Syntaxe et Sémantique

Laboratoire LSD12

Xavier Devroey, Michaël Marcozzi et Alain Solheid

{xde, mmr, aso}@info.fundp.ac.be

FUNDP 2011 - 2012

Plan

- Introduction
- Méthodologie
- Support
- Compilateur
  - Analyseur lexical
  - Analyseur syntaxique
  - Analyseur sémantique
  - Générateur de code
- Aspects pratiques

Références

Les principales références sont celles du cours théorique :

- Aho, Sethi, Ullman, **Compilateurs : principes, techniques et outils**, InterEditions/Dunod. BUMP #I 412/037, ISBN 2100051261.
- R. Wilhelm, D. Maurer,**Les compilateurs : théorie, construction, génération**, Masson, 1994. BUMP #I 412/030, ISBN 2225846154.

Introduction

Objectifs:

- Travail de groupe
- Mise en pratique des concepts vus au cours
- Lecture et compréhension d'un spécification formelle
- Construction d'un compilateur correct
- Utilisation d'un outil de feed-back automatique
  - http://concours.info.fundp.ac.be/automate/

Méthodologie

- Travail à partir d'une spécification formelle
- Tests en black-box (`concours`)
- Evaluation
  - Résultats aux échéances
  - Choix techniques et architecturaux
  - Fonctionnement du groupe
  - Etc.
- Au travers des soumissions à `concours`, du rapport et de l'interrogation orale

Support

- Webcampus
- La page web du TP : : http://info.fundp.ac.be/~xde/pmwiki/index.php/Teaching/INFOB314
- L'automate : http://concours.info.fundp.ac.be/automate/
- La GPMachine : http://info.fundp.ac.be/~gpm/
- Les autres groupes (attention au plagiat)
- Google (attention au plagiat !!!)
- Les assistants
  - Xavier Devroey
  - Michaël Marcozzi
  - Alain Solheid

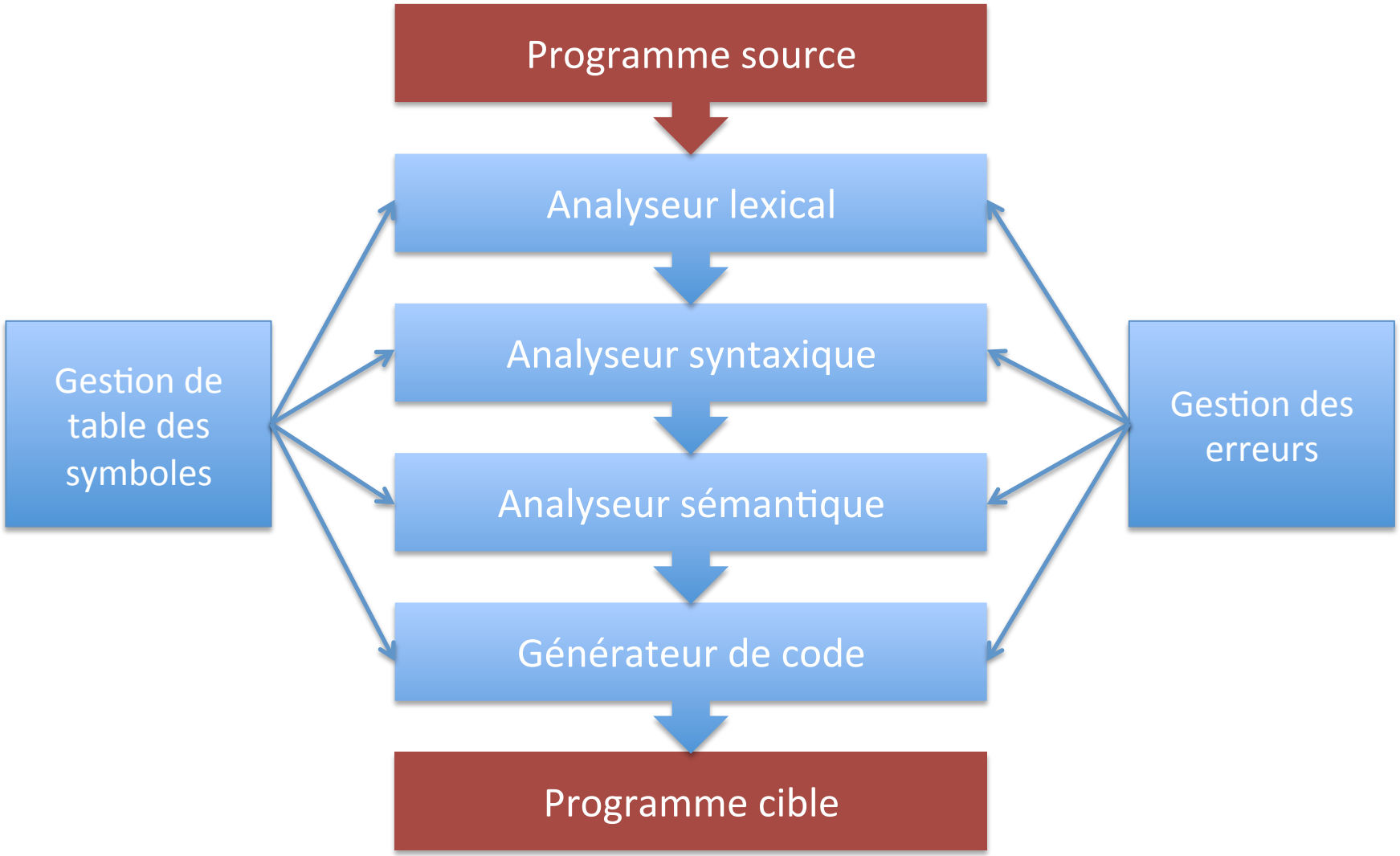
Outils

- Utilisation obligatoire de :
  - flex
  - bison
  - gcc
- Utilisation conseillée de :
  - Linux
  - Outils de versioning de code (SVN, CVS, Git, …)
  - Outils de débuging (Valgrind)
  - …

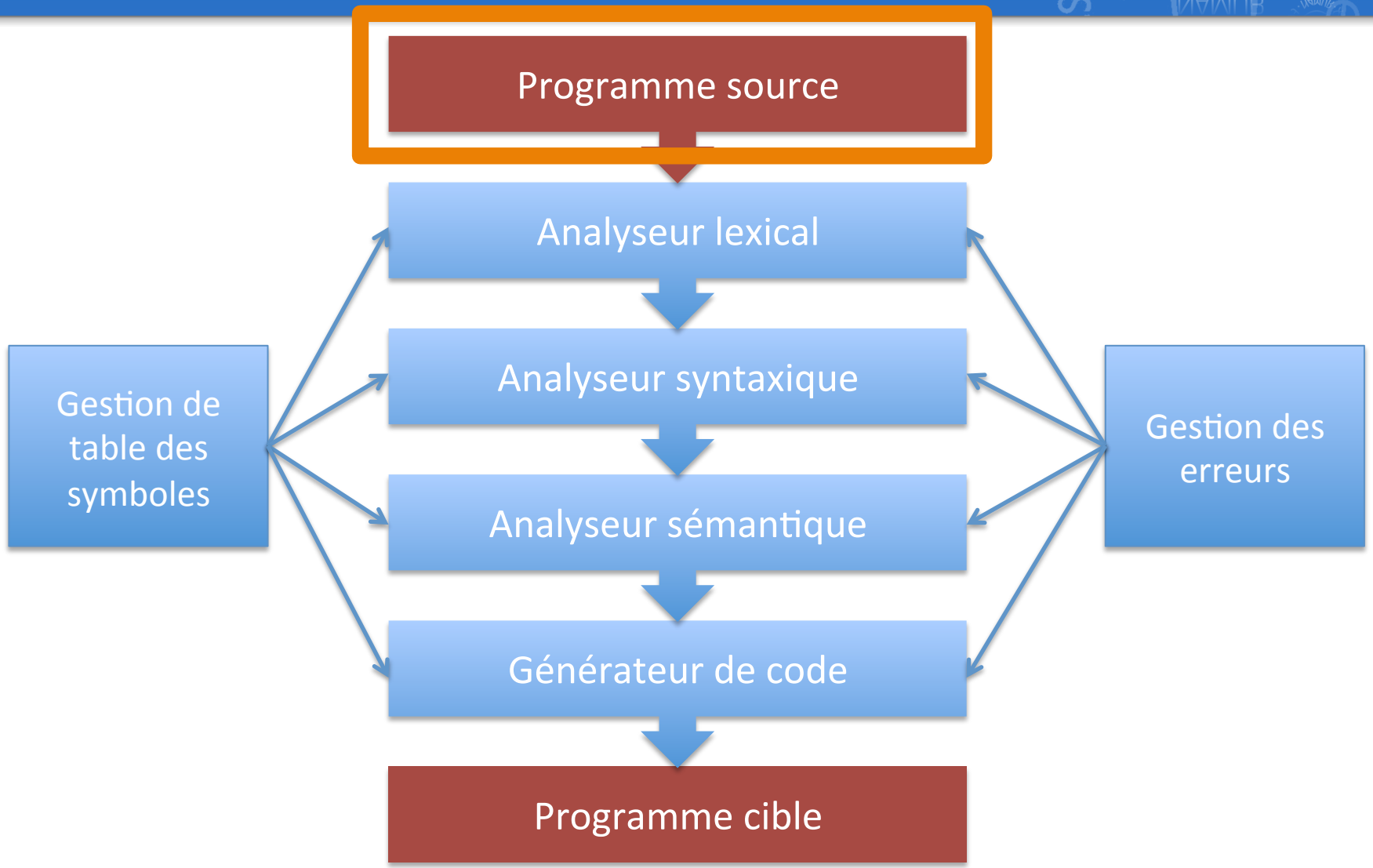
Compilateur

- Traduit
  - un programme source (ex : Nabuchodonosor.java)
  - écrit dans un langage source (ex : Java)
- en
  - Un programme cible (ex : Nabuchodonosor.class)
  - écrit dans un langage cible (ex : bytecode Java)
- Attention : un compilateur ne fait que traduire du code, il ne l'exécute pas !
- Exemples : javac, gcc, gpc, etc.

