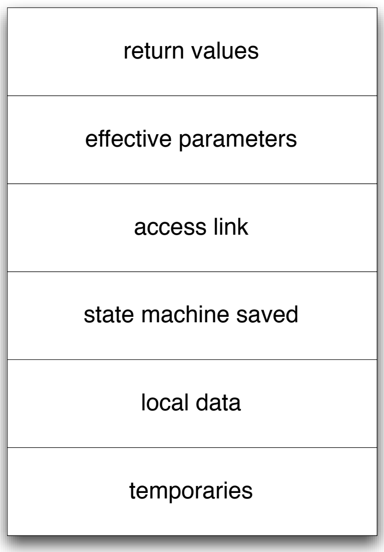
Example (of access link)

If *x* is in the "grandparent's" block, the access to *x* is done through 2 access links.



Frame structure (run-time activation frame)

Most complete case : frame for a function call



Remark on the access to an array's component

Example (of access to an array's component)

- If V is an array with 3 dimensions $n_1 \times n_2 \times n_3$,
- if we suppose that the language records the elements "line by line"
- and that the first component is $V[0, 0, 0]$
- The address of $V[i, j, k]$ is $*V + i.n_1.n_2 + j.n_2 + k$
- that can be written (by Horner): $*V + ((i.n_1) + j).n_2 + k$
- Note that to compute this expression, n_3 is not needed.

Dynamic memory allocation?: yes

Allocations are done during new
A supplementary memory zone is needed: the heap which has no structure (FIFO, LIFO, ...).
The memory zone therefore contains:

- zones allocated for the program and
- other zones available managed by run-time functions

Memory space explicitly freed?: yes (with delete)

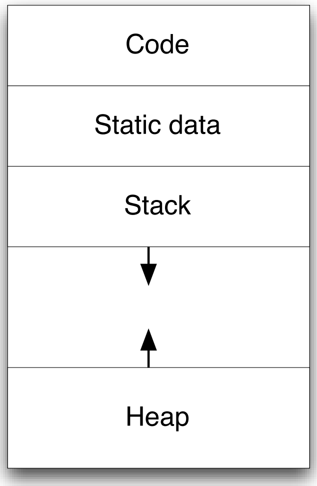
- Otherwise, a garbage collector is needed to be able to get back the parts the heap that are no more accessible by the program.
- This allows the run-time to satisfy further memory allocation requests.

Remark:

The garbage collector can interrupt the program execution unexpectedly during its work
=> problem for real-time systems where predictable timing is crucial!

Allocation of memory at run-time

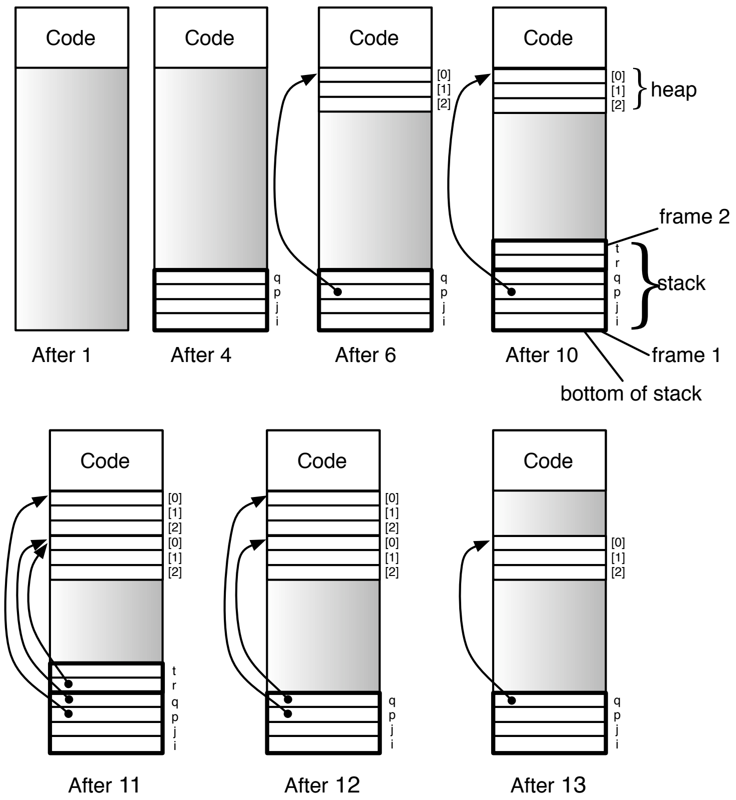
The memory of a running program is generally composed of 4 zones



An example: program and associated memory

Note:
To simplify the figures, only variables are represented on the heap

```
int main()              //1
{                       //2
  int i,j;              //3
  int *p, *q;           //4
  cin >>i;              //5
  p = new int[i];       //6
  if (i==3)             //7
  {                     //8
    int *r;             //9
    int t=4;            //10
    r=q=new int[i];     //11
  }                     //12
  delete[] p;           //13
  ...
}                       //14
```



Outline

Intermediate languages

Two classical approaches do exist:

- An intermediate language is used with 3 address instructions of general form $x := y \text{ op } z$
- Byte-code of virtual machines is used (example JVM = Java Virtual Machine). This can avoid the production of machine code altogether.

An example of simplified virtual machine : the P-machine

Defined in Wilhelm and Maurer's book; (adapted to the Pascal language).

- An evaluation and context stack
- *SP* register: pointer to highest occupied location of the stack (stack: [0..*maxstr*])
- *PC* register: pointer to the next instruction to be executed
- *EP* register: pointer to the highest location occupied throughout the execution of the procedure as a whole (used to determine possible collisions between the stack and the heap)
- *MP* register: pointer to the start of the stack frame of the current block
- *NP* register: pointer to the last occupied location on the heap
- code : [0..*codemax*]; 1 instruction per word, init: $PC = 0$
- types
  - `i` : integer
  - `r` : float (real)
  - `b` : boolean
  - `a` : address
- notations
  - *N* means numerical
  - *T* means any type or address

Expressions

Instr	Semantics	Cond	Result
`add` N	$STORE[SP-1] = STORE[SP-1] + STORE[SP]; SP--$	(N, N)	(N)
`sub` N	$STORE[SP-1] = STORE[SP-1] - STORE[SP]; SP--$	(N, N)	(N)
`mul` N	$STORE[SP-1] = STORE[SP-1] * STORE[SP]; SP--$	(N, N)	(N)
`div` N	$STORE[SP-1] = STORE[SP-1] / STORE[SP]; SP--$	(N, N)	(N)
`neg` N	$STORE[SP] = -STORE[SP]$	(N)	(N)
`and` N	$STORE[SP-1] = STORE[SP-1]$ *and* $STORE[SP]; SP--$	(b, b)	(b)
`or` N	$STORE[SP-1] = STORE[SP-1]$ *or* $STORE[SP]; SP--$	(b, b)	(b)
`not` N	$STORE[SP] =$ *not* $STORE[SP]$	(b)	(b)
`equ` T	$STORE[SP-1] = STORE[SP-1] == STORE[SP]; SP--$	(T, T)	(b)
`geq` T	$STORE[SP-1] = STORE[SP-1] >= STORE[SP]; SP--$	(T, T)	(b)
`leq` T	$STORE[SP-1] = STORE[SP-1] <= STORE[SP]; SP--$	(T, T)	(b)
`les` T	$STORE[SP-1] = STORE[SP-1] < STORE[SP]; SP--$	(T, T)	(b)
`grt` T	$STORE[SP-1] = STORE[SP-1] > STORE[SP]; SP--$	(T, T)	(b)
`neq` T	$STORE[SP-1] = STORE[SP-1] != STORE[SP]; SP--$	(T, T)	(b)

Example: `add i`

Load and store

Instr	Semantics	Cond	Result
`ldo` $T\ q$	$SP++; STORE[SP] = STORE[q]$	$q \in [0..maxstr]$	(T)
`ldc` $T\ q$	$SP++; STORE[SP] = q$	$Type(q) = T$	(T)
`ind` T	$STORE[SP] = STORE[STORE[SP]]$	(a)	(T)
`sro` $T\ q$	$STORE[q] = STORE[SP]; SP--$	(T) $q \in [0..maxstr]$	
`sto` T	$STORE[STORE[SP-1]] = STORE[SP]; SP = SP - 2$	(a, T)	

Use

- `ldo` : put the word at address q on top of stack
- `ldc` : put the constant q on top of stack
- `ind` : replaces the address on top of stack by the content of the corresponding word
- `sro` : put the top of stack at address q
- `sto` : put the value on top of stack at the address given just below on the stack

