Pourquoi LISP / Scheme ?

Double intérêt

Pour l’analyse, la modélisation, la programmation

Pour l’enseignement

Double intérêt

Pour l'analyse, la modélisation, la programmation

- **« Pour manipuler des concepts abstraits de haut niveau, Lisp, Prolog et OCaml (et leurs cousins respectifs) ont tous les trois des avantages. »**
- **« Lisp est un langage de prédilection pour la plupart des domaines de l'Intelligence Artificielle, où la programmation exploratoire joue un rôle prépondérant. »**

Double intérêt

Pour l'analyse, la modélisation, la programmation

- **Lisp is very simple and powerful.**
- **Lisp is an excellent prototyping tool.**
- **Lisp supports symbolic programming well.**
- **Typical AI areas for computing with symbols: computer algebra, theorem proving, planning systems, diagnosis, rewrite systems, knowledge representation and reasoning, logic languages, machine translation, expert systems, and more.**
-

Double intérêt

Pour l'analyse, la modélisation, la programmation

Many famous AI applications are in Lisp:

*** Macsyma - the first computer algebra**
*** ACL2 - a widely used theorem prover**
*** DART - the logistics planner used during the first Gulf war by the US military.**
*** SPIKE - the planning and scheduling application for the Hubble Space Telescope.**
*** CYC - one of the largest software systems written. Representation and reasoning in the domain of human common sense knowledge.**
*** METAL - one of the first commercially used natural language translation systems.**
*** American Express' Authorizer's Assistant, which checks credit card transactions.**
*** GNOME game Aislerot uses Scheme ...**

Double intérêt

Pour l’enseignement

Langage d'initiation à la programmation :

- **simplicité**
- **facilité de syntaxe**
- **faible nombre des mots-clés**
- **interactivité**
- **...**

Double intérêt

Pour l’enseignement

Universités (Montpellier, Paris 6, ...)
Ecoles de MINES (Saint-Étienne)
INSA
SUPINFO, ...

MIT
Berkelay
Houston
Stanford
Yale, ...

Lisp, The Quantum Programmer's Choice
https://www.youtube.com/watch?v=svmPz5oxMlI

Un peu d’histoire

Précurseur : lambda-calcul

Histoire

Pour l'analyse, la modélisation, la programmation

- **Dans les années 1930 : λ-calcul introduit par Alonzo Church**
  - **définir et caractériser les fonctions et les fonctions récursives**
  - **important pour la théorie de la calculabilité**
  - **langage théorique et aussi métalangage pour la démonstration formelle assistée par ordinateur**

- **λ-expression, exemple en LISP : (λ (x) (-x))**

Histoire

Pour l'analyse, la modélisation, la programmation

- **En 1958, John McCarthy (MIT) a proposé LISP (list processing)**

- **programmation fonctionnelle**
  - **syntaxe simple : équivalente aux listes en notation préfixée**
  - **manipuler le code source en tant que structure de données (liste)**
  - **typage dynamique des données**
  - **analyse syntaxique simple**
  - **méta-programmation et réflexivité : des programmes qui créent d'autres programmes ou modifient le programme courant**
  - **utilisation en IA**

Éléments

Quelques types de base (non exhaustif)

Type	Pour vérifier le type de x
• **Nombre**	**(number? x)**
Entier	**(integer? x)**
Rationnel	**(rational? x)**
Réel	**(real? x)**
Complexe	**(complex? x)**
• **Caractère**	**(char? x)**
• **String**	**(string? x)**
• **Booléen**	**(boolean? x)**

Ces objets sont « auto-évalués » par l'interprète

Éléments

Quelques types de base (non exhaustif)

Ces objets sont « auto-évalués » par l'interprète

- **3**
  3 (réponse)
- **2/3**
  2/3 (réponse)
- **#f**
  #f (réponse)

Éléments

Quelques types de base

Type	Pour vérifier le type de x
• **Paire**	**(pair? x)**
• **Liste**	**(list? x)**

On va les détailler (constructions, accès…)

• **Symbole**	**(symbol? x)**

Éléments
Quelques types de base

- **Symbole** **(symbol? x)**
- **Faire des symboles :**

  1) interdire l'évaluation :