Compilateur



Compilateur

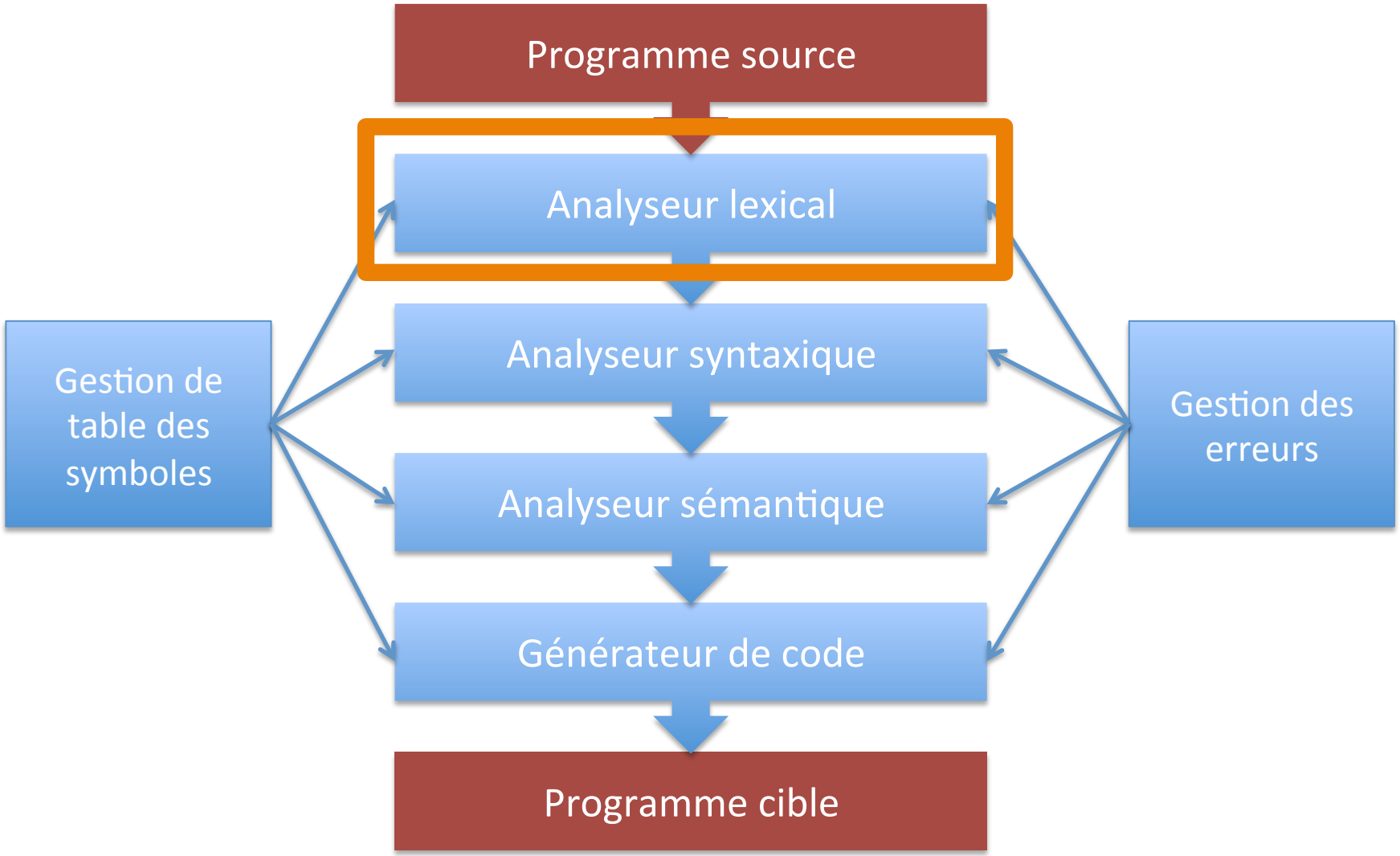


Programme source

- Ecrit en LSD12
  - Langage impératif classique
  - Manipule des entiers et des booléens
  - Décrit dans le document : “LSD12 : Et ça repart”
  - Exemple:

```
program a;
    function main():void;
        var
            x int;
        begin
            x := 2;
            x := x + 3 * 2;
        end;
end;
```

Compilateur

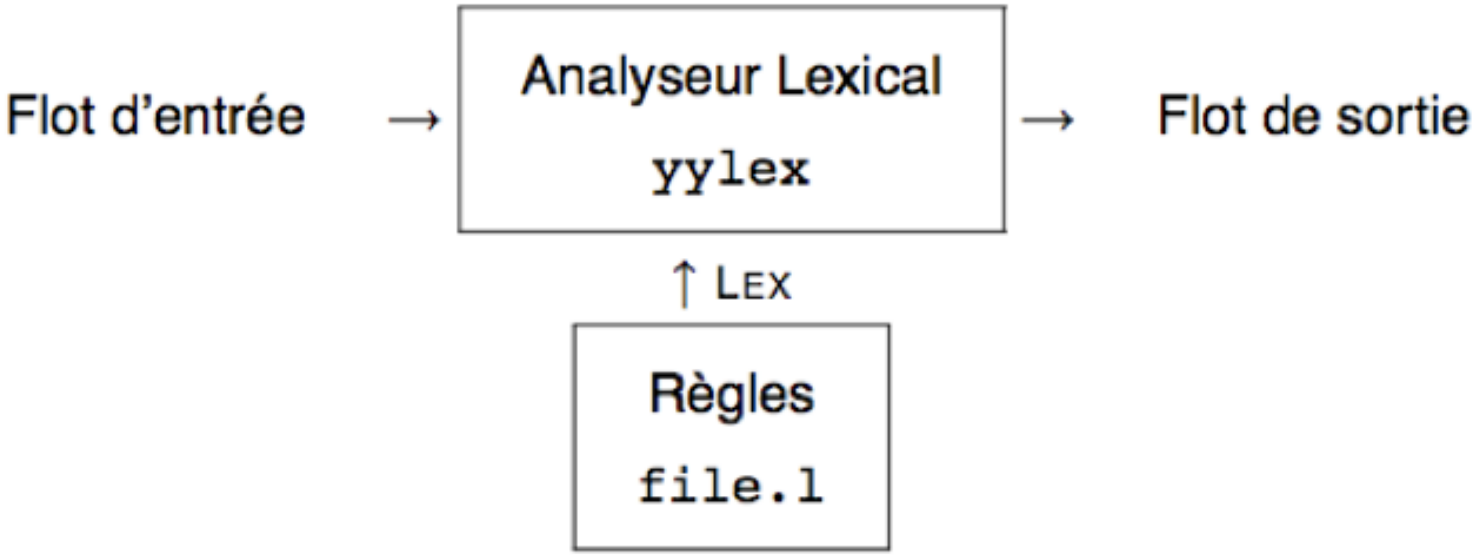


Analyseur lexical

- But : vérifier que tous les mots et symboles appartiennent bien au langage
- Exemples de mots (ou lexèmes) :
  - Mots réservés : `program`, `function`, `void`, `;`, `(`, `)`, *etc*.
  - Identificateurs : `a`, `main`, `toto`, *etc*.
  - Constantes : `0`, `1`, `2`, `-21349`, *etc*.
- Peuvent être décrits au moyen d'expressions régulières

Analyseur lexical

- Construit automatiquement par Lex/Flex sur base d'un ensemble de règles



Structure d'un fichier Lex

- Un fichier Lex se compose de 3 parties :
  - Déclarations C
  - Noms de sécification
  - Déclaration de conditions

  %%

  - Règles sous la forme : <exp reg> <action>

  %%

  - Toute fonction C utile
- Cf. demo.l pour un exemple fonctionnel

Expression régulière

Symbole	Signification
“ ”	Une chaîne de caractères entourée de double-quotes représente la chaîne elle-même.
[ ]	Une chaîne de caractères entre crochets représente un de ses éléments. Exemple : [abc] représente “a” ou “b” ou “c”
.	tout caractère sauf \n
|	Opérateur d’alternance. Exemple : “a”|”b”|”c” = [abc]
*	Opérateur de répétition : zéro ou plusieurs fois
+	Opérateur de répétition : au moins une fois
?	Opérateur d’occurrence : zéro ou une fois
/	Condition de reconnaissance
^ et $	Début de ligne et fin de ligne
{}	Opérateur de répétition bornée. Exemple : a{3} = “aaa”

Gestion des conflits

- En cas de conflit, Lex choisit toujours la règle qui produit le plus long lexème. Exemple :

```
prog action1
program action2
```

  La deuxième règle sera choisie.

- Si plusieurs règles donnent des lexèmes de mêmes longueurs, Lex choisit la première. Exemple :

```
prog action1
[a-z]+ action2
```

  La première règle sera choisie.

- Si aucune règle ne correspond au flot d'entrée, Lex choisit sa règle par défaut implicite :

```
. |\n {ECHO}
```

  Qui recopie le flot d'entrée sur le flot de sortie

Actions

- Action = bloc d'instructions exécuté lorsque la chaîne de caractères lue correspond à la chaîne spécifiée avant l'action
- Peut utiliser certaines variables prédéfinies :
  - **`yytext`** : chaîne reconnue correspondant à l'expression régulière
  - **`yylen`** : longueur de `yytext`. Par exemple une règle qui permet de compter nombre de mots et le nombre de caractères dans les mots peut s'écrire :

```
[a-zA-Z]+ { words++;chars=chars+yylen; }
```

  - **`yyin`** : fichier d'entrée (FILE * en C)
  - **`yyout`** : fichier de sortie (FILE * en C)

Actions

Peut aussi contenir certaines macros prédéfinies :

- | qui indique que l'action à exécuter est la même que celle de la règle suivante
- ECHO

  `[a-z]+ {printf("%s",yytext);}` équivaut à `[a-z]+ {ECHO;}`
- REJECT en C qui permet d'envisager la "deuxième meilleure" règle

```
%%
frob {special();REJECT;}
[ ^ \t\n]+ {++word_count;}
```

compte le nombre de mots du flot d'entrée et exécute special à chaque fois que la chaîne `frob` apparaît