Code for `x = y;`

Stack ← `@x`; Stack ← `y`; `sto i`

Jumps

Instr	Semantics	Cond	Result
`ujp` q	$PC = q$	$q \in [0..codemax]$	
`fjp` q	if $STORE[SP] == false$ then $PC = q$ fi $SP--$	(b) $q \in [0..codemax]$	
`ixj` q	$PC = STORE[SP] + q; SP--$	(i)	

Memory allocation and address computations (static or dynamic arrays)

Instr	Semantics	Cond	Result
`ixa` q	$STORE[SP-1] = STORE[SP-1]+$ $STORE[SP] * q;\ SP--$	(a, i)	(a)
`inc` $T\ q$	$STORE[SP] = STORE[SP] + q$	(T) and $type(q) = i$	(T)
`dec` $T\ q$	$STORE[SP] = STORE[SP] - q$	(T) and $type(q) = i$	(T)
`chk` $p\ q$	if $(STORE[SP] < p)$ or $(STORE[SP] > q)$ then error("value out of range") fi	(i, i)	(i)
`dpl` T	$SP++; STORE[SP] = STORE[SP-1]$	(T)	(T, T)
`ldd` q	$SP++;$ $STORE[SP] = STORE[STORE[SP-3] + q]$	$(a, T1, T_2)$	(a, T_1, T_2, i)
`sli` T_2	$STORE[SP-1] = STORE[SP];\ SP--$	(T_1, T_2)	(T_2)
`new`	if $(NP - STORE[SP] \leq EP)$ then error("store overflow") fi else $NP = NP - STORE[SP];$ $STORE[STORE[SP-1]] = NP;$ $SP = SP - 2;$ fi	(a, i)	

With by definition
$base(p, a) \equiv if\ (p == 0)\ then\ a\ else\ base(p-1, STORE[a+1])$

Instr	Semantics	Comments
`lod` $T\ p\ q$	$SP++;\ STORE[SP] = STORE[base(p, MP) + q]$	load value
`lda` $p\ q$	$SP++;\ STORE[SP] = base(p, MP) + q$	load address
`str` $T\ p\ q$	$STORE[base(p, MP) + q] = STORE[SP];\ SP--$	store
`mst` p	$STORE[SP+2] = base(p, MP);$ $STORE[SP+3] = MP;$ $STORE[SP+4] = EP;$ $SP = SP + 5$	static link dynamic link save EP
`cup` $p\ q$	$MP = SP - (p+4);$ $STORE[MP+4] = PC;$ $PC = q$	p is the location for the parameters save the address of return branch in q
`ssp` p	$SP = MP + p - 1$	p = place for the static variables
`sep` p	$EP = SP + p;$ $if\ EP \geq NP$ $then\ error(\text{"store overflow"})\ fi$	p is the max depth of the stack collision control stack / heap
`ent` $p\ q$	$SP = MP + q - 1;$ $EP = SP + p;$ $if\ EP \geq NP$ $then\ error(\text{"store overflow"})\ fi$	q data zone p is the max depth of the stack collision control stack / heap

Stack management (variables, procedures,...)

Use

- `lod` : put the value of address (p, q) on the stack : p static link, q offset in the frame
- `lda` : idem but the address of the word is put on the stack
- `str` : store
- `mst` : put on the stack: static and dynamic link, *EP*
- `cup` : branches with saving of the return address and update of *MP*
- `ssp` : allocation on the stack of p entries
- `sep` : controls if the stack can be increased by p locations
- `ent` : execution in raw of `ssp` and `sep`

Static and dynamic links: see reference (Wilhelm and Maurer).

Stack management (procedures, parameter passing,...)

Instr	Semantics	Comments
`retf`	$SP = MP$;	result of the function on the stack
	$PC = STORE[MP + 4]$;	return
	$EP = STORE[MP + 3]$;	restores EP
	if $EP \geq NP$ *then*	
	error(“*store overflow*”) *fi*	
	$MP = STORE[MP + 2]$	restores dynamic link
`retp`	$SP = MP - 1$;	procedure without result
	$PC = STORE[MP + 4]$;	return
	$EP = STORE[MP + 3]$;	restores EP
	if $EP \geq NP$ *then*	
	error(“*store overflow*”) *fi*	
	$MP = STORE[MP + 2]$	restores dynamic link

Preliminary note: considered languages
Features and memory management of imperative languages
Intermediate code
Processor architecture
Code generation

Stack management (procedures, parameter passing,...)

Instr	Semantics	Cond	Results
`movs` q	*for* $(i = q - 1; i \geq 0; - - i)$	(a)	
	$STORE[SP + i] = STORE[STORE[SP] + i]$;		
	od		
	$SP = SP + q - 1$		
`movd` q	*for* $(i = 1; i \leq STORE[MP + q + 1]; + + i)$		
	$STORE[SP + i] = STORE[STORE[MP + q]+$		
	$STORE[MP + q + 2] + i - 1]$;		
	od		
	$STORE[MP + q] = SP + 1 - STORE[MP + q + 2]$;		
	$SP = SP + STORE[MP + q + 1]$;		

Use

- `movs` : copies a block of data of fixed size on the stack
- `movd`: copies a block of size known at execution time

Stack management (procedures, parameter passing,...)

With $base(p, a) = if\ p = 0\ then\ a\ else\ base(p-1, STORE[a+1])$

Instr	Semantics	Comments
`smp` p	$MP = SP - (p+4);$	set MP
`cupi` $p\ q$	$STORE[MP+4] = PC;$ $PC = STORE[base(p, STORE[MP+2]+q]$	returns address
`mstf` $p\ q$	$STORE[SP+2] = STORE[base(p, MP)+q+1];$ $STORE[SP+3] = MP;$ $STORE[SP+4] = EP;$ $SP = SP+5$	dynamic link saves EP

Label, I/O, and stop

Instr	Semantics	Comments
`define` @i		@i = address of the next instruction
`prin`	$Print(STORE[SP]); SP--$	print the top of the stack
`read`	$SP++; STORE[SP] = integer\ input$	read an integer and put it on the stack
`stp`		end of program

Outline

Kinds of processors

Mainly, two big classes of processors exist:

- Register machines
  Instructions use a set of registers to achieve the computation. Generally the registers are specialized:
  - universal registers
  - floating point registers
  - predicate registers (condition code) (1 bit)
  - program counter (pointer to the next instruction) (PC)
  - stack pointer (SP)
  - status and control registers
  - data and address register (instead of Universal registers)
  - ...

  Instructions are generally 1, 2 or 3 operand.
  (of type opcode a1 a2 a3).

Kinds of processors

- Stack machines
  Instructions use an evaluation stack which allows to achieve most of the operations (calculations, branches, ...)
  (e.g. Java Virtual Machine (JVM))

Kind of processors (cont'd)

We also distinguish two types of processors:

- CISC (Complex Instruction Set Computers): have a huge instruction set with, generally, specialized registers (e.g. x86, Pentium, 68k)
- RISC (Reduced Instruction Set Computers): have a limited instruction set and, in general, a lot of universal registers (e.g. SPARC, PowerPC, ARM, MIPS, new architectures, ...)