  - `'a2`
    `a2`

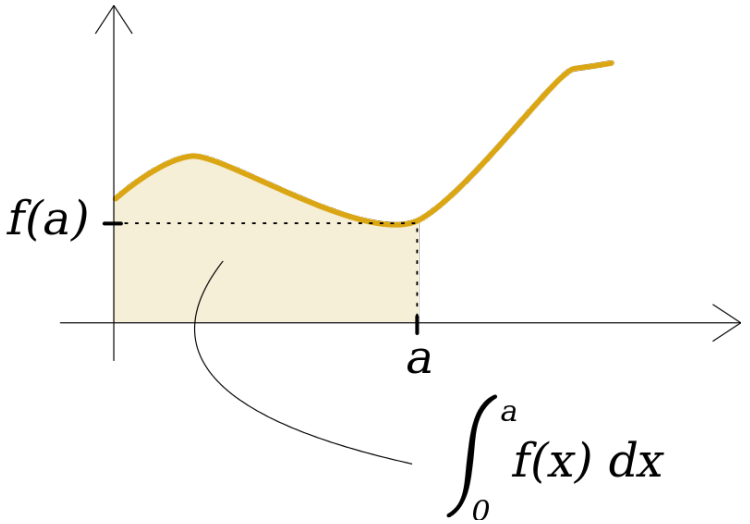
  2) définir

  - `(define z 5)`
  - `z`
    `5`

Éléments

Fonctions

- **Opération (instruction) + Opérandes**

- **Exemples**
- **2 + 4**
- 

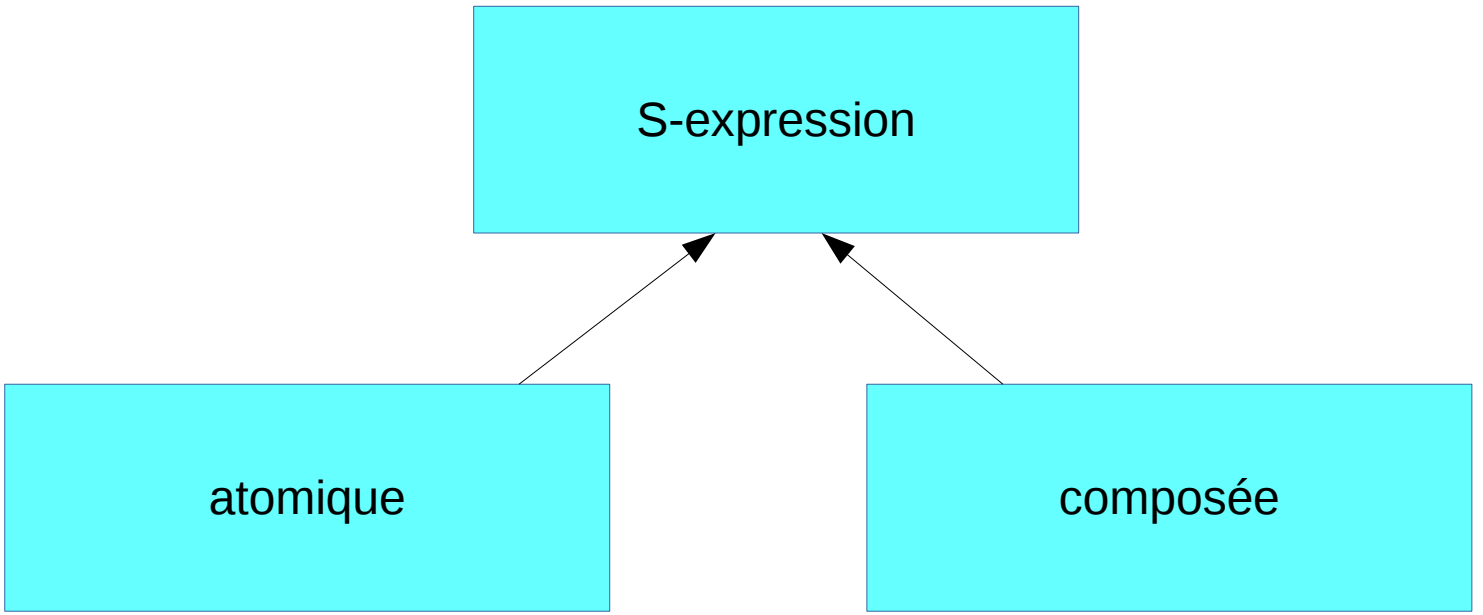
Un format possible :