Définitions

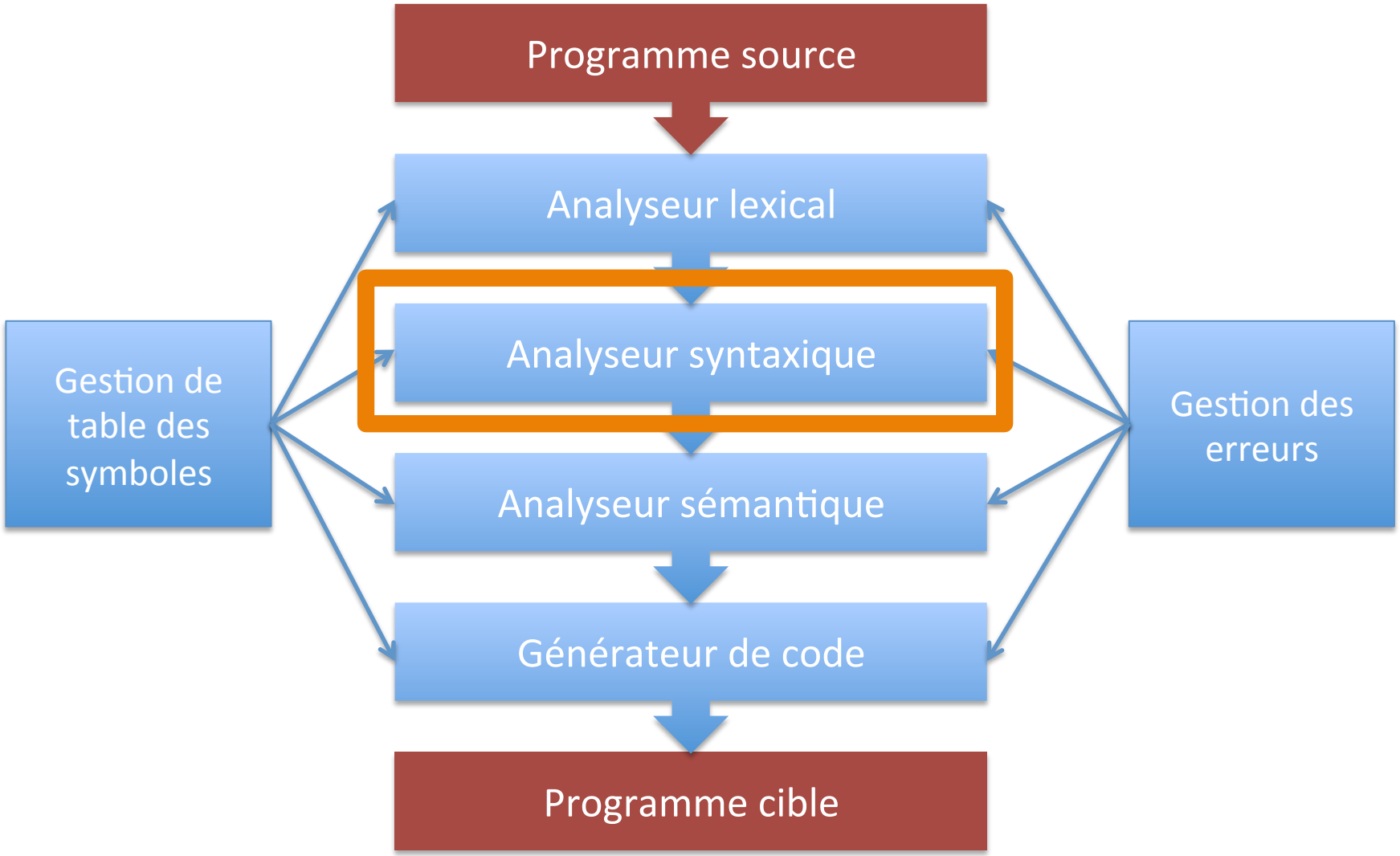
La spécification des définitions

- Contient les déclarations “C” (entre `%{` et `%}`)
- Permet d’associer un nom à une spécification
- Permet de déclarer des “conditions de déclenchement”

Exemple :

```
DIGIT [0-9]
...
{DIGIT}+"."{DIGIT}* ;
```

Compilateur



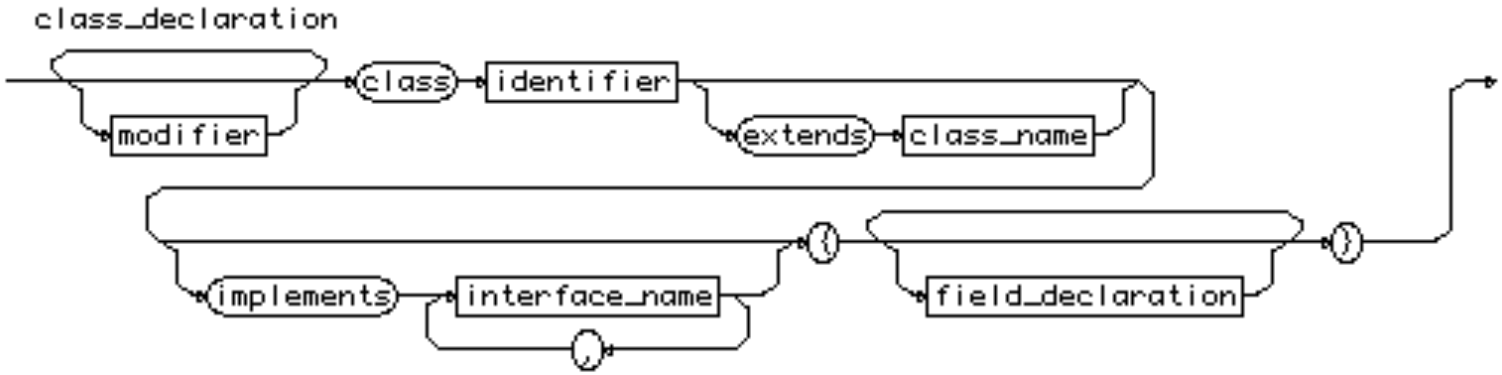
Analyseur syntaxique

- But : vérifier que les enchainements de mots et symboles forment bien des phrases appartenant au langage
- Exemples de phrases :
  - `program pgm; function fct():void; var begin write 3;end; end;`
  - `program x; function main():void; var a int; function fct0():void; var begin write 3;end; begin fct0 () ;end; end;`
- Peuvent être décrits au moyen d'une grammaire non contextuelle

Grammaire non contextuelle

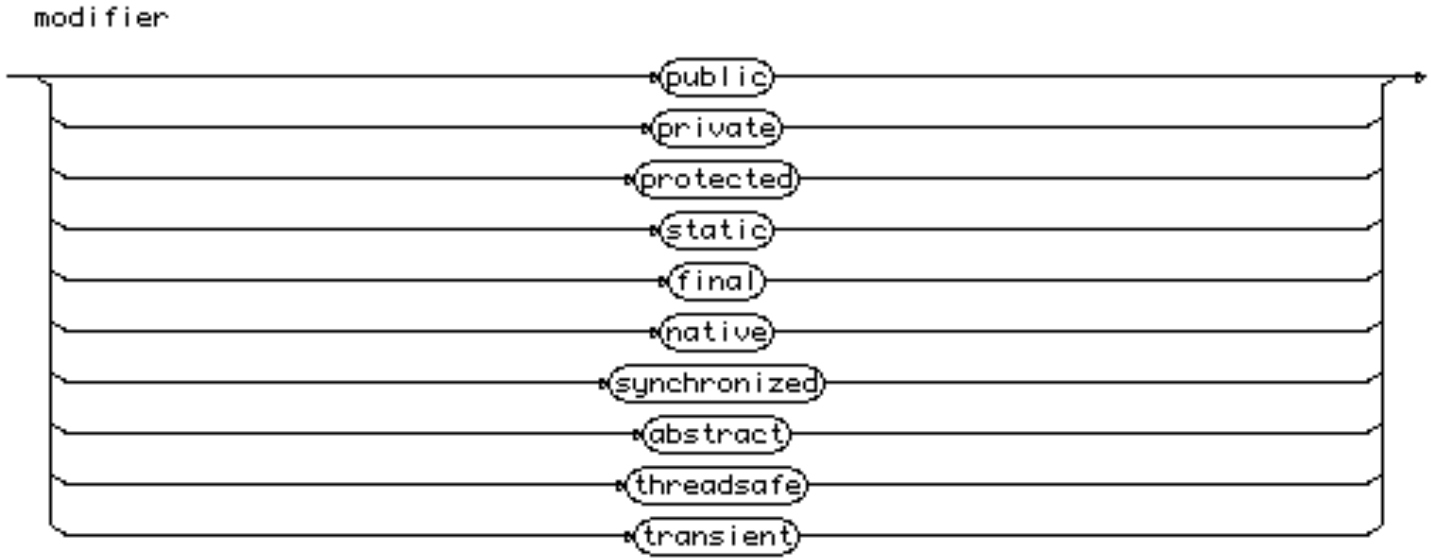
- But : représenter de manière finie un langage à priori infini
- Exemple : Java (http://cui.unige.ch/java/JAVAF/)

```
class_declaration
        ::=
        { modifier } "class" identifier
        [ "extends" class name ]
        [ "implements" interface name { "," interface name } ]
        "{" { field declaration } "}"
```



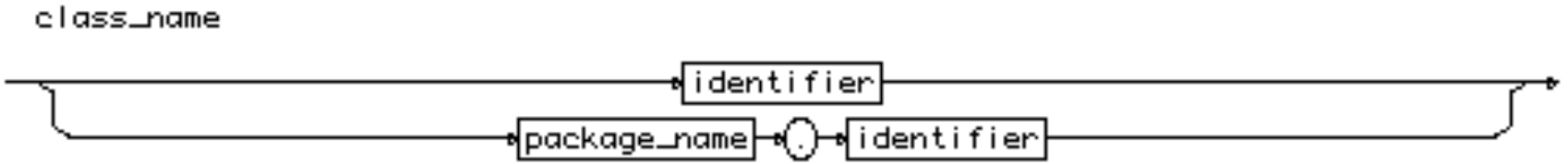
Grammaire non contextuelle


```
modifier
        ::=
        "public"
          | "private"
          | "protected"
          | "static"
          | "final"
          | "native"
          | "synchronized"
          | "abstract"
          | "threadsafe"
          | "transient"
```

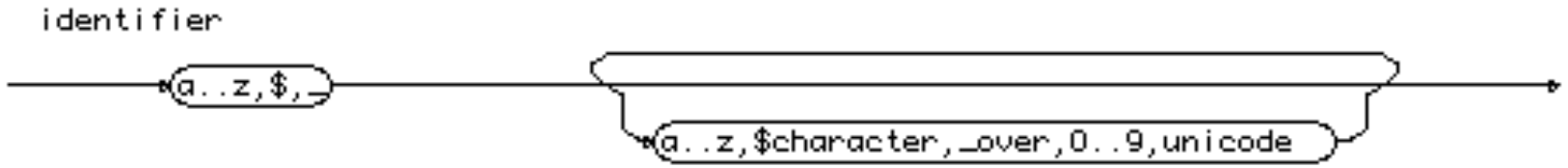


Grammaire non contextuelle


```
class_name
        ::=
        identifier
        | ( package name "." identifier )
```




```
identifier
        ::= "a..z,$,_" { "a..z,$,_,0..9,unicode character over 00C0" }
```



Grammaire non contextuelle

- Composée de 4 éléments :
  - Ensemble (fini) des symboles terminaux Vt
    - Eg. Vt = {“class”, “public”, “private”, …}
  - Ensemble (fini) des non-terminaux Vn
    - Eg. Vn = {modifier, identifier, class_name, …}
  - Ensemble (fini) des productions P
    - Eg. http://cui.unige.ch/java/JAVAF/
  - Symbole de départ S (non-terminal)
    - Eg. compilation_unit