Outline

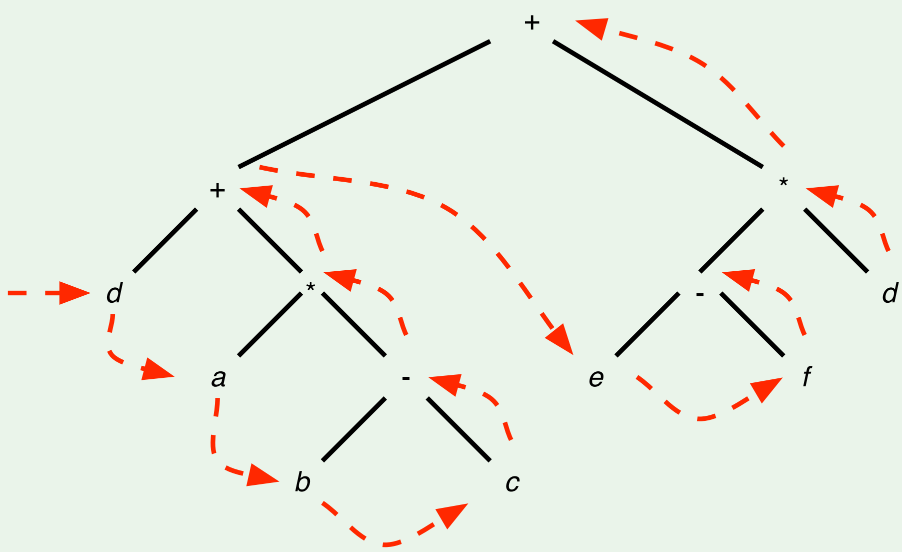
Reminder: results of semantic analysis

The output of the semantic analyser includes:

- a decorated abstract syntax tree (AST)
- (part of) a control flow graph
- a structured symbol table which allows to determine the scope of each identifier

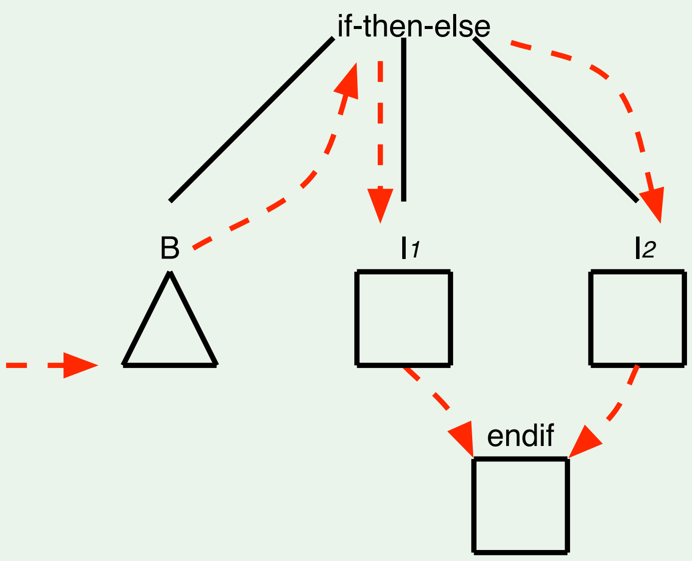
Reminders: AST and control flow graph

Example (decorated AST and control flow graph of the expression $d + a * (b - c) + (e - f) * d$)



Reminders: AST and control flow graph

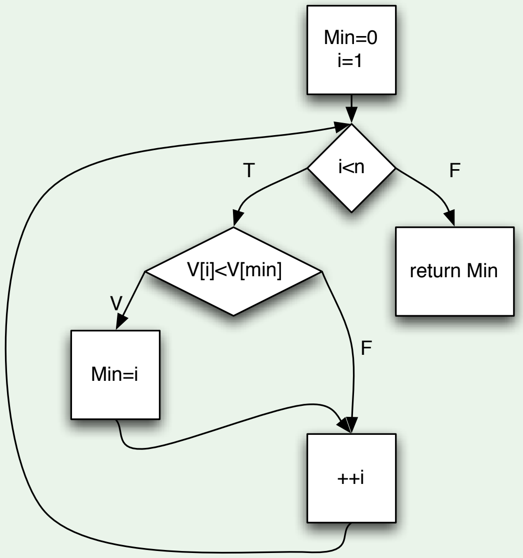
Example (decorated AST and control flow graph of an instruction if B then I_1 else I_2 endif)



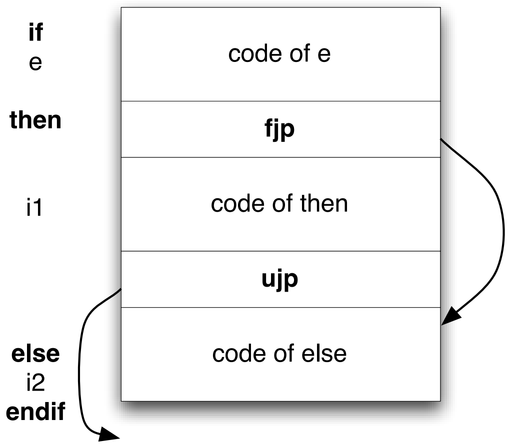
Example of control flow graph

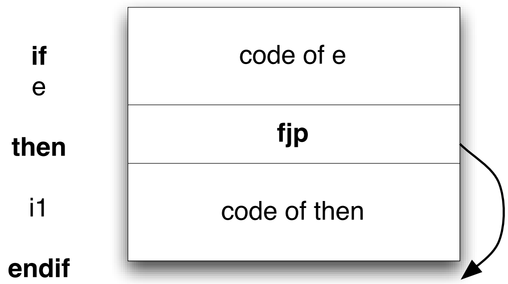
Example (Control flow graph corresponding to a simple code)

```
{
  int Min=0;
  for (int i=1; i<n; ++i)
    if (V[i]<V[Min])
      Min = i;
  return Min;
}
```

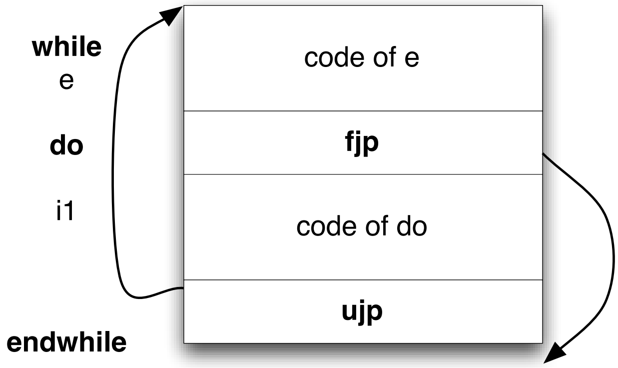


Code corresponding to an if





Code corresponding to a while



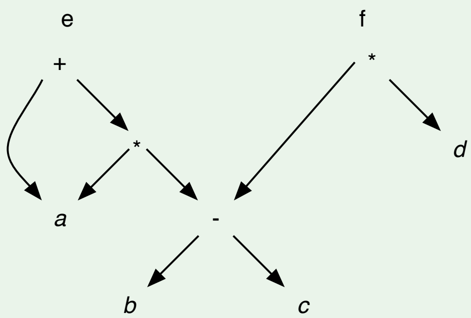
Representation of a basis block as a Directed Acyclic Graph (DAG)

The DAG contains

- one node for each operator and
- a leaf for each used variable / constant

Example (code of a block and associated DAG)

```
e = a+a*(b-c);
f = (b-c)*d;
```



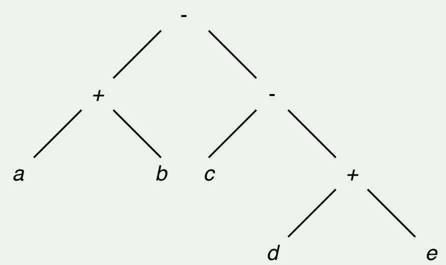
PUB Cours-Librairie, av. P. Héger 42, B-1000 Bruxelles / INFO-F-403_A

Code generation corresponding to a basis block

- Using the AST (or AS-DAG),
- some algorithms optimize the use of registers and
- the order in which operations are evaluated

Example (For $(a+b)-(c-(d+e))$ with 3 registers)

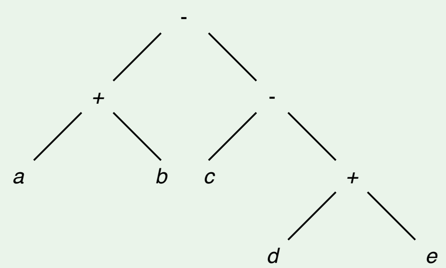
```
lw  R1 , a
lw  R2 , b
add R1 , R1 , R2
lw  R2 , d
lw  R3 , e
add R2 , R2 , R3
lw  R3 , c
sub R2 , R3 , R2
sub R1 , R1 , R2
```



Code corresponding to an expression

Example (For $(a+b)-(c-(d+e))$ with 2 registers)

```
lw  R1 , a
lw  R2 , b
add R1 , R1 , R2
sw  R1 , T
lw  R1 , d
lw  R2 , e
add R1 , R1 , R2
lw  R2 , c
sub R1 , R2 , R1
lw  R2 , T
sub R1 , R2 , R1
```



Chapter 13: Turing machines

Outline

Definition of Turing Machines (TM)

Definition (Turing Machine (TM))

$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$
with

- Q: *finite set of states*
- Σ: *finite input alphabet*
- Γ: *finite tape alphabet with* $\Sigma \subset \Gamma$
- δ: *transition function* $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times D$ *with* $\delta(p, X)$ *(not complete) and* $D \in \{L, R\}$ *(Left,Right)*
- q_0 *initial state*
- B *the* blank *symbol with* $B \in \Gamma \backslash \Sigma$
- F *the set of accepting states with* $F \subseteq Q$

Language of a TM

$$L(M) = \{w \in \Sigma^* | q_0 w \overset{*}{\vdash} \alpha p \beta \ \wedge p \in F\}$$

When the TM reaches an accepting state, it stops.