(+ 2 4)

Éléments

Expressions

- **En scheme, tous sont des expressions symboliques : S-expressions**



Expressions

Notations, arbre syntaxique, évaluation

-
-

Expressions

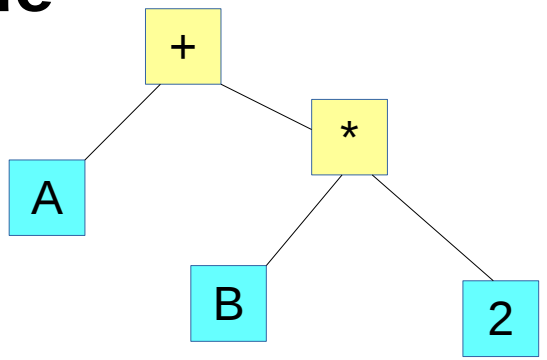
Notations, arbre syntaxique, évaluation

Pour simplifier, on va parler des expressions algébriques de base (opérations binaires)

- **Forme symétrique**

  A + B * 2

- **Arbre syntaxique**



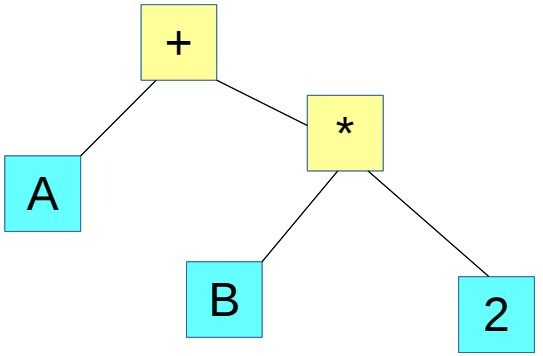
- **Parcours symétrique**

  A + B * 2

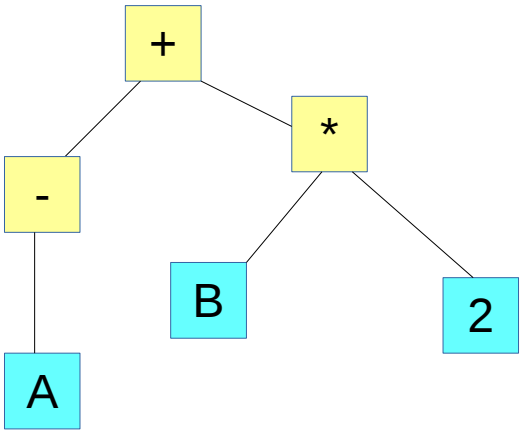
Expressions

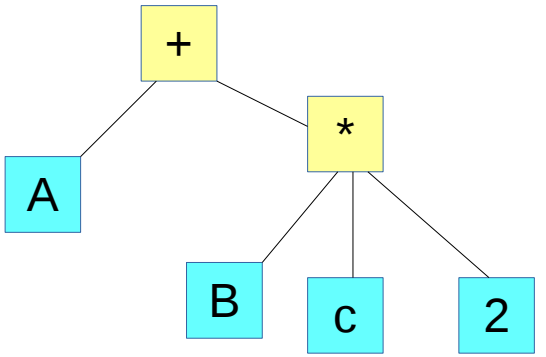
Notations, arbre syntaxique, évaluation

- **Arbre syntaxique**