Grammaire non contextuelle

- Utilisation d'un grammaire :
  - Partir du symbole de départ (ou axiome) S ;
  - Répéter : employer une règle ;
  - jusqu'à ce qu'il n'y ait plus que des symboles terminaux dans le texte.
- Dérivation = suite de pas de cet algorithme

$$S \Rightarrow \Psi 1 \Rightarrow \Psi 2 \Rightarrow \Psi 3 \Rightarrow w$$

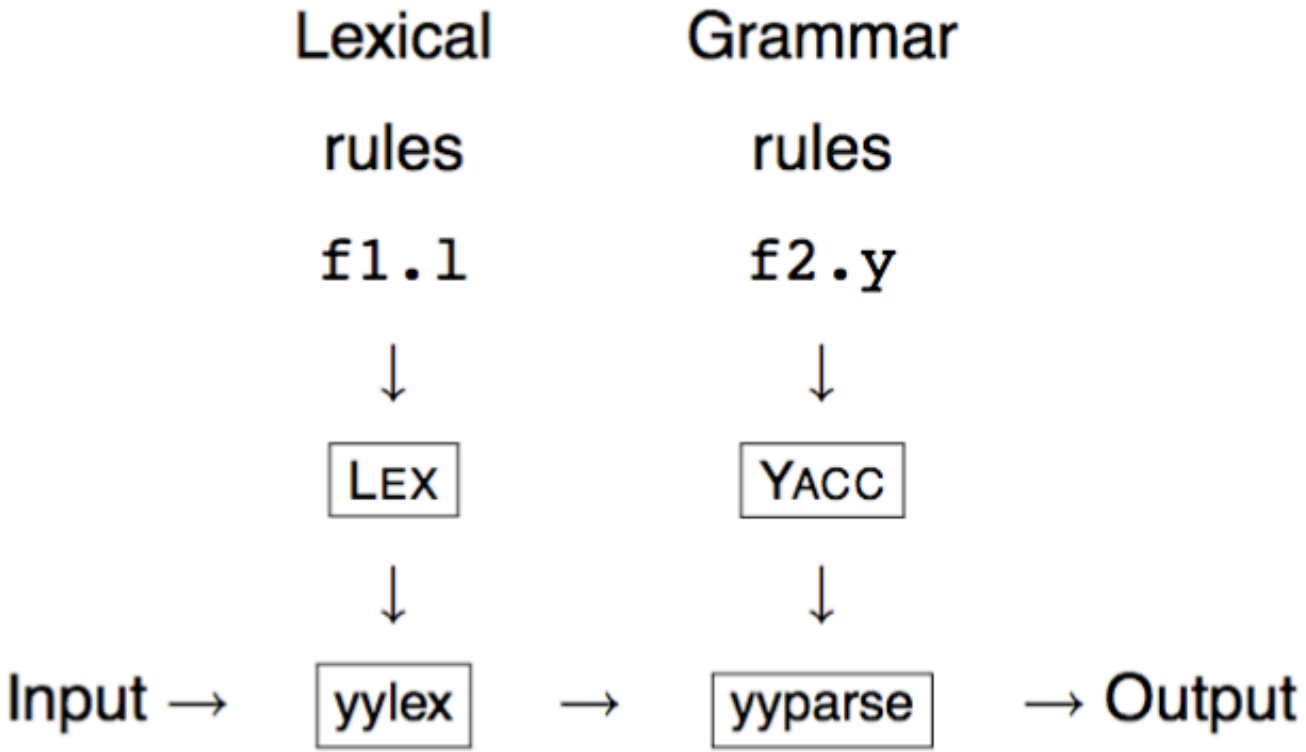
Si l'algo permet de passer de ϕ à ψ en zéro, une ou plusieurs étapes, on dit que ϕ produit ψ, noté $\phi \Rightarrow^* \psi$

Grammaire non contextuelle

- Chaque chaîne que l'on peut produire grâce à l'algorithme est une **phrase** de G.
- L'ensemble des phrases est le langage de G, noté **L(G)**.
- Les étapes intermédiaires sont des **proto-phrases** de G.

Analyseur syntaxique

- Construit automatiquement par yacc/bison sur base d'un ensemble de règles



Structure d'un fichier Yacc

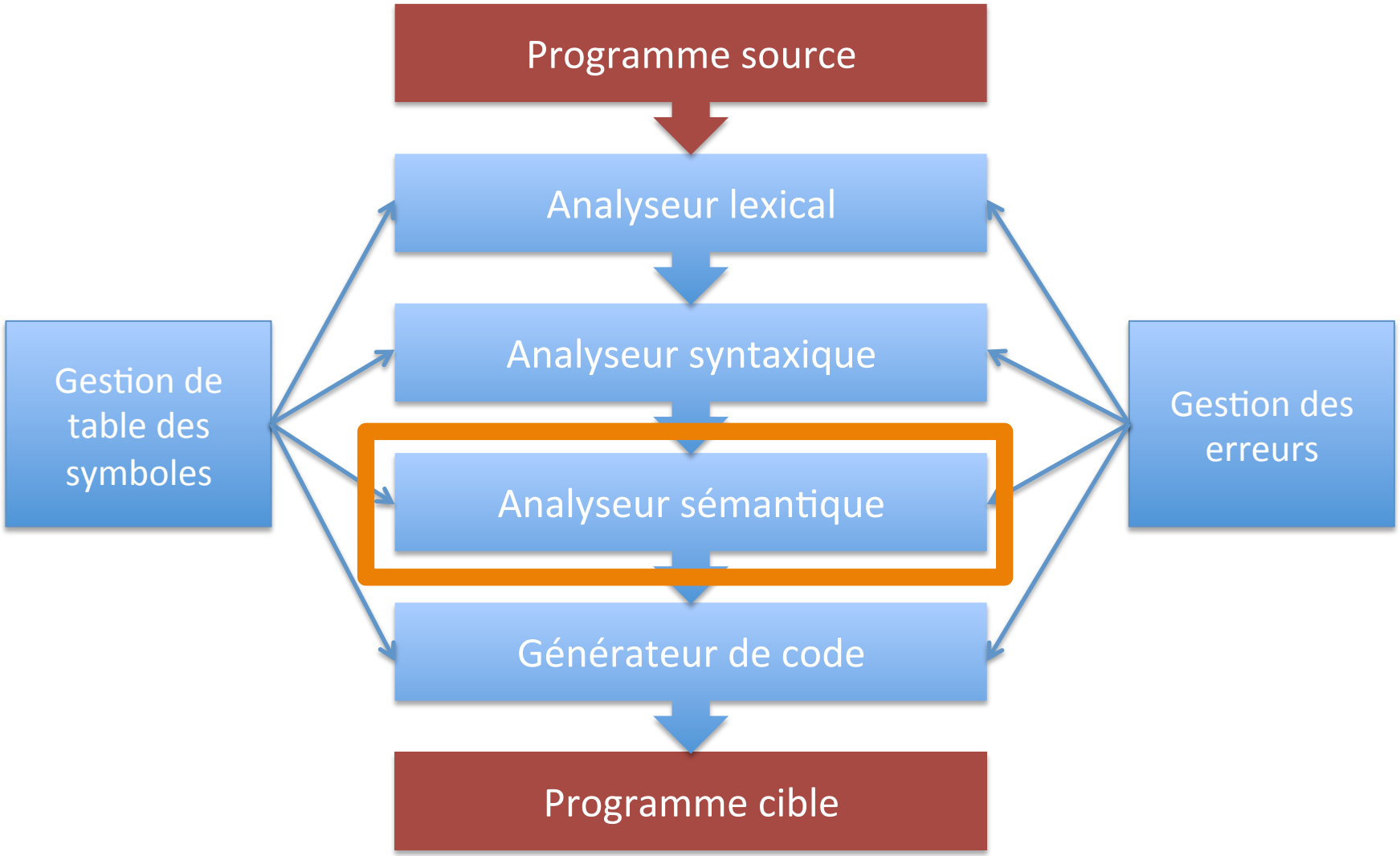
- Un fichier Yacc se compose de 3 parties :
  - Déclarations C
  - Déclarations de tokens
  - Associativité, priorité

  %%
  - Règles sous la forme :

```
<NT > : <patern_1> <action_1> | ...
      | <patern_i> <action_i> ;
```

  %%
  - Toute fonction C utile
- CF. demo.y pour un exemple fonctionnel

Compilateur



Analyseur sémantique

- But : Vérifier toutes les autres contraintes
- Exemples :
  - Vérification des types :
    - int a; a = a <= 5;
  - Vérification des appels de fonctions
  - Etc.
- Peuvent être vérifiées en C
  - Nécessite une représentation interne du programme compilé

Arbre syntaxique

- N-tree représentant le programme compilé
  - Ex : `if(ok and i < 5) then write i; else write 0; …`