Example of TM

Example

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_4\})$$

δ	Symbol				
State	0	1	X	Y	B
q_0	(q_1, X, R)	–	–	(q_3, Y, R)	–
q_1	$(q_1, 0, R)$	(q_2, Y, L)	–	(q_1, Y, R)	–
q_2	$(q_2, 0, L)$	–	(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	(q_4, B, R)
q_4	–	–	–	–	–

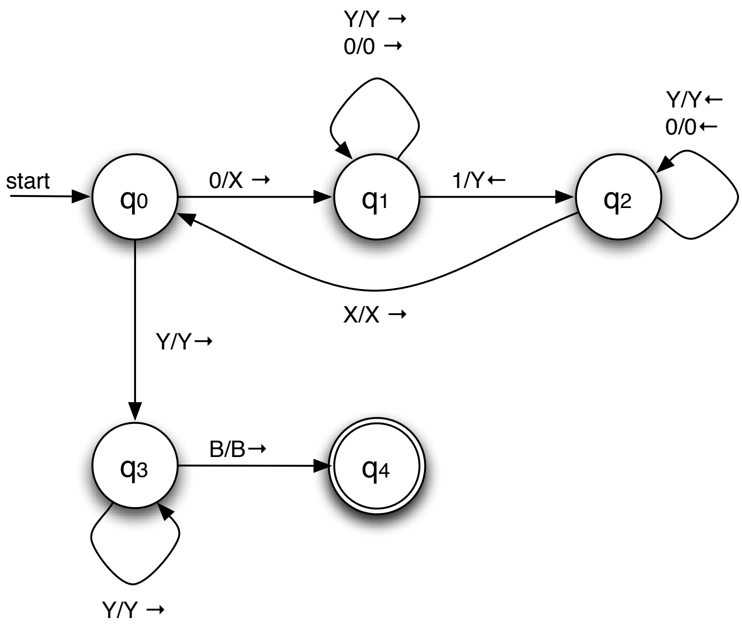
Example (Accepted sequence: 0011)

$q_0 0011 \vdash Xq_1 011 \vdash X0q_1 11 \vdash Xq_2 0Y1 \vdash q_2 X0Y1 \vdash Xq_0 0Y1 \vdash XXq_1 Y1 \vdash XXYq_1 1 \vdash XXq_2 YY \vdash Xq_2 XYY \vdash XXq_0 YY \vdash XXYq_3 Y \vdash XXYYq_3 B \vdash XXYYBq_4 B$

Example (Non accepted sequence: 0010)

$q_0 0011 \vdash Xq_1 010 \vdash X0q_1 10 \vdash Xq_2 0Y0 \vdash q_2 X0Y0 \vdash Xq_0 0Y0 \vdash XXq_1 Y0 \vdash XXYq_1 0 \vdash XXY0q_1 B$

Transition diagram of a TM



Recursively enumerable (RE) and Recursive (R)

Definition (Recursively enumerable Language (RE))

Language accepted by a TM, i.e. language composed of all the strings accepted by a TM

Definition (Recursive Language (R))

Language decided by a TM, i.e. language composed of all the strings accepted by a TM which stops for all the inputs.

Note

A recursive language can be seen as a language which has an algorithm (effective procedure) to recognize its words

Outline

Equivalence TM and language of class 0

Theorem (The languages of class 0 are RE)

Proof:
With $G = < N, \Sigma, P, S >$,
one can build a non deterministic TM M with two tapes which accepts the same language.

- *the first tape contains the input string w*
- *the second is used to to put the sentential form* α

Equivalence TM and class 0 language

Theorem ((cont'd) the class 0 languages are RE)

Operation of M:

Init *Initially S is put on the second tape, then, the TM*

1. *non deterministically selects a position i in α*
2. *non deterministically selects a production $\beta \rightarrow \gamma$ de G*
3. *if β is in position i in α, replaces β by γ by shifting what follows α (left or right)*
4. *compares the obtained sentential form with w on the first tape:*
   - *if both match, w is accepted*
   - *else, goes back to step 1*

Equivalence TM and class 0 language

Theorem (The RE class is included in the class 0)

Principle of the proof:
With $M =< Q, \Sigma, \Gamma, \delta, q_0, B, F >$,
$G =< N, \Sigma, P, S >$ *is built with* $L(G) = L(M)$.

Strict inclusion of the type 2 class of languages (context-free) in the type 1 class of languages (context-sensitive)

Reminder and properties

- A context-sensitive grammar has rules of the form $\alpha \rightarrow \beta$ with $|\alpha| \leq |\beta|$
- A language is context-sensitive if it is defined by a context-sensitive grammar
- The context-free languages are included in the context-sensitive languages
- Some context-sensitive languages are not context-free (the inclusion is strict)

Example ($L_{02^i} = \{0^{2^i} \mid i \geq 1\}$ is context-sensitive but not context-free)

A grammar for L_{02^i}:

1. $S \rightarrow DF$
2. $S \rightarrow DH$
3. $DH \rightarrow 00$
4. $D \rightarrow DAG$

1. $GF \rightarrow AF$
2. $AF \rightarrow H0$
3. $AH \rightarrow H0$
4. $GA \rightarrow AAG$

Languages R versus class 1 (context-sensitive)

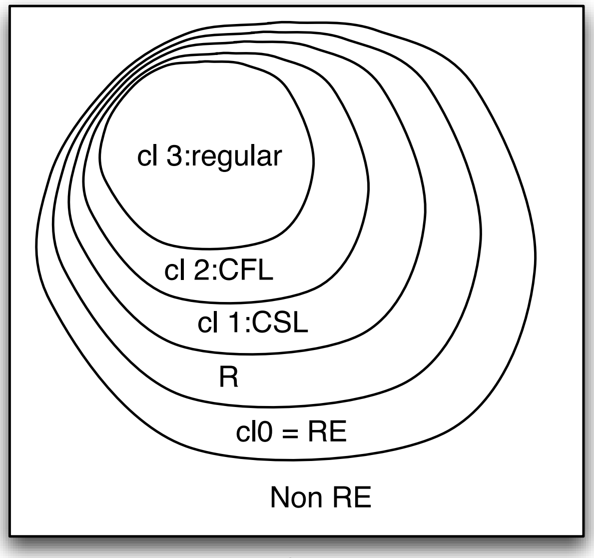
R vs context sensitive

One can prove that:

- Every context-sensitive language (class 1) is recursive.
  - With $G =< N, \Sigma, P, S >$ a grammar with the rules $\alpha \rightarrow \beta$ where $|\beta| \geq |\alpha|$, and w,
  - a graph of all sentential forms accessible from S and of size $\leq |w|$ can be built.
  - and we can decide the “accessibility” $S \overset{*}{\Rightarrow} w$
- Some recursive languages are not of class 1 (not proven here)

Inclusion of classes of languages

In summary we have :



set of languages

Introduction to Language Theory and Compilation

Exercises

Academic year 2011-2012

PUB Cours-Librairie, av. P. Héger 42, B-1000 Bruxelles / INFO-F-403_A

Session 1: Regular languages

For theory reminders, refer to chapter(s) 2.

Some exercises of this session are taken or adapted from the exercises of the *Introduction to automata theory, languages and computation* textbook, second edition, by J. Hopcroft, R. Motwani, J. Ullman. Addison-Wesley, 2000.

Ex. 1. Consider the alphabet $\Sigma = \{0, 1\}$. Using the inductive definition of regular languages, prove that the following languages are regular:

1. The set of words made of an arbitrary number of ones, followed by 01, followed by an arbitrary number of zeroes.

2. The set of odd binary numbers.

Ex. 2. Prove that any finite language is regular. Is the language $L = \{0^n 1^n \mid n \in \mathbb{N}\}$ regular? Explain.

Ex. 3. For each of the following languages (defined on the alphabet $\Sigma = \{0, 1\}$), design a nondeterministic finite automaton (NFA) that accepts it.