- **Remarque : on peut envisager des arbres n-aires (n = 1, 2, 3, ...)**



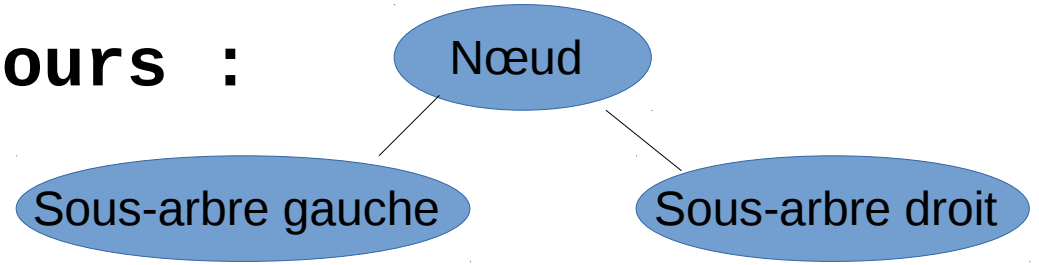


Expressions

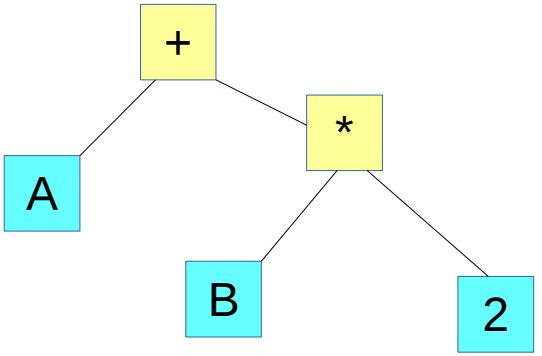
Arbre syntaxique, parcours

Pour simplifier, on va parler des expressions algébriques de base (opérations binaires)

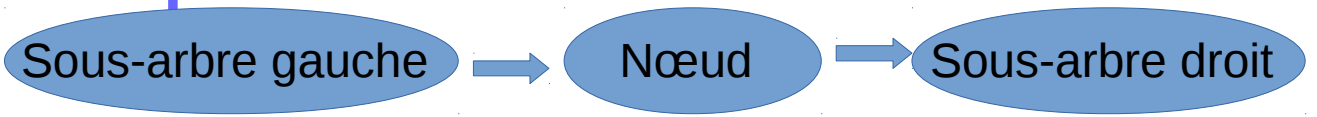
- **Pour parler des parcours :**



- **Arbre syntaxique**



- **Parcours symétrique**



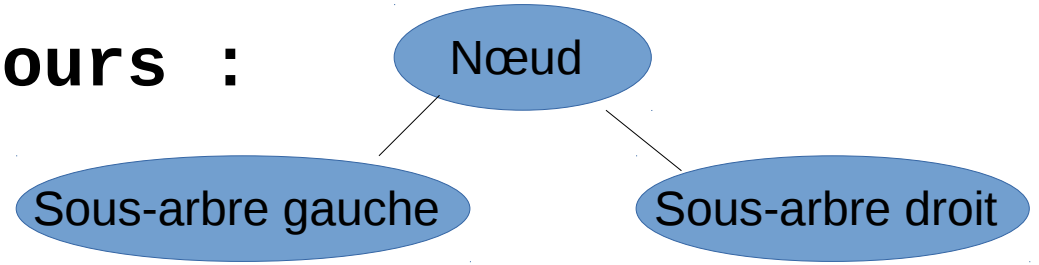
(A + (B * 2)) éliminer certaines parenthèses

Expressions

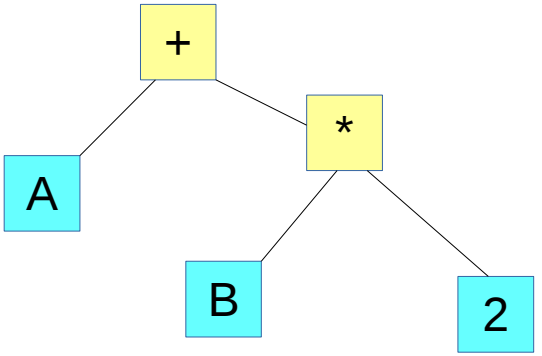
Arbre syntaxique, parcours

Pour simplifier, on va parler des expressions algébriques de base (opérations binaires)