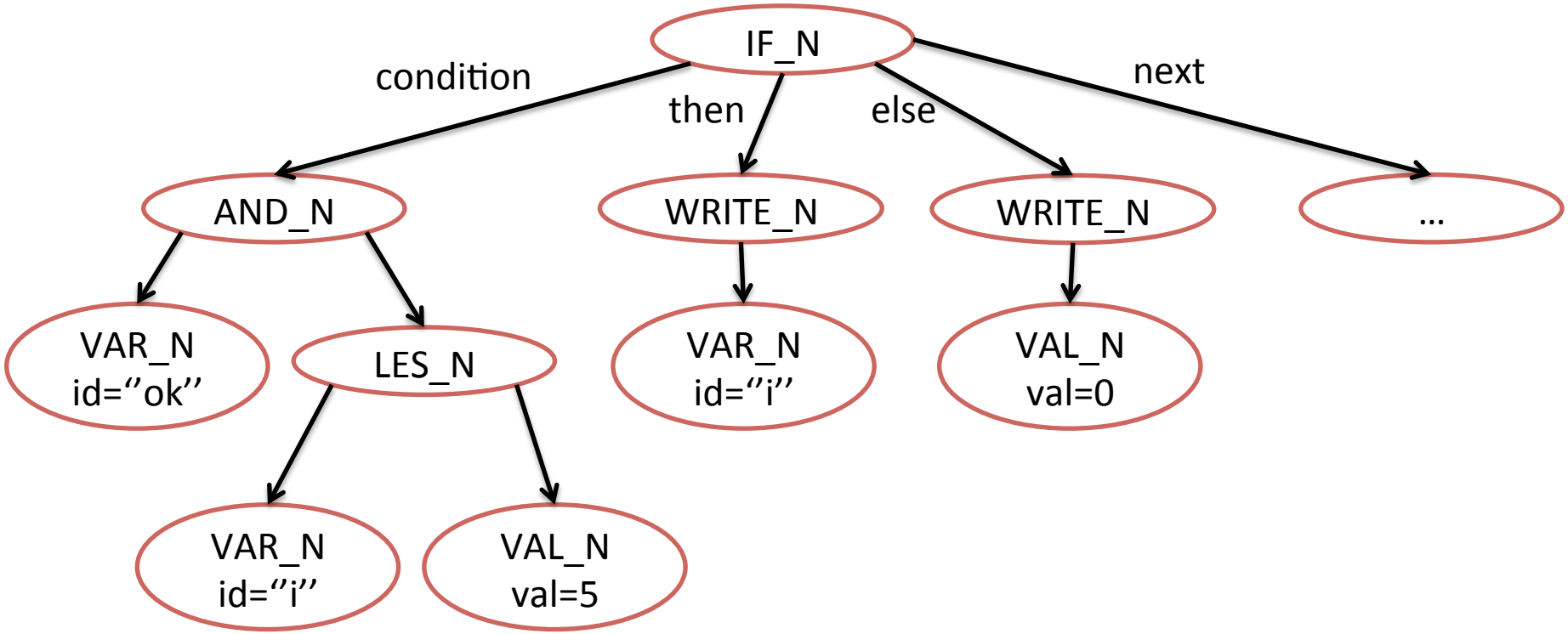
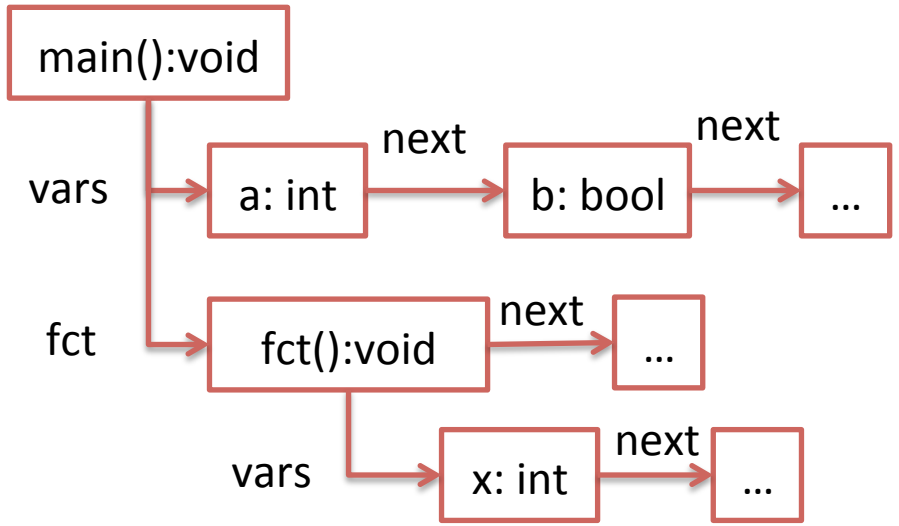


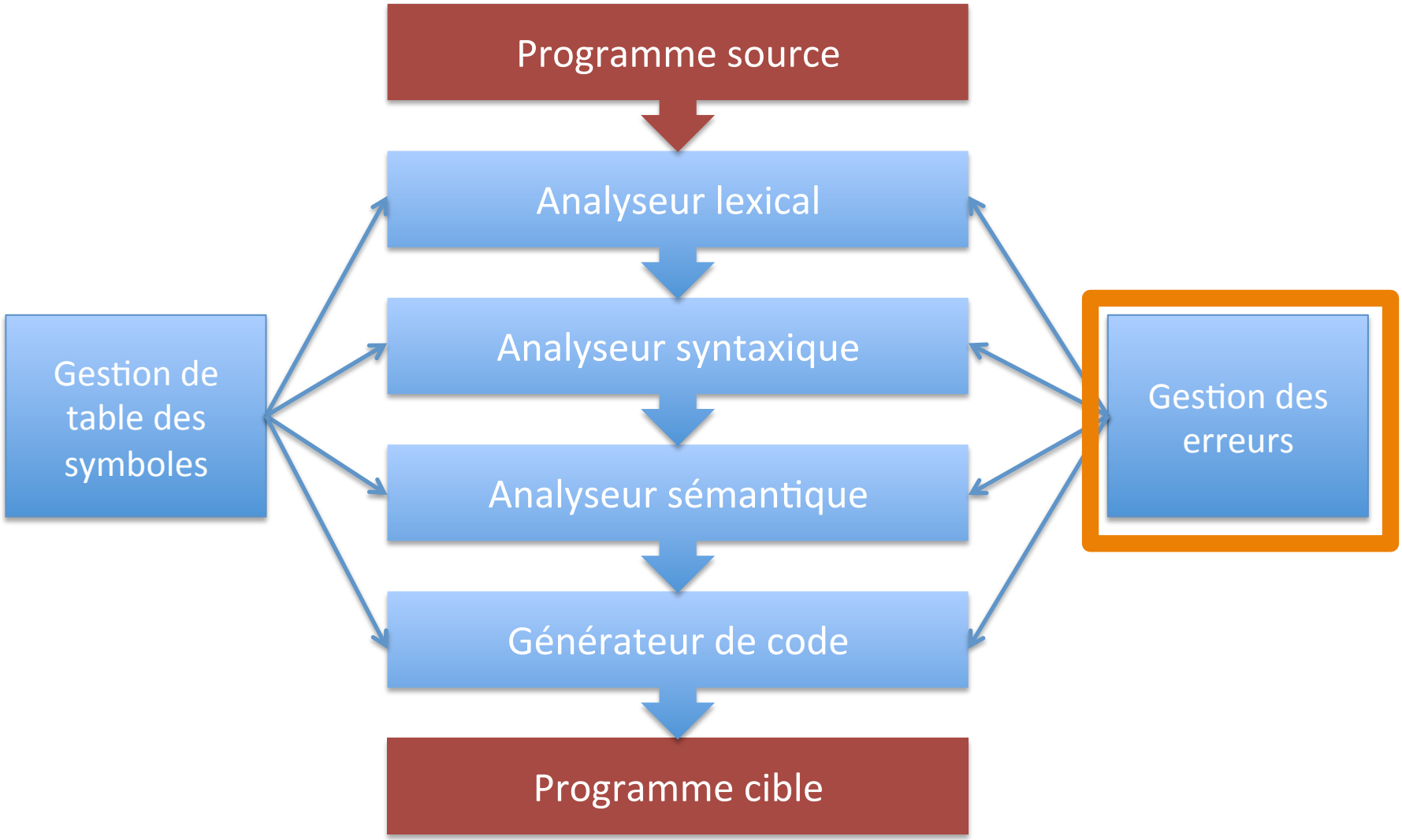
Table des symboles

- Représentation des variables et fonctions déclarées
  - Ex :

```
main():void;
var
    a : int; b : bool; …
    function fct():void;
    var
        x : int; …
    begin … end;
    …
begin … end;
```



Compilateur

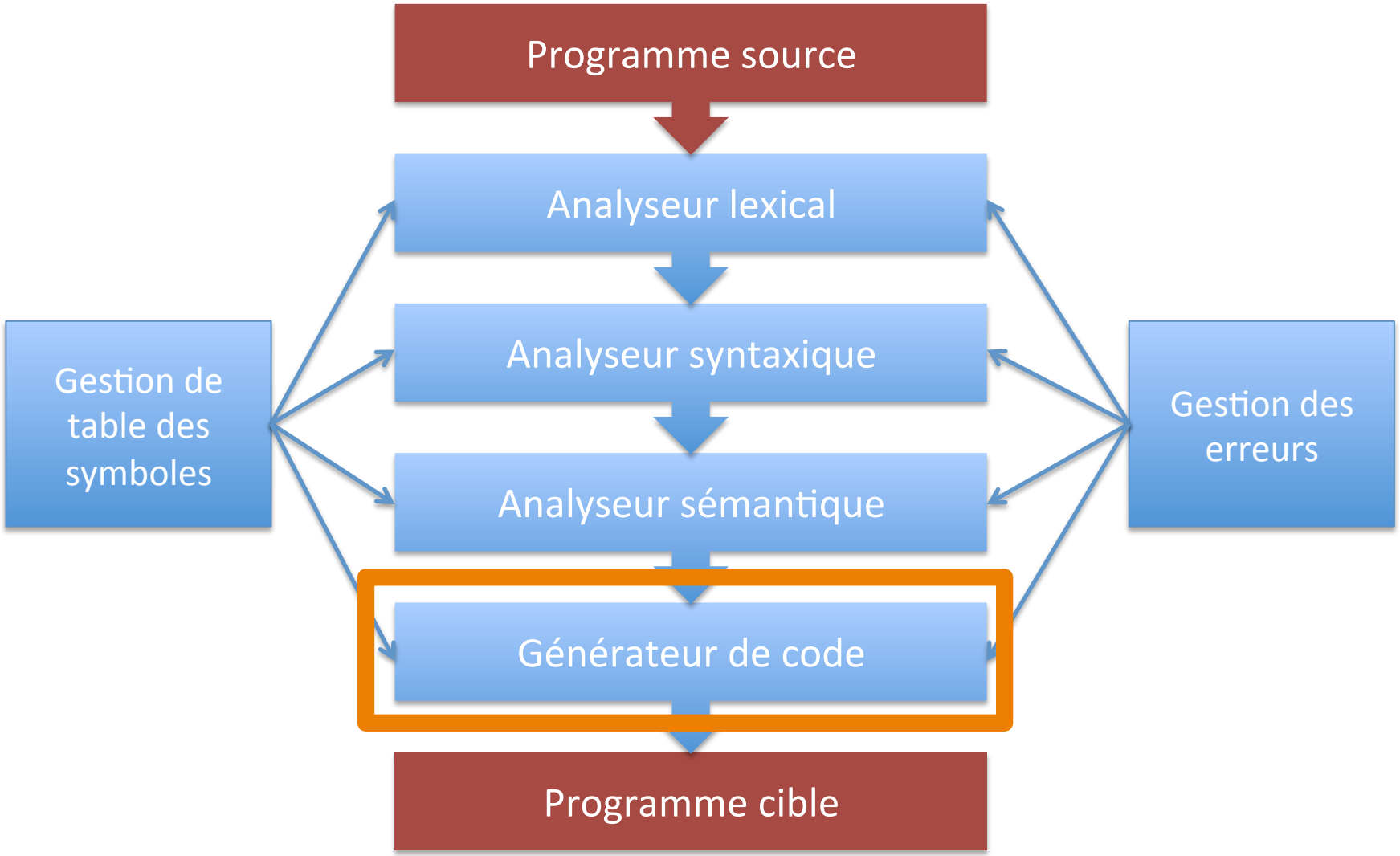


Gestion des erreurs

- Multiples raisons pouvant provoquer l'arrêt du compilateur

- Vivement conseillé d'avoir une interface centralisée de reporting
  - NB : En C, une simple fonction est suffisante