1. The set of strings ending with 00.

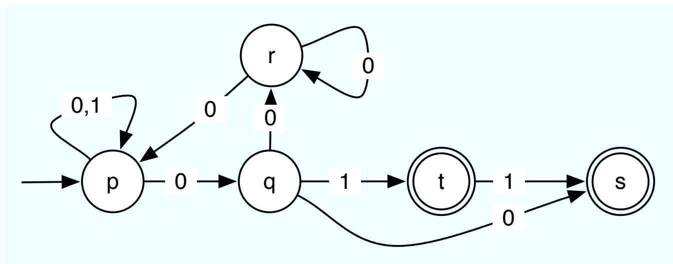
2. The set of strings whose 10^{th} symbol, counted from the end of the string, is a 1.

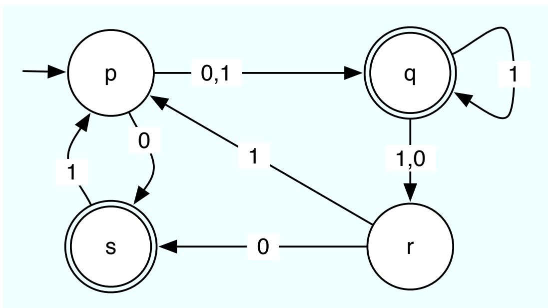
3. The set of strings where each pair of zeroes is followed by a pair of ones.

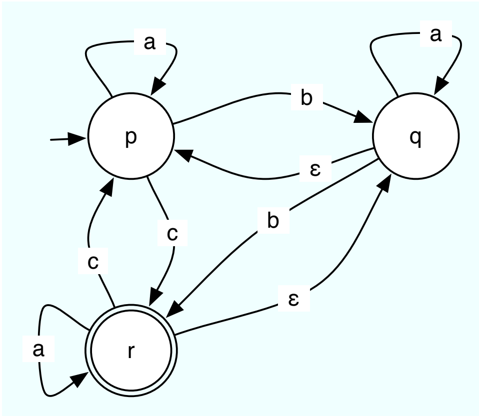
4. The set of strings not containing 101.

5. The set of binary numbers divisible by 4.

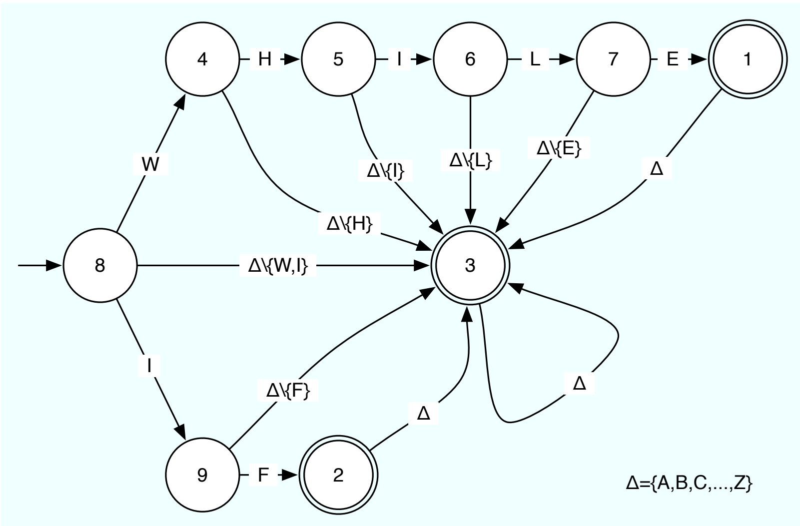
Ex. 4. Transform the following (ε-)NFAs into DFAs:







Ex. 5. Write a C/C++/Java function that implements the following automaton and returns the accepting state number.



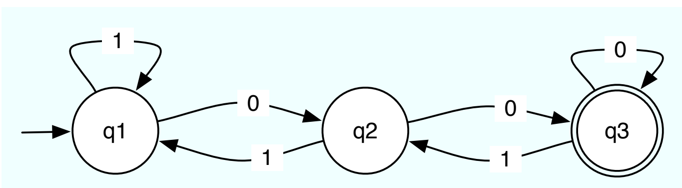
Session 2: Regular expressions

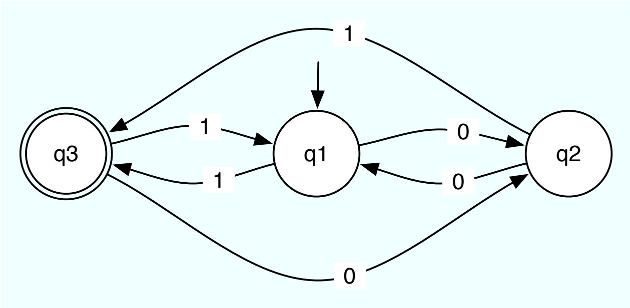
For theory reminders, refer to chapter(s) 2 and 3.

Ex. 1. For each of the following languages (defined on the alphabet $\Sigma = \{0, 1\}$), design a regular expression that recognizes it:

1. The set of strings ending with 00.
2. The set of strings whose 10^{th} symbol, counted from the end of the string, is a 1.
3. The set of strings where each pair of zeroes is followed by a pair of ones.
4. The set of strings not containing 101.
5. The set of binary numbers divisible by 4.

Ex. 2. For each of the following DFAs, give a regular expression accepting the same language:





Ex. 3. Convert the following REs into ε-NFAs:

1. 01^*
2. $(0 + 1)01$
3. $00(0 + 1)^*$

Ex. 4.
1. Give an extended regular expression (ERE) that targets any sequence of 5 characters, including the newline character `\n`
2. Give an ERE that targets any string starting with an arbitrary number of \ followed by any number of `*`
3. UNIX-like shells (such as `bash`) allow the user to write *batch* files in which comments can be added. A line is defined to be a comment if it starts with a `#` sign. What ERE accepts such comments?
4. Design an ERE that accepts numbers in scientific notation. Such a number must contain at least one digit and has two optional parts:
   - A "decimal" part : a dot followed by a sequence of digits
   - An "exponential" part: an `E` followed by an integer that may be prefixed by + or −
   - Examples : `42`, `66.4E-5`, `8E17`, `...`
5. Design an ERE that accepts "correct" phrases that fulfill the following criteria:
   - The first word must start with a capital letter
   - The phrase must end with a full stop **.**
   - The phrase must be made of one or more words (made of the characters `a...z` and `A...Z`) separated by a single space
   - There cannot be two phrases on the same line.

   Punctuation signs other than a full stop are not allowed.
6. Craft an ERE that accepts old school DOS-style filenames (8 characters in `a...z`, `A...Z` and _) whose extension is `.ext` and that begin with the string `abcde`. We ask that the ERE only accept the filename *without the extension!*
   - Example: on `abcdeLOL.ext`, the ERE must accept `abcdeLOL`

Session 3: Introduction to grammars

For theory reminders, refer to chapter(s) 4 through 6.

Ex. 1. Informally describe the languages generated by the following grammars and also specify what kind of grammars (in terms of the Chomsky hierarchy) they are:

1. $$\begin{array}{|rcl|}\hline S & \rightarrow & abcA \\ & & Aabc \\ A & \rightarrow & \varepsilon \\ Aa & \rightarrow & Sa \\ cA & \rightarrow & cS \\ \hline\end{array}$$

2. $$\begin{array}{|rcl|}\hline S & \rightarrow & 0 \\ & & 1 \\ & & 1S \\ \hline\end{array}$$

3. $$\begin{array}{|rcl|}\hline S & \rightarrow & a \\ & & *SS \\ & & +SS \\ \hline\end{array}$$

Ex. 2. Let G be the following grammar:

$$\begin{array}{rcl} S & \rightarrow & AB \\ A & \rightarrow & Aa \\ & & bB \\ B & \rightarrow & a \\ & & Sb \end{array}$$

1. Is this grammar regular?
2. Give the parse tree for each of the following phrases:
   - $baabaab$
   - $bBABb$
   - $baSb$
3. Give the leftmost and rightmost derivations for $baabaab$.

Ex. 3. Write a context-free grammar that generates all strings of as and bs (in any order) such that there are more as than bs. Test your grammar on the input $baaba$ by giving a derivation.

Ex. 4. Write a context-sensitive grammar that generates all strings of as, bs and cs (in any order) such that there are as many of each. Give a derivation of $cacbab$ using your grammar.

Session 4: Pushdown automata and parsing

For theory reminders, refer to chapter(s) 7 and 8.

Ex. 1. Design a pushdown automaton that accepts the language made of all words of the form ww^R where w is any given word on the alphabet $\Sigma = \{a, b\}$ and w^R is the mirror image of w.