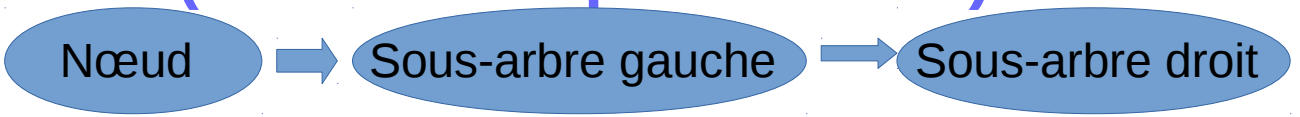
- **Pour parler des parcours :**



- **Arbre syntaxique**



- **Parcours préfixé (notation polonaise)**



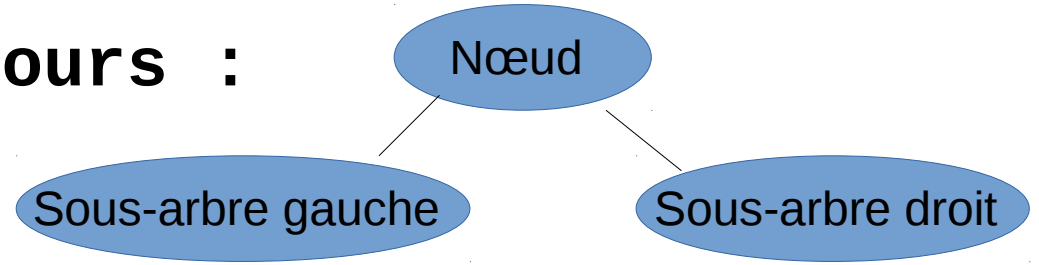
(+ A (* B 2)) parenthèses inutiles

Expressions

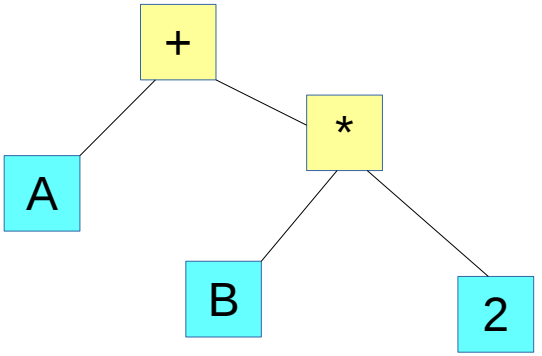
Arbre syntaxique, parcours

Pour simplifier, on va parler des expressions algébriques de base (opérations binaires)

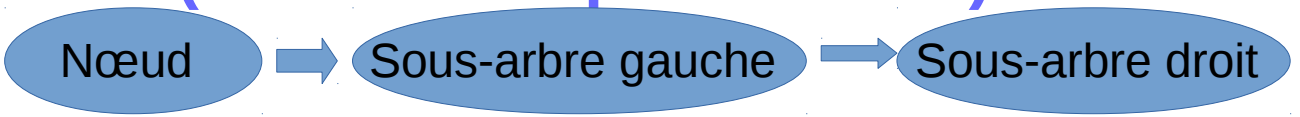
- **Pour parler des parcours :**



- **Arbre syntaxique**



- **Parcours préfixé (notation polonaise)**



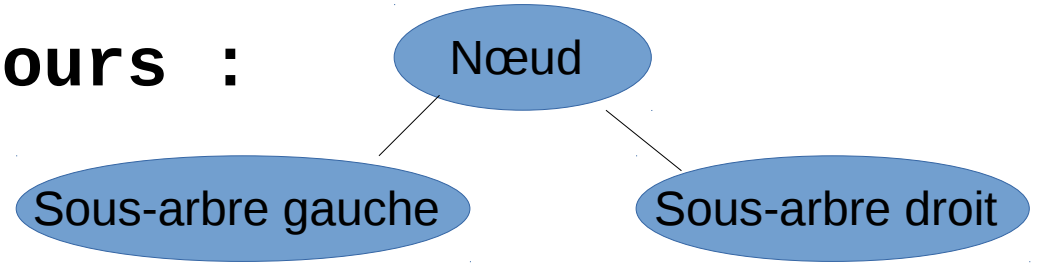
+ A * B 2 parenthèses inutiles

Expressions

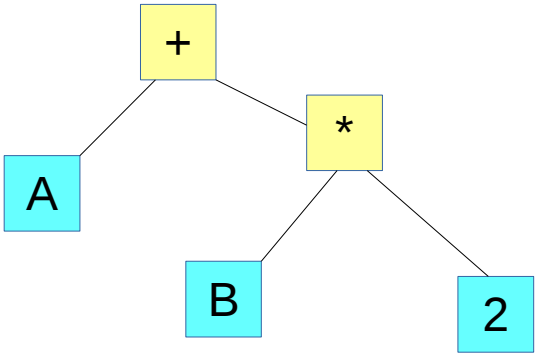
Arbre syntaxique, parcours

Pour simplifier, on va parler des expressions algébriques de base (opérations binaires)