Compilateur

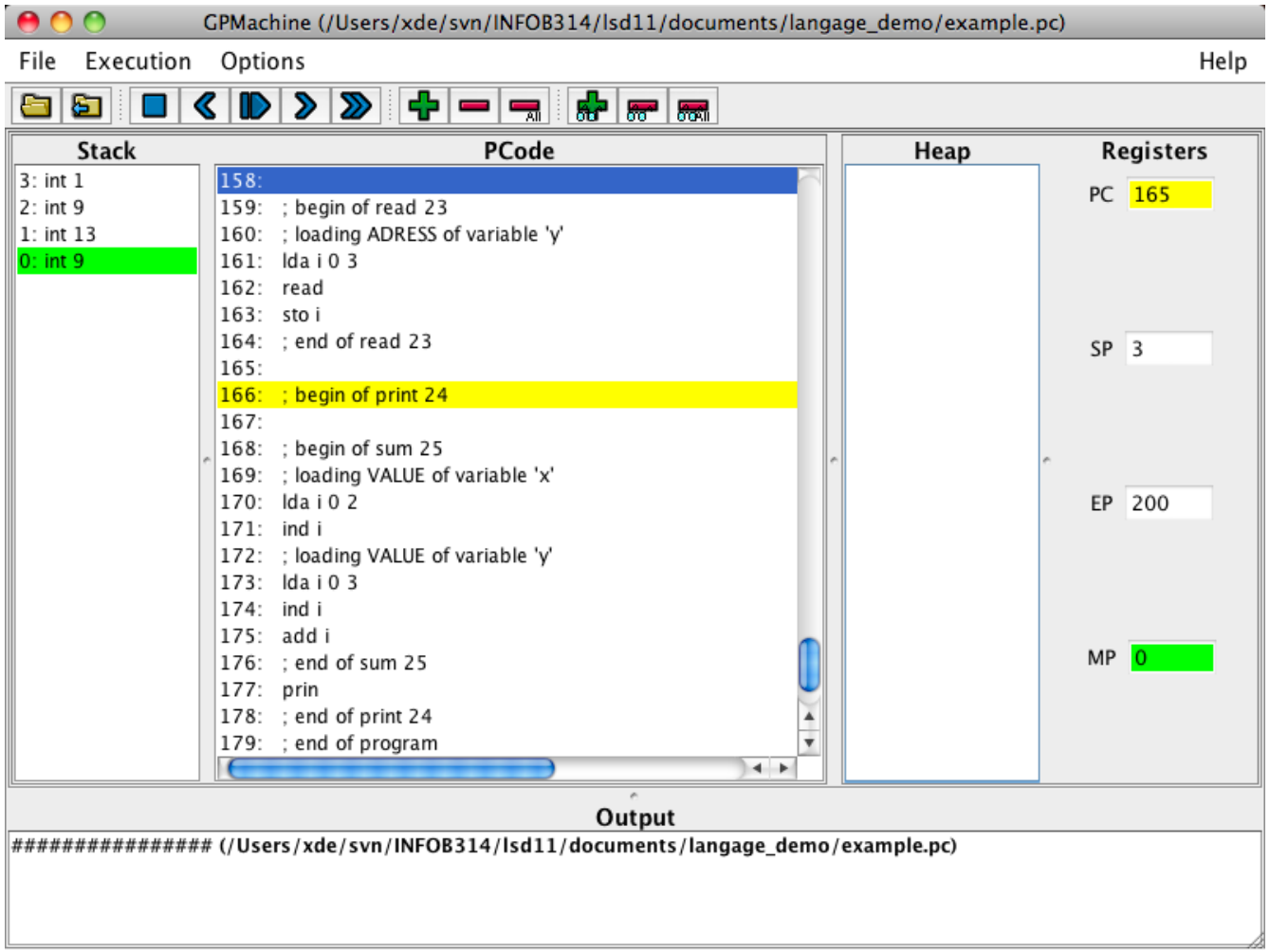


Génération de code

- But : passer de la représentation interne valide par rapport la spécification au code cible
- Exemples:
  - JVM pour Java
  - CAM pour CAML
  - WAM pour Prolog
  - P-Machine pour Pascal
  - Code binaire pour de l'assembleur
- P-Code pour la P-Machine dans notre cas