Parsers

Consider the context-free grammar shown in Figure 1 where <system goal> is the start symbol (see last rule) and $ denotes the end of the input:

Ex. 2. Give the parse tree for the following input:

```
begin
  ID := ID - INTLIT + ID ;
end
```

Ex. 3. Simulate a top-down parser on the following input:

(1)	<program>	$\rightarrow$	begin <statement list> end
(2)	<statement list>	$\rightarrow$	<statement> <statement tail>
(3)	<statement tail>	$\rightarrow$	<statement> <statement tail>
(4)	<statement tail>	$\rightarrow$	ε
(5)	<statement>	$\rightarrow$	ID := <expression> ;
(6)	<statement>	$\rightarrow$	read (<id list>) ;
(7)	<statement>	$\rightarrow$	write (<expr list>) ;
(8)	<id list>	$\rightarrow$	ID <id tail>
(9)	<id tail>	$\rightarrow$	, ID <id tail>
(10)	<id tail>	$\rightarrow$	ε
(11)	<expr list>	$\rightarrow$	<expression> <expr tail>
(12)	<expr tail>	$\rightarrow$	, <expression> <expr tail>
(13)	<expr tail>	$\rightarrow$	ε
(14)	<expression>	$\rightarrow$	<primary> <primary tail>
(15)	<primary tail>	$\rightarrow$	<add op> <primary> <primary tail>
(16)	<primary tail>	$\rightarrow$	ε
(17)	<primary>	$\rightarrow$	(<expression>)
(18)	<primary>	$\rightarrow$	ID
(19)	<primary>	$\rightarrow$	INTLIT
(20)	<add op>	$\rightarrow$	$+$
(21)	<add op>	$\rightarrow$	$-$
(22)	<system goal>	$\rightarrow$	<program> $

Figure 1: Grammar used in Session 4.

```
begin
  A :=  BB - 314 + A ;
end
```

Ex. 4. Simulate a bottom-up parser on the same input.

Session 5: First sets, Follow sets and LL(1) parsing

For theory reminders, refer to chapter(s) 8 and 9.

$First^k$ sets construction algorithm

begin
- **foreach** $a \in T$ **do** $\text{First}^k(a) \leftarrow \{a\}$
- **foreach** $A \in V$ **do** $\text{First}^k(A) \leftarrow \emptyset$
- **repeat**
  - **foreach** $A \in V$ **do**
    - $\text{First}^k(A) \leftarrow \text{First}^k(A) \cup \{x \in T^* \mid A \rightarrow Y_1 Y_2 \dots Y_n \wedge x \in \text{First}^k(Y_1) \oplus^k \text{First}^k(Y_2) \oplus^k \cdots \oplus^k \text{First}^k(Y_n)\}$
- **until** *stability*

$Follow^k$ sets construction algorithm

begin
- **foreach** $A \in V$ **do** $\text{Follow}^k(A) \leftarrow \emptyset$;
- **repeat**
  - **if** $B \rightarrow \alpha A \beta \in P$ **then**
    - $\text{Follow}^k(A) \leftarrow \text{Follow}^k(A) \cup \{\text{First}^k(\beta) \oplus^k \text{Follow}^k(B)\}$;
- **until** *stability*;

(1)	<program>	$\rightarrow$	begin <statement list> end
(2)	<statement list>	$\rightarrow$	<statement> <statement tail>
(3)	<statement tail>	$\rightarrow$	<statement> <statement tail>
(4)	<statement tail>	$\rightarrow$	ε
(5)	<statement>	$\rightarrow$	ID := <expression> ;
(6)	<statement>	$\rightarrow$	read (<id list>) ;
(7)	<statement>	$\rightarrow$	write (<expr list>) ;
(8)	<id list>	$\rightarrow$	ID <id tail>
(9)	<id tail>	$\rightarrow$	, ID <id tail>
(10)	<id tail>	$\rightarrow$	ε
(11)	<expr list>	$\rightarrow$	<expression> <expr tail>
(12)	<expr tail>	$\rightarrow$	, <expression> <expr tail>
(13)	<expr tail>	$\rightarrow$	ε
(14)	<expression>	$\rightarrow$	<primary> <primary tail>
(15)	<primary tail>	$\rightarrow$	<add op> <primary> <primary tail>
(16)	<primary tail>	$\rightarrow$	ε
(17)	<primary>	$\rightarrow$	(<expression>)
(18)	<primary>	$\rightarrow$	ID
(19)	<primary>	$\rightarrow$	INTLIT
(20)	<add op>	$\rightarrow$	$+$
(21)	<add op>	$\rightarrow$	$-$
(22)	<system goal>	$\rightarrow$	<program> $

Figure 2: Grammar for exercises 1 and 4 (Session 5).

Action table construction algorithm

```
begin
    M ← × ;
    foreach A → α do
        foreach a ∈ First¹(α) do
            M[A, a] ← M[A, a] ∪ Produce(A → α) ;
        if ε ∈ First¹(α) then
            foreach a ∈ Follow¹(A) do
                M[A, a] ← M[A, a] ∪ Produce(A → α) ;
    foreach a ∈ T do M[a, a] ← Match ;
    M[$, ε] ← Accept ;
```

Ex. 1. With regards to the grammar given by Figure 2:

1. Give the $\text{First}^1(A)$ and the $\text{Follow}^1(A)$ sets for each $A \in V$.
2. Give the First^2(<expression>) and the Follow^2(<expression>) sets.

Ex. 2. Which of these grammars are LL(1)?

1. $$\begin{cases} S & \rightarrow & ABBA \\ A & \rightarrow & a \\ & & \varepsilon \\ B & \rightarrow & b \\ & & \varepsilon \end{cases}$$

2. $\begin{cases} S \rightarrow aSe \\ \quad\;\; B \\ B \rightarrow bBe \\ \quad\;\; C \\ C \rightarrow cCe \\ \quad\;\; d \end{cases}$

3. $\begin{cases} S \rightarrow ABc \\ A \rightarrow a \\ \quad\;\; \varepsilon \\ B \rightarrow b \\ \quad\;\; \varepsilon \end{cases}$

4. $\begin{cases} S \rightarrow Ab \\ A \rightarrow a \\ \quad\;\; B \\ \quad\;\; \varepsilon \\ B \rightarrow b \\ \quad\;\; \varepsilon \end{cases}$

Ex. 3. Give the action table for the following grammar:

(1)	<S>	→	<expr> $
(2)	<expr>	→	− <expr>
(3)	<expr>	→	(<expr>)
(4)	<expr>	→	<var> <expr-tail>
(5)	<expr-tail>	→	− <expr>
(6)	<expr-tail>	→	ε
(7)	<var>	→	ID <var-tail>
(8)	<var-tail>	→	(<expr>)
(9)	<var-tail>	→	ε

Ex. 4. Program a recursive descent parser (in C, C++, . . .) for rules (14) through (21) of the grammar given by Figure 2.

Session 6: Grammars revisited

For theory reminders, refer to chapter(s) 6.

Grammar `RemoveUnproductiveSymbols`(**Grammar** $G = \langle V, T, P, S\rangle$) **begin**
- $V_0 \leftarrow \emptyset$;
- $i \leftarrow 0$;
- **repeat**
  - $i \leftarrow i + 1$;
  - $V_i \leftarrow \{A \mid A \rightarrow \alpha \in P \wedge \alpha \in (V_{i-1} \cup T)^*\} \cup V_{i-1}$;
- **until** $V_i = V_{i-1}$;
- $V' \leftarrow V_i$;
- $P' \leftarrow$ set of rules of P that do not contain variables in $V \setminus V'$;
- `return`($G' = \langle V', T, P', S\rangle$) ;

Grammar `RemoveInaccessibleSymbols`(**Grammar** $G = \langle V, T, P, S\rangle$) **begin**
- $V_0 \leftarrow \{S\}$; $i \leftarrow 0$;
- **repeat**
  - $i \leftarrow i + 1$;
  - $V_i \leftarrow \{X \mid \exists\, A \rightarrow \alpha X \beta$ in $P \wedge A \in V_{i-1}\} \cup V_{i-1}$;
- **until** $V_i = V_{i-1}$;
- $V' \leftarrow V_i \cap V$; $T' \leftarrow V_i \cap T$;
- $P' \leftarrow$ set of rules of P that only contain variables from V_i ;
- `return`($G' = \langle V', T', P', S\rangle$) ;