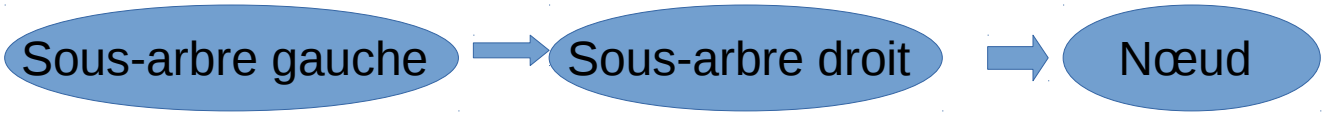
- **Pour parler des parcours :**



- **Arbre syntaxique**



- **Parcours postfixé (notation polonaise)**

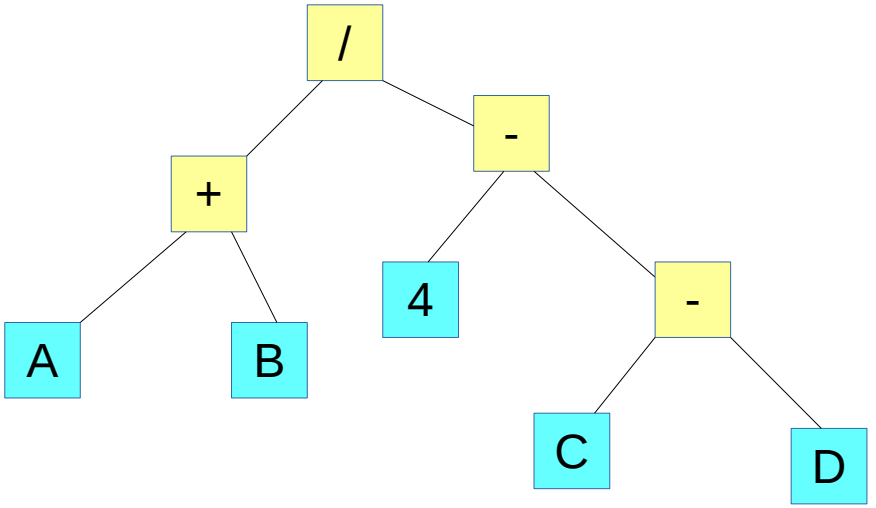


A B 2 * + parenthèses inutiles

Expressions

Arbre syntaxique, parcours

Exercice

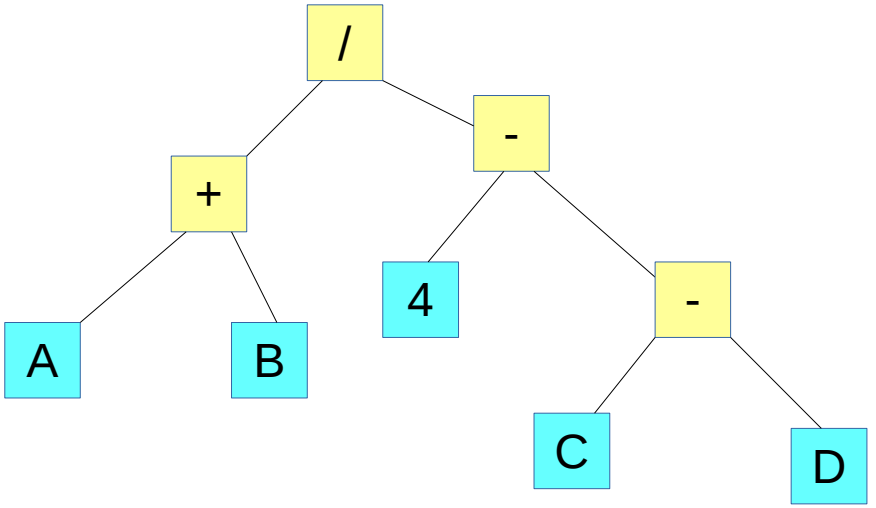


- **Parcours symétrique**
- **Parcours préfixé**
- **Parcours postfixé**

Expressions

Arbre syntaxique, parcours

Exercice



- **Parcours symétrique**