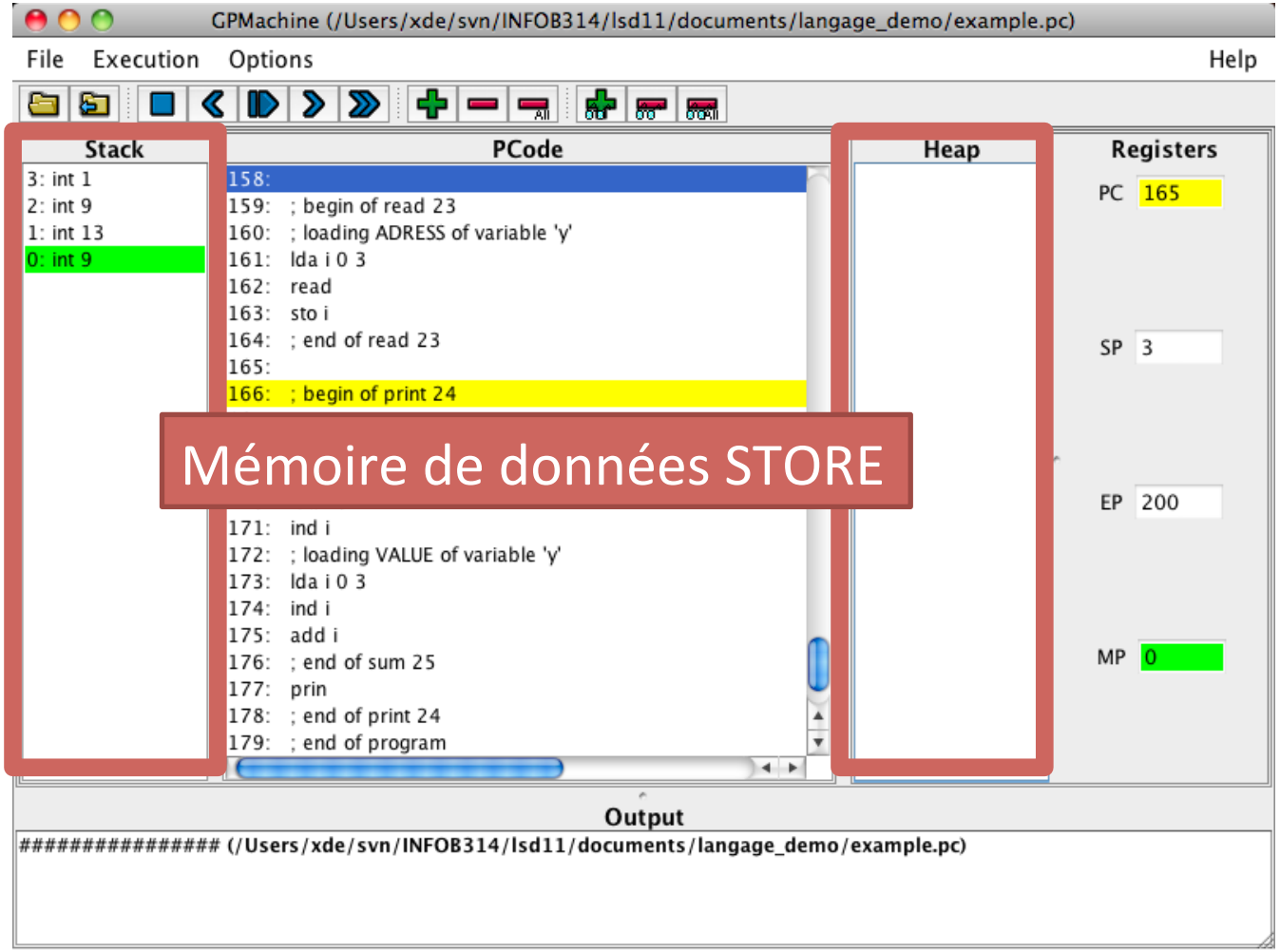
GPMachine

- Disponible à l'adresse : http://www.info.fundp.ac.be/~gpm



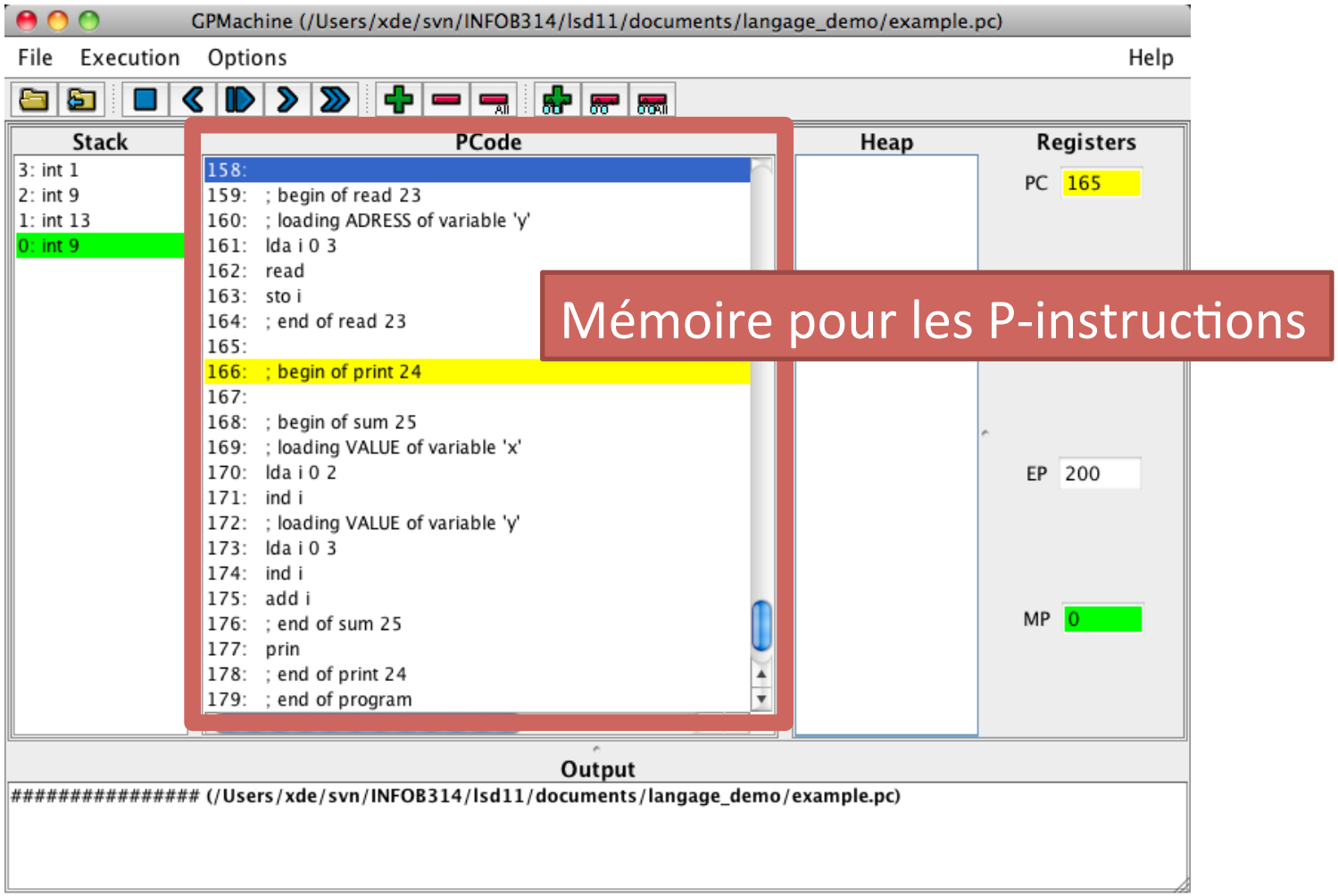
GPMachine

- Disponible à l'adresse : http://www.info.fundp.ac.be/~gpm



GPMachine

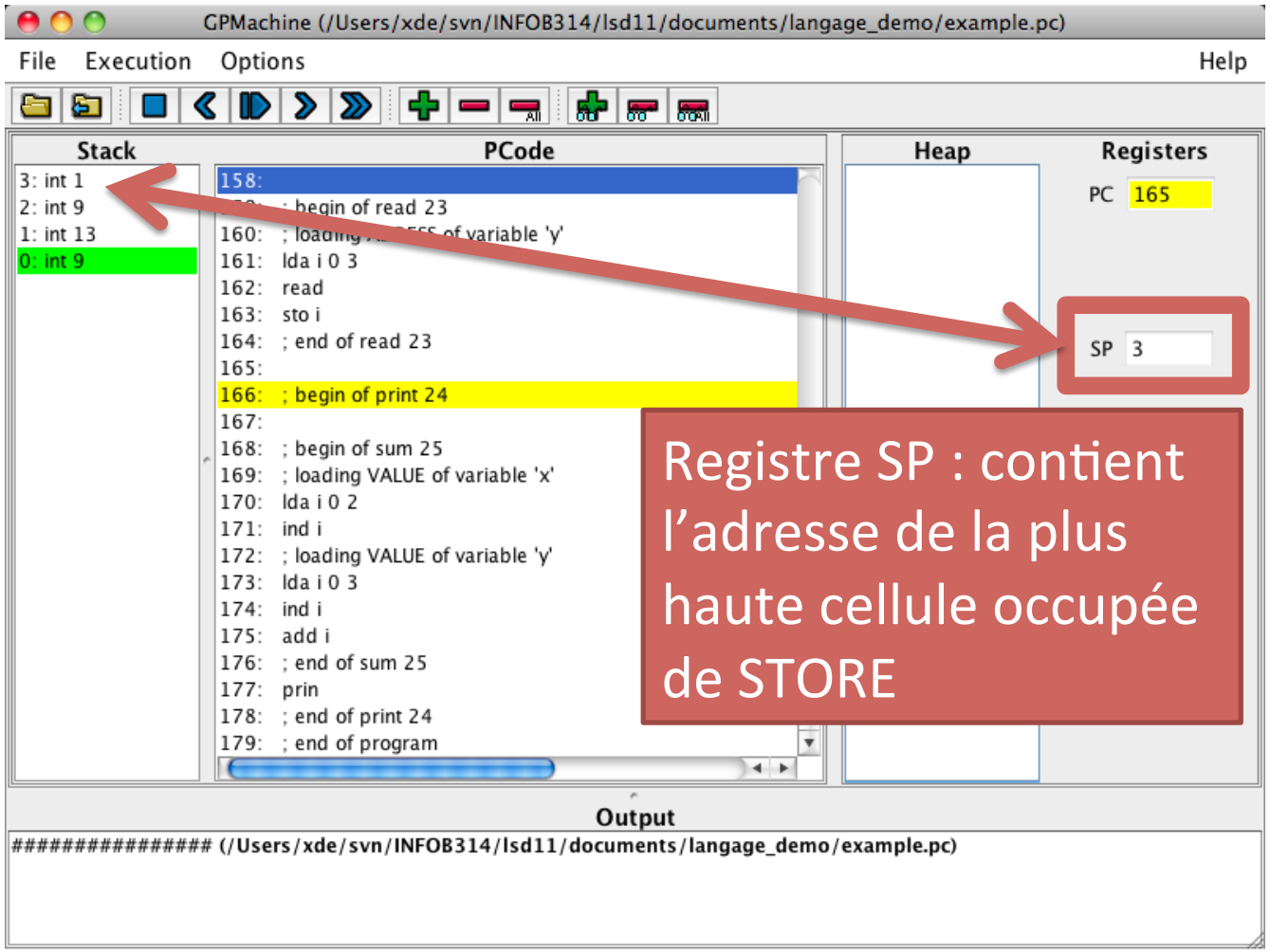
- Disponible à l'adresse : http://www.info.fundp.ac.be/~gpm