Grammar `RemoveUselessSymbols(`**Grammar** $G = \langle V, T, P, S\rangle$`)` **begin**
- **Grammar** $G_1 \leftarrow$ `RemoveUnproductiveSymbols(`G`)` ;
- **Grammar** $G_2 \leftarrow$ `RemoveInaccessibleSymbols(`G_1`)` ;
- `return(`G_2`)` ;

`LeftFactor(`**Grammar** $G = \langle V, T, P, S\rangle$`)` **begin**
- **while** *G has at least two rules with the same left-hand side and a common prefix* **do**
  - Let $E = \{A \to \alpha\beta, \ldots, A \to \alpha\zeta\}$ be such a set of rules ;
  - Let $\mathcal{V}$ be a new variable;
  - $V = V \cup \mathcal{V}$;
  - $P = P \setminus E$;
  - $P = P \cup \{A \to \alpha\mathcal{V}, \mathcal{V} \to \beta, \ldots, \mathcal{V} \to \zeta\}$;

`RemoveLeftRecursion(`**Grammar** $G = \langle V, T, P, S\rangle$`)` **begin**
- **while** *G contains a left recursive variable A* **do**
  - Let $E = \{A \to A\alpha, A \to \beta, \ldots, A \to \zeta\}$ be the set of rules that have A as left-hand side ;
  - Let $\mathcal{U}$ and $\mathcal{V}$ be two new variables ;
  - $V = V \cup \{\mathcal{U}, \mathcal{V}\}$;
  - $P = P \setminus E$;
  - $P = P \cup \{A \to \mathcal{U}\mathcal{V}, \mathcal{U} \to \beta, \ldots, \mathcal{U} \to \zeta, \mathcal{V} \to \alpha\mathcal{V}, \mathcal{V} \to \varepsilon\}$;

Ex. 1. Remove the useless symbols in the following grammars:

$$(1)\left\{\begin{array}{lll} S & \to & a \mid A \\ A & \to & AB \\ B & \to & b \end{array}\right.$$

$$(2)\left\{\begin{array}{lll} S & \to & A \\ & & B \\ A & \to & aB \\ & & bS \\ & & b \\ B & \to & AB \\ & & Ba \\ C & \to & AS \\ & & b \end{array}\right.$$

Ex. 2. Consider the following grammar:

$$\left\{\begin{array}{lll} E & \to & E\ op\ E \\ & & ID[E] \\ & & ID \\ op & \to & * \\ & & / \\ & & + \\ & & - \\ & & -> \end{array}\right.$$

- Show that the above grammar is ambiguous.
- The priorities of the various operators are as follows: $\{[], ->\} > \{*, /\} > \{+, -\}$.

  Modify the grammar in order for it to take operator precedence into account as well as left associativity.

Ex. 3. Left-factor the following production rules:

<stmt> $\to$ **if** <expr> **then** <stmt-list> **end if**
<stmt> $\to$ **if** <expr> **then** <stmt-list> **else** <stmt-list> **end if**

Ex. 4. Apply the left recursion removal algorithm to the following grammar:

$$\begin{cases} E & \rightarrow & E+T \\ & & T \\ T & \rightarrow & T*P \\ & & P \\ P & \rightarrow & ID \end{cases}$$

Ex. 5. **(Exam-level question)** Transform the following grammar into an LL(1) grammar:

$$\begin{cases} S & \rightarrow & aE \mid bF \\ E & \rightarrow & bE \mid \epsilon \\ F & \rightarrow & aF \mid aG \mid aHD \\ G & \rightarrow & Gc \mid d \\ H & \rightarrow & Ca \\ C & \rightarrow & Hb \\ D & \rightarrow & ab \end{cases}$$

Session 7: LR(0) and LR(k) parsing

For theory reminders, refer to chapter(s) 10.

LR(0) parsing

```
Closure(I) begin
    repeat
        I' ← I ;
        foreach item [A → α • Bβ] ∈ I, B → γ ∈ G' do
            I ← I ∪ [B → •γ] ;
    until I' = I;
    return(I) ;
```

```
Transition(I,X) begin
    return(Closure({[A → αX • β] | [A → α • Xβ] ∈ I})) ;
```

```
Items(G') begin
    C ←Closure({[S' → •S]}) ;
    repeat
        C' ← C;
        foreach I ∈ C, X ∈ T' ∪ V' do
            C' ← C'∪ Transition(I,X) ;
    until C' = C;
```

LR(0) action table construction algorithm:

```
foreach state s of the CFSM do
    if s contains A → α • aβ then Action[s] ← Action[s] ∪ Shift ;
    else if s contains A → α• that is the i^th rule then Action[s] ← Action[s] ∪ Reduce_i ;
    else if s contains S' → S$• then Action[s] ← Action[s] ∪ Accept ;
```

Ex. 1. Consider the following grammar:

(0)	$S' \rightarrow S\$$	(5)	$C \rightarrow Fg$
(1)	$S \rightarrow aCd$	(6)	$C \rightarrow CF$
(2)	$S \rightarrow bD$	(7)	$F \rightarrow z$
(3)	$S \rightarrow Cf$	(8)	$D \rightarrow y$
(4)	$C \rightarrow eD$		

Give the corresponding LR(0) CFSM and its action table.

Ex. 2. Simulate the parser you built during the previous exercise on the following string : "`aeyzzd`".

Extra Ex. Consider the following grammars:

1.

(0)	$S' \rightarrow S\$$	(3)	$L \rightarrow *R$
(1)	$S \rightarrow L = R$	(4)	$L \rightarrow \mathbf{id}$
(2)	$S \rightarrow R$	(5)	$R \rightarrow L$

2.

(0)	$S' \rightarrow S\$$	(1)	$S \rightarrow \varepsilon$
(2)	$S \rightarrow SaSb$		

Give the grammars' corresponding LR(0) CFSMs and their action tables.

LR(k) parsing

```
Closure(I) begin
    repeat
        I' ← I ;
        foreach item [A → α • Bβ, σ] ∈ I, B → γ ∈ G' do
            foreach u ∈ First^k(βσ) do  I ← I ∪ [B → •γ, u];
    until I' = I;
    return(I) ;
```

```
Transition(I,X) begin
    return(Closure({[A → αX • β, u] | [A → α • Xβ, u] ∈ I})) ;
```

LR(k) action table construction algorithm:

```
foreach state s of the CFSM do
    if s contains [A → α • aβ, u] then
        foreach u ∈ First^k(aβu) do
            Action[s, u] ← Action[s, u] ∪ Shift ;
    else if s contains [A → α•, u ], that is the i^th rule then
        Action[s,u ] ← Action[s, u] ∪ Reduce_i ;
    else if s contains [S' → S$•, ε] then
        Action[s, ·] ← Action[s, ·] ∪ Accept ;
```

The other algorithms are identical to the LR(0) case.

Ex. 3. Give the LR(1) CFSM for the following grammar and its action table:

(1)	$S' \rightarrow S\$$	(5)	$B \rightarrow cC$
(2)	$S \rightarrow A$	(6)	$B \rightarrow cCe$
(3)	$A \rightarrow bB$	(7)	$C \rightarrow dAf$
(4)	$A \rightarrow a$		

Is the grammar LR(0) ? Explain.

Ex. 4. Give the LR(1) CFSM for the following grammar and its action table:

(1) $S' \rightarrow S\$$
(2) $S \rightarrow SaSb$
(3) $S \rightarrow c$
(4) $S \rightarrow \varepsilon$

Simulate the parser on the following input: "`abacb`".

Session 8: SLR(1) and LALR(1) parsing

For theory reminders, refer to chapter(s) 10.

Reminder: SLR(1) action table construction algorithm

With the LR(0) items in hand, we build the action table as follows ($a \in \Sigma$):

foreach *state s of the CFSM* **do**
  if *s contains* $A \to \alpha \bullet a\beta$ **then**
    $\text{Action}[s, a] \leftarrow \text{Action}[s, a] \cup \text{Shift}$;
  else if *s contains* $A \to \alpha\bullet$ *that is the* i^{th} *rule* **then**
    foreach $a \in \mathit{Follow}^1(A)$ **do**
      $\text{Action}[s, a] \leftarrow \text{Action}[s, a] \cup \text{Reduce}_i$;
  else if *s contains* $S' \to S\$\bullet$ **then**
    $\text{Action}[s] \leftarrow \text{Action}[s] \cup \text{Accept}$;

Ex. 1. Build the SLR(1) parser for the following grammar:

(1) $S' \to S\$$
(2) $S \to A$
(3) $A \to bB$
(4) $A \to a$
(5) $B \to cC$
(6) $B \to cCe$
(7) $C \to dAf$

Ex. 2. Build the LALR(1) parser for the same grammar.