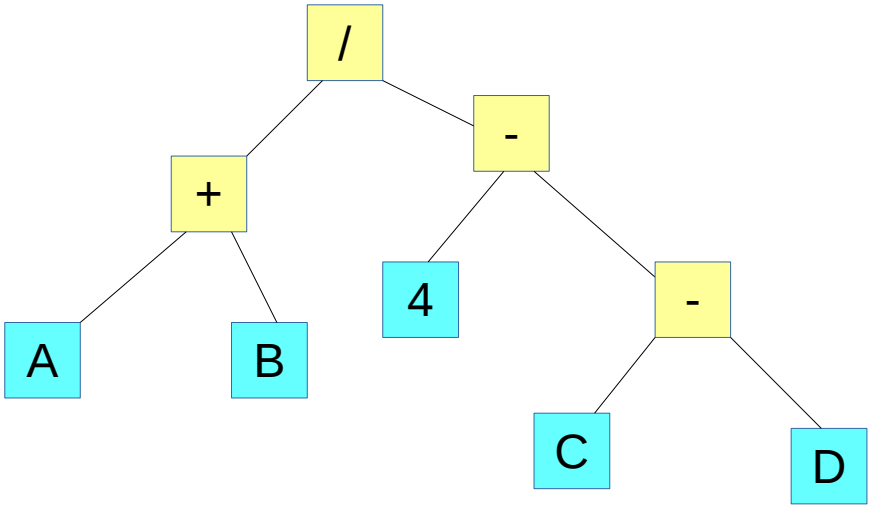
A + B / 4 – C – D hmmm

→ (A + B) / (4 – (C – D))

Expressions

Arbre syntaxique, parcours

Exercice



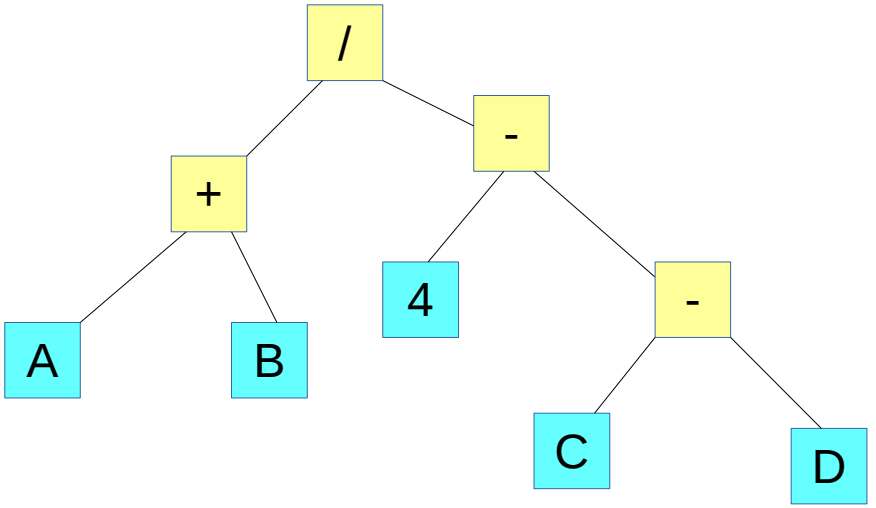
- **Parcours préfixé (DGD)**

/ + A B – 4 – C D

Expressions

Arbre syntaxique, parcours

Exercice



- **Parcours postfixé (AGD)**

A B + 4 C D - - /

Evaluation d'expressions

Evaluation d'une expression postfixée à l'aide d'une machine à pile

Fonctionnement.