Mémoire pour les P-instructions

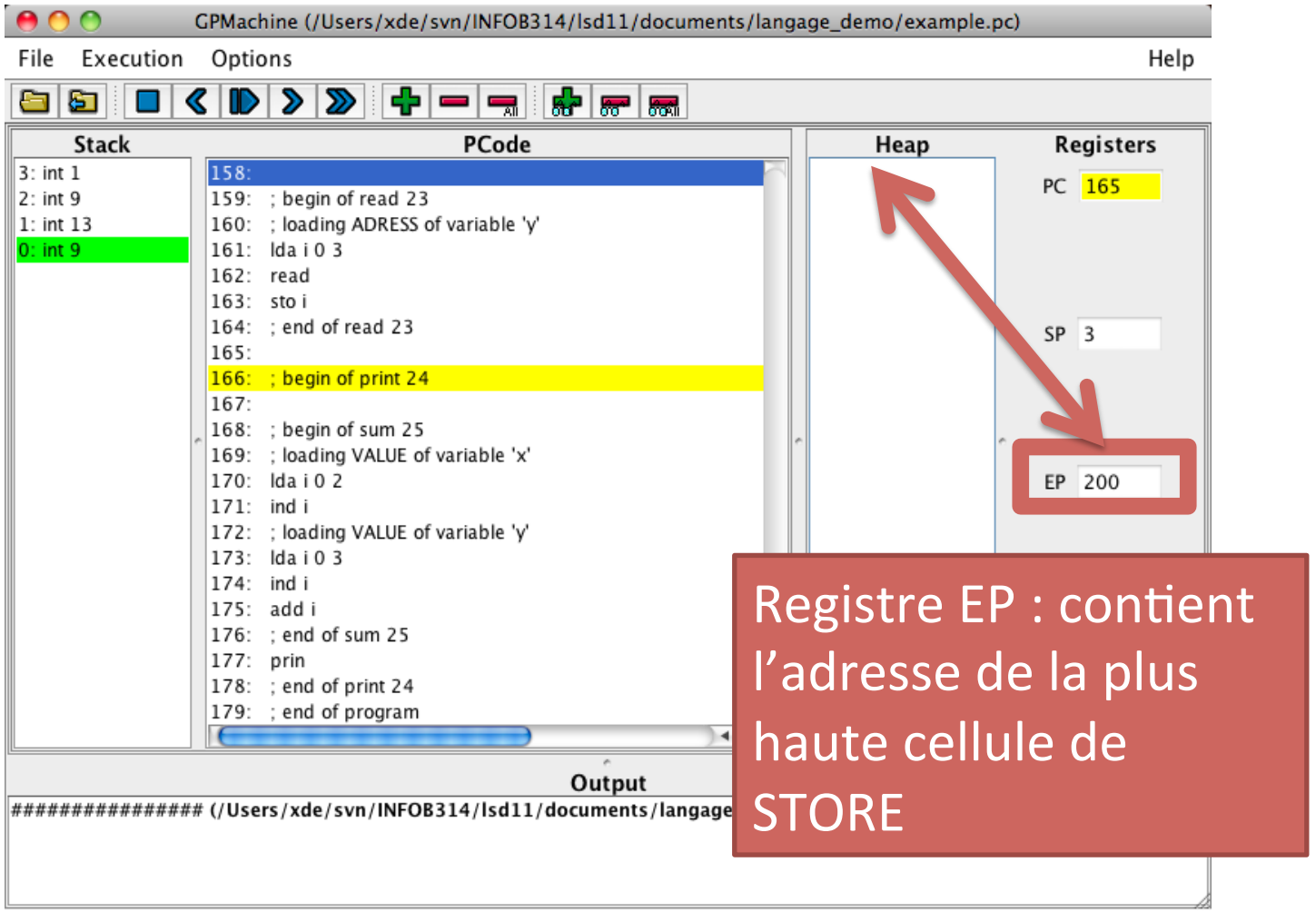
GPMachine

- Disponible à l'adresse : http://www.info.fundp.ac.be/~gpm



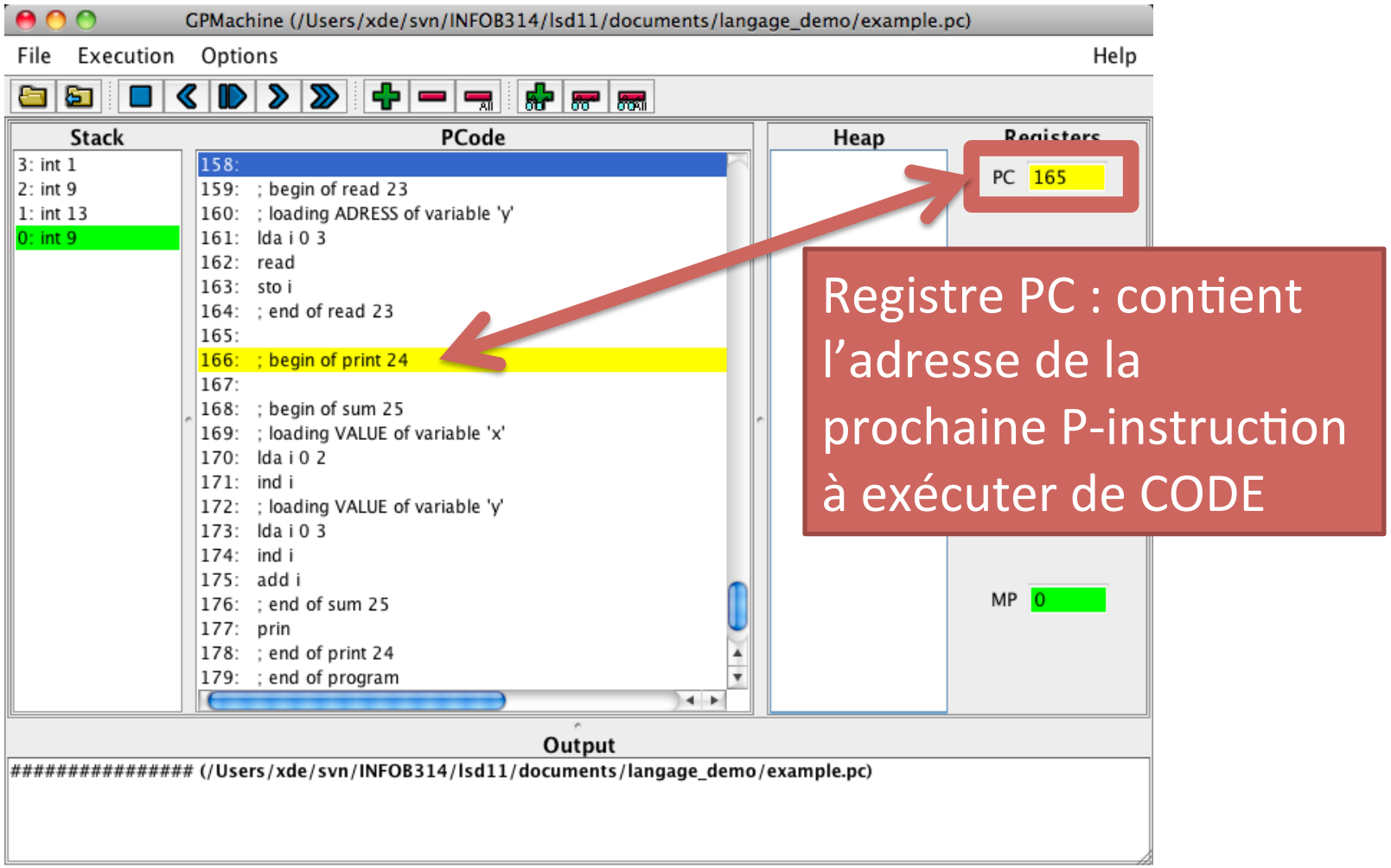
GPMachine

- Disponible à l'adresse : http://www.info.fundp.ac.be/~gpm



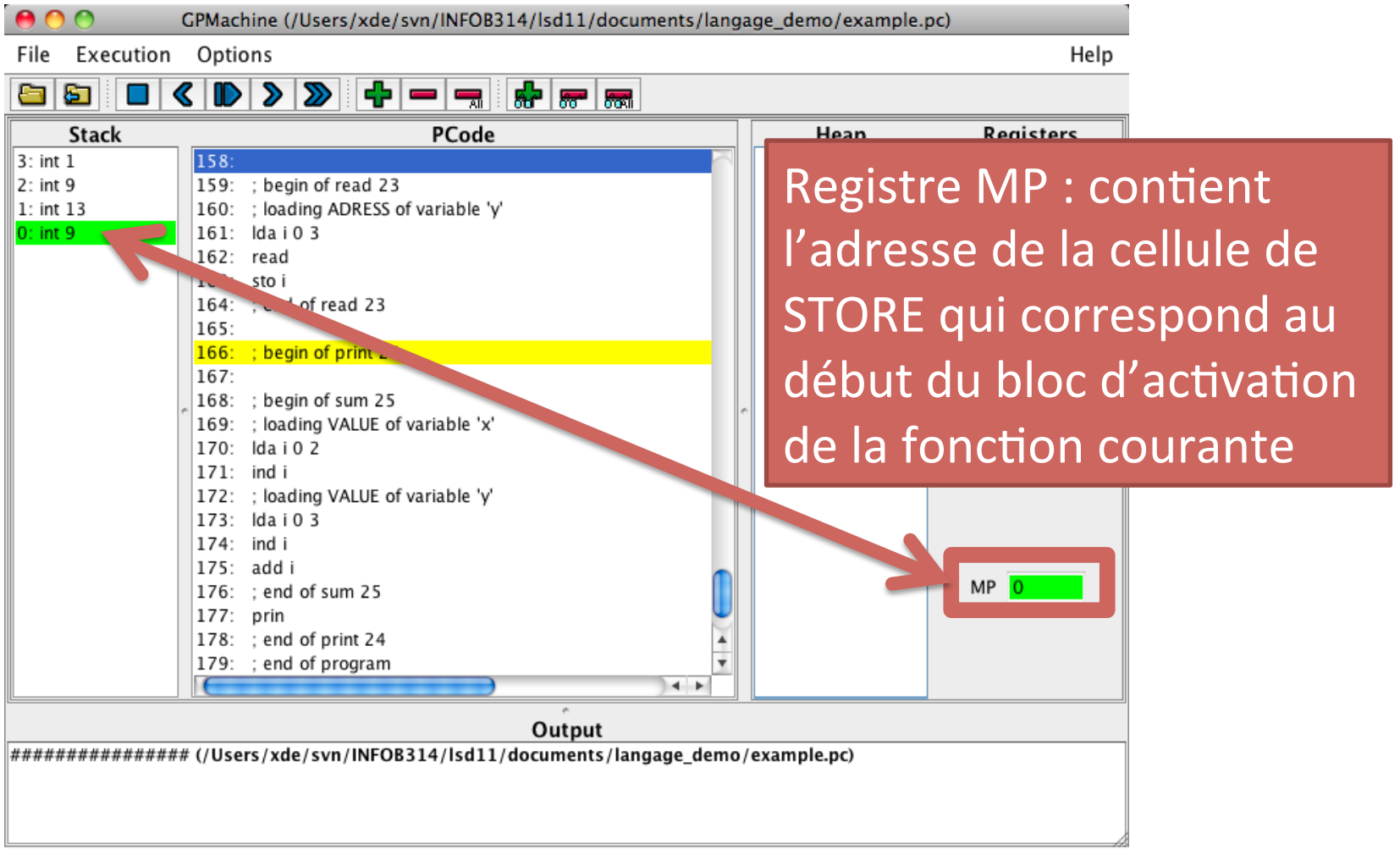
GPMachine

- Disponible à l'adresse : http://www.info.fundp.ac.be/~gpm



GPMachine

- Disponible à l'adresse : http://www.info.fundp.ac.be/~gpm



Registre MP : contient l'adresse de la cellule de STORE qui correspond au début du bloc d'activation de la fonction courante

GPMachine

- Fonctionne en notation polonaise inversée
  - Eg. (1 + 2) = (1 2 +)
- Manipule
  - entier ("i"), réel ("r"), booléen ("b"), adresse ("a")
  - "N" représente un type numérique : N ∈ {i,r,a}
  - "T" représente un type quelconque : T ∈ {i,r,a,b}
- Remarques
  - P-Code créé pour la traduction de Pascal
  - LSD12 utilise une petite partie de P-Code
  - On ne présente que la partie de P-Code nécessaire à la traduction de LSD12
  - Quelques différences existent entre l'implémentation de la P-Machine qui vous est fournie et la définition qui est donnée dans WilHelm et Maurer (registre EP,instruction define,...)

P-Instructions pour les expressions

P-Instruction	Signification	Condition	Résultat
add N	STORE[SP-1] := STORE[SP-1] $+_N$ STORE[SP] ; SP := SP − 1	(N,N)	(N)
mul N	STORE[SP-1] := STORE[SP-1] $*_N$ STORE[SP] ; SP := SP − 1	(N,N)	(N)
neg N	STORE[SP] := $-_N$ STORE[SP]	(N)	(N)
or	STORE[SP-1] := STORE[SP-1] *or* STORE[SP] ; SP := SP − 1	(b,b)	(b)
not	STORE[SP] := *not* STORE[SP]	(b)	(b)
equ	STORE[SP-1] := STORE[SP-1] $=_T$ STORE[SP] ; SP := SP − 1	(T,T)	(b)
les	STORE[SP-1] := STORE[SP-1] $<_T$ STORE[SP] ; SP := SP − 1	(T,T)	(b)

- (N,N) indique que STORE[SP] et STORE[SP-1] sont de type numérique.
- Les valeurs VRAI et FAUX sont codées respectivement par 1 et 0 dans la P-Machine

Lecture et écriture en mémoire

P-Inst	Signification	Condition	Résultat
ldc T c	SP := SP + 1 ; STORE[SP] := c	() Type(c) = T	(T)
lod T d q	SP := SP + 1 ; STORE[SP] := STORE[ad(d,q)]	() Type(STORE[ad(d,q)]=T	(T)
lda T d q	SP := SP + 1 ; STORE[SP] := ad(d,q)	() Type(STORE[ad(d,q)]=T	(a)
ind T	STORE[SP]:=STORE[STORE[SP]]	(a)	(T)
sto T	STORE[STORE[SP-1]]:=STORE[SP] ; SP := SP − 2	(a,T)	()

- d := différence entre profondeur de l'appel et de la déclaration
- q := adresse relative
- ad(d,q) := base(d,MP)+q
- base(d,MP) := if (d=0) then MP
  else base(d-1,STORE[MP+1]) f

Traduction des instructions

- Idée : travailler récursivement

Fonction		Condition
$PCode(z := e) =$	$PCode_G(z);$ $PCode_D(e);$ $sto\ T$	$Type(z) = Type(e) = T$
$PCode_D(e_1 + e_2) =$	$PCode_D(e_1);$ $PCode_D(e_2);$ $add\ N$	$Type(e_1) = Type(e_2) = N$
$PCode_D(e_1 * e_2) =$	$PCode_D(e_1);$ $PCode_D(e_2);$ $mul\ N$	$Type(e_1) = Type(e_2) = N$
$PCode_D(c) =$	$ldc\ T\ c$	c constante et $Type(c) = T$
$PCode_G(z) =$	$lda\ T\ d(z)\ q(z)$	z variable et $Type(z) = T$
$PCode_D(z) =$	$PCode_G(z);$ $ind\ T$	z variable et $Type(z) = T$

Où $d(z)$ et $q(z)$ sont respectivement la profondeur relative et l'adresse relative de z.

Traduction des instructions

Exemple :

$$
\begin{aligned}
&PCode(x := 2 * 3) \\
&\quad = PCode_G(x);\ PCode_D(2 * 3);\ sto\ i \\
&\quad = lda\ i\ d(x)\ q(x);\ PCode_D(2 * 3);\ sto\ i \\
&\quad \vdots \\
&\quad = lda\ i\ d(x)\ q(x);\ ldc\ i\ 2;\ ldc\ i\ 3;\ mul\ i;\ sto\ i
\end{aligned}
$$

- Cf. Document “gpmachine-reference.pdf”