Session 9: `lex/flex` scanner generator

For theory reminders, refer to chapter(s) 3.

A *filter* is a program that reads text on the standard input and prints it modified on standard output. For example, a filter that replaces all `a`s with `b`s and that receives `abracadabra` on input would output `bbrbcbdbbrb`.

Specification format

A `lex` specification is made of three parts separated by lines with `%%`:

- **Part 1**: regular expression definitions and arbitrary C code (between `%{` and `%}`) to be inserted at the start of the scanner program
  - The regular expression definitions are used as "macros" in part 2.
  - The C code usually comprises header includes and declarations of variables, functions, etc.
- **Part 2**: translation rules of the following shape: `Regex   {Action}`
  - `Regex` is an *extended regular expression* (ERE)
  - `Action` is a *C code snippet* that will be executed each time a *token* matching `Regex` is encountered.
  - The regular expressions defined in Part 1 can be used by putting their names in curly braces `{ }`.
- **Part 3**: Arbitrary C code to be inserted at the end of the generated program.
  - For example: `main()` if the *scanner* isn't used in conjunction with `yacc` or `bison`.

Variables and special actions

When writing *actions*, some special variables and macros can be accessed:

- `yyleng` contains the *length* of the recognized token
- `yytext` is a `char*` (C string) that points to the actual string that was matched by the regular expression.
- `yylval` is a special variable that will be used to pass information (attributes) to `yacc`
- `ECHO` is a macro (defined by `lex` itself) that is equivalent to `printf("%s",yytext)` and can be used when some recognized strings are to be output as is.

Compiling

To obtain the scanner executable :

1. Generate the scanner code with `lex myspec.l` (creates `lex.yy.c`)
2. Compile the code generated by `lex` into an object file: `gcc -c lex.yy.c` (creates `lex.yy.o`)
3. Compile other `.c` files as needed into object files
4. *Link* all object files together with the `libl` (for `lex`) or `libfl` (for `flex`) library:
   `gcc -o myscanner file1.o ... fileN.o lex.yy.o -lfl`

Note that the `-lfl` flag (meaning "link against `libfl`") is put *after* the file names.

Example

```
%{
/* Arbitrary C code to be prepended to generated code */
#include <stdlib.h>
%}
number [0-9]
letter [a-zA-Z]
identifier {letter}({number}|{letter})*
%%
{identifier}  { printf("ID %s of length %d\n", yytext, yyleng); }
({number})+ { printf("Integer : "); ECHO; }
%%
/* Arbitrary C code to be appended to generated code */
int main() {
    yylex();
}
```

Exercises

Ex. 1. Write a filter that outputs the number of alphanumeric characters, alphanumeric words and of lines in the input file.

Ex. 2. Write a filter that outputs its input file with line numbers in front of every line.

Ex. 3. Write a filter that only outputs comments in the input file. Such comments are comprised within curly braces `{ }`.

Ex. 4. Write a filter that transforms the input text by replacing the word "compiler" with "ewww" if the line starts with an "a", with "???" if it starts with a "b" and by "profit!!!" if it starts with a "c".

Ex. 5. Write a *lexical analysis function* that recognises the following *tokens*:

- Decimal numbers in scientific notation
- C variable identifiers
- Relational operators ($<$, $>$, $==$, etc.)
- The `if`, `then` and `else` keywords

The point of this function is then to be used by `yacc`. As such, each action should *return* an integer value representing the kind of token that was found and should store the *value* in the `yylval` variable. For example, if an integer is found, we would return a value representing that fact, and we would store the actual integer value in `yylval` before returning.

Extra Ex. Write a program using `lex/flex` that *pretty prints* C code. Your program should take a C file as input and should then print the following to the terminal:

- Keywords in bold face (`while`, `for`, ...)
- String literals (delimited by `"`) in green
- Integer literals in blue
- Comments (delimited by `/*` and `*/` for block comments, or by `//` and a newline for line comments) in black over white (reverse colouring).
- Correctly indented code

To this end, you may use the `textcolor(attr, fg, bg)` function available in an archive on the exercises' Web site.

- `attr` allows text to be made bold face (valeur `BRIGHT`), shown in reverse video mode (`REVERSE`) or in normal mode (`RESET`).
- `fg` et `bg` are used to specify the colors to be used for foreground and background (values `GREEN`, `BLUE`, `WHITE`, `BLACK`...)

Session 10: `yacc/bison` parser generator

For theory reminders, refer to chapter(s) 10.

You have received a `lex` and a `yacc` specification (they can also be downloaded off the Web site).

1. Informally describe the accepted language of the compiler we'd generate from the specifications.
2. Adjust the specification so it only accepts polynomials of a single variable. We input a polynomial per line, but there can only be one variable used on each line.
3. Add the necessary code to show the first derivative of a polynomial. For example, if `2x^3+2x^2+5` was given on input, we would output :
   `First derivative:  6x^2+4x`
4. Add a way to recognize polynomial products and adjust the derivative calculation. For example, if `(3x^2+6x)*(9x+4)` is given on input, we would output:
   `First derivative:  ((3x^2+6x)*(9))+((6x+6)*(9x+4))`
5. Add a way to evaluate a polynomial and its first derivative for a given value. The user should be able to input the variable value, followed by a semicolon, followed by the polynomial (all this on the same line). For example :

```
2 ; (3x^2+6x)*(9x+4)
First derivative : ((3x^2+6x)*(9))+((6x+6)*(9x+4))
p(2) = 528, p'(2) = 612
```

Appendix : `lex` specification

```
/* ************************************************** */
/* *  Introduction to Language Theory and Compilation  * */
/* *                                                   * */
/* *           Session 10: yacc/bison                  * */
/* *                                                   * */
/* *             lex specification                     * */
/* ************************************************** */


number [0-9]
letter [a-zA-Z]
integer {number}+
var {letter}+
%{
#include "derive.tab.h"
%}
%%

{integer} {return INTEGER ;}
{var}     {return VAR ;}
" "       {}
.         {return yytext[0] ;}
"\n"      {return yytext[0] ;}

%%
```

Appendix : `yacc` specification

```
/* *************************************************** */
/* *  Introduction to Language Theory and Compilation  * */
/* *                                                   * */
/* *            Session 10: yacc/bison                 * */
/* *                                                   * */
/* *              yacc specification                   * */
/* *************************************************** */

%{
#include <stdio.h>
%}

%token INTEGER
%token VAR

%left '+' '-'
%left '*' '/'

%%

input : line input      {}
      | line            {}
      ;

line : polynomial '\n'  {printf("OK\n") ;}
     ;

polynomial : polynomial '+' terme {}
           | polynomial '-' terme {}
           | terme {}
           ;

terme     : '-' terme                 {}
          | VAR '^' INTEGER           {}
          | INTEGER VAR '^' INTEGER {}
          | VAR                       {}
          | INTEGER VAR               {}
          | INTEGER                   {}
          ;
%%

int main (void)
{
  yyparse() ;
}


int yyerror(char * s)
{
printf("yyerror: I encountered an error: %s.\n\n",s) ;
}
```

Session 11: Code generation

For theory reminders, refer to chapter(s) 11 and 12.

P-code

Ex. 1. Write a P-code program that computes and outputs the value of:

$$(3 + x) * (9 - y)$$

where x is a value read on input and y is the value stored at address 0.

Ex. 2. Write a program that outputs all odd values in the interval $[7, 31]$. In order to do this, you'll need the `dpl i` instruction that duplicates the integer value on top of the stack.

Ex. 3. Write the code that:

- Allocates memory for two static variables we'll call a and b
- Initializes a and b with values read on input
- Adds 5 to a
- Divides b by 2
- If $a > b$, output a, else output b

Make sure that the memory slots allocated for a and b are consistent after every step above.

Attribute grammars

Ex. 4. Rewrite the following grammar in order to account for operator precedence and associativity:

<E>	→	<E> <op> <E> | (<E>) | int
<op>	→	+ | − | ∗ | /

Associate the rules and attributes necessary to compute the value of an expression E. Finally, remove left recursion from the grammar.

Ex. 5. The following is a set of rules that defines an **if** of an imperative programming language:

<if>	→	**if** <cond> **then** <code> <if-tail>
<if-tail>	→	**else** <code> **endif**
<if-tail>	→	**endif**

Give the P-code instructions you'd have to generate to translate this kind of construct in a compiler. You can assume <cond> and <code> are already decorated to generate the correct code.