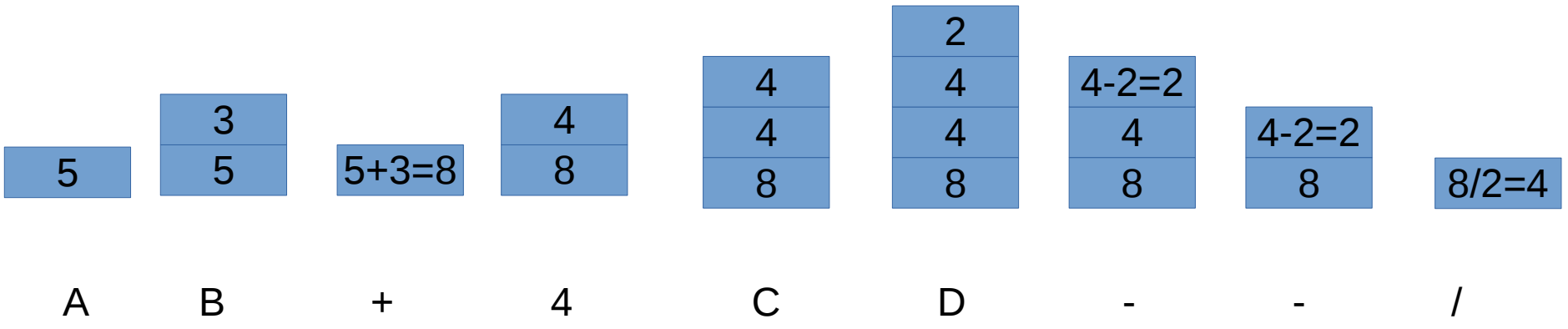
Lire les éléments de l'expression (de gauche à droite) :

si opérande : l'empiler
si opérateur: dépiler deux éléments et exécuter l'opération → empiler le résultat

Evaluation d'expressions

Evaluation d'une expression postfixée à l'aide d'une machine à pile

Exemple : **`A B + 4 C D - - /  A=5, B=3, C=4, D=2`**



(S-)Expressions

Les expressions non atomiques sont des expressions (application des fonctions) préfixée.

Exemples :

```
(+ 6 11)
(sin 3.5)
(* 2 pi (/ D 2))

(/ (+ A B) (- 4 (- C D)))
```

Evaluation d'expressions

Expressions atomiques (valeur et symboles) :

- on retourne leur valeur

5

→ 5

Expressions non atomique :

- on évalue les arguments (l'ordre n'est pas défini)
- on applique la fonction sur les arguments déjà évalués (naturellement, récursivement)

(* 2 pi)

→2

→3.14

→**6.28**

Evaluation d'expressions

Exercice

Indiquer (par un diagramme) comment l'expression est évaluée

(/ (+ 5 7) 3)

Evaluation d'expressions

Exercice

Indiquer (par un diagramme) comment l'expression est évaluée

(/ (+ 5 7) 3)

Fonctions

Prédéfinies

Nombreuses fonctions existent

+ - / * or and not quote ...

Définir une fonction

La base

(lambda (arguments) expression-instruction)

Exemple :

(lambda (d) (* 2 * pi (/ d 2)))

Attention : lambda n'attribue pas de nom à la fonction

Fonctions

Utiliser une fonction lambda

Comment appliquer une fonction ?

(fonction argument1 arguement2 …)

Lambda correspond à une fonction :

((lambda (d) (* 2 (* pi (/ d 2)))) 6)

→ 18,84

((lambda (a b) (or (not a) (not b))) #t #f)

→ #t

((lambda (x y) (* x y) (- 17 5) (+ 3 2))

→ 60

Fonctions

Une fonction qui protège son argument
(pas d'évaluation)

(quote argument)

'argument

Exemples :

(quote (+ 2 3))
→ (+ 2 3)

'toto
→ toto

Fonction define

Une fonction qui associe un nom à quelque chose

(define nom argument)

Exemples :

```
(define a 3)
a
→3
(define b (+ 2 3))
b
→5
```

Fonction define

Donner un nom à une fonction

```
(define nand (lambda (a b) (or (not a) (not b))))
```

Appel :

```
(nand #t #f)
```

→ #t

Ecriture simplifiée :

```
(define (nand a b) (or (not a) (not b)))
```

Interpréteur Scheme

drRacket

Racket, the Programming Language

Mature | Jet Fueled | Extensible | Robust | Polished | Vibrant Community

```
#lang racket/gui

(define my-language 'English)

(define translations
  #hash([English . "Hello world"]
        [French . "Bonjour le monde"]
        [German . "Hallo Welt"]
        [Greek . "Γειά σου, κόσμε"]
        [Portuguese . "Olá mundo"]
        [Spanish . "Hola mundo"]
        [Thai . "สวัสดีชาวโลก"]))

(define my-hello-world
  (hash-ref translations my-language
            "hello world"))

(message-box "" my-hello-world)
```

Racket, the Language-Oriented Programming Language

Little Macros | General Purpose | Big Macros | Easy DSLs | IDE Support | Any Syntax

Interpréteur Scheme

drRacket