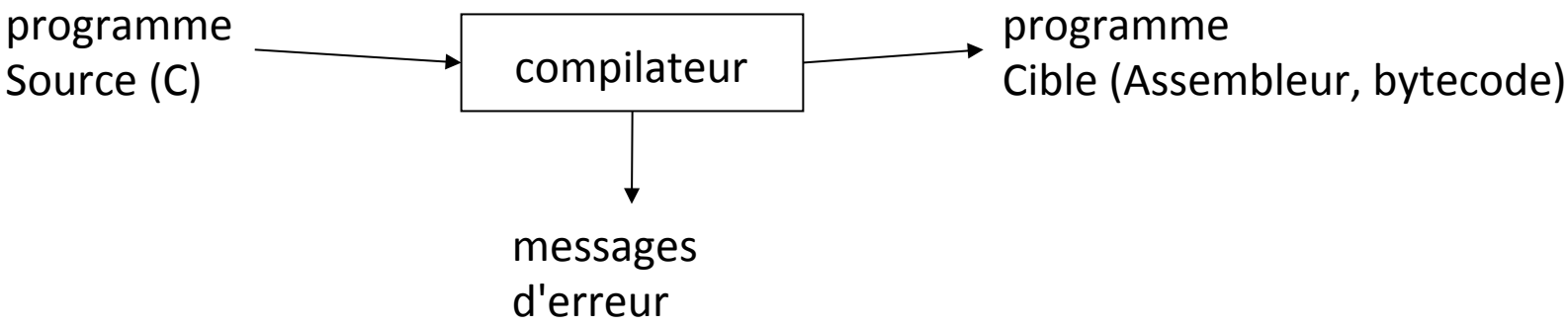
EISE4 : Compilation 1

Plan de cours/TP

- Généralités sur les compilateurs
- Lex / Yacc, analyseurs lexicaux, analyseurs syntaxiques, générateurs d'analyseurs
- Lex / Yacc, aspects sémantiques, Structures de données, Arbres de Syntaxe Abstraite (AST)
- Génération de code pour machine virtuelle
- Multithreading
- Assembleur, machine virtuelle
- Structures de données, manipulation des AST, dot
- Compilateur EISE (1)
- Compilateur EISE (2)
- Compilateur EISE (3)

Compilateur

Un compilateur est un programme qui traduit un programme source en programme cible (généralement de niveau d'abstraction inférieur)



Premier compilateur : compilateur Fortran de J. Backus (1957)

Langage source : langage de haut niveau (C, C++, Java, Pascal, Fortran...)

Langage cible : langage de bas niveau (assembleur, langage machine)

Bibliographie

Aho, Sethi, Ullman, 1986/2007. *Compilateurs. Principes, techniques et outils*, Pearson Education France.

http://igm.univ-mlv.fr/~laporte/compil/plan.html

Ce support de cours est inspiré du manuel ci-dessus et du cours de Dominique Perrin

Cousins des compilateurs

Interpréteurs

Diffèrent d'un compilateur par l'intégration de l'exécution et de la traduction.
Utilisés pour les langages de commande ou langages interprétés (python, perl, php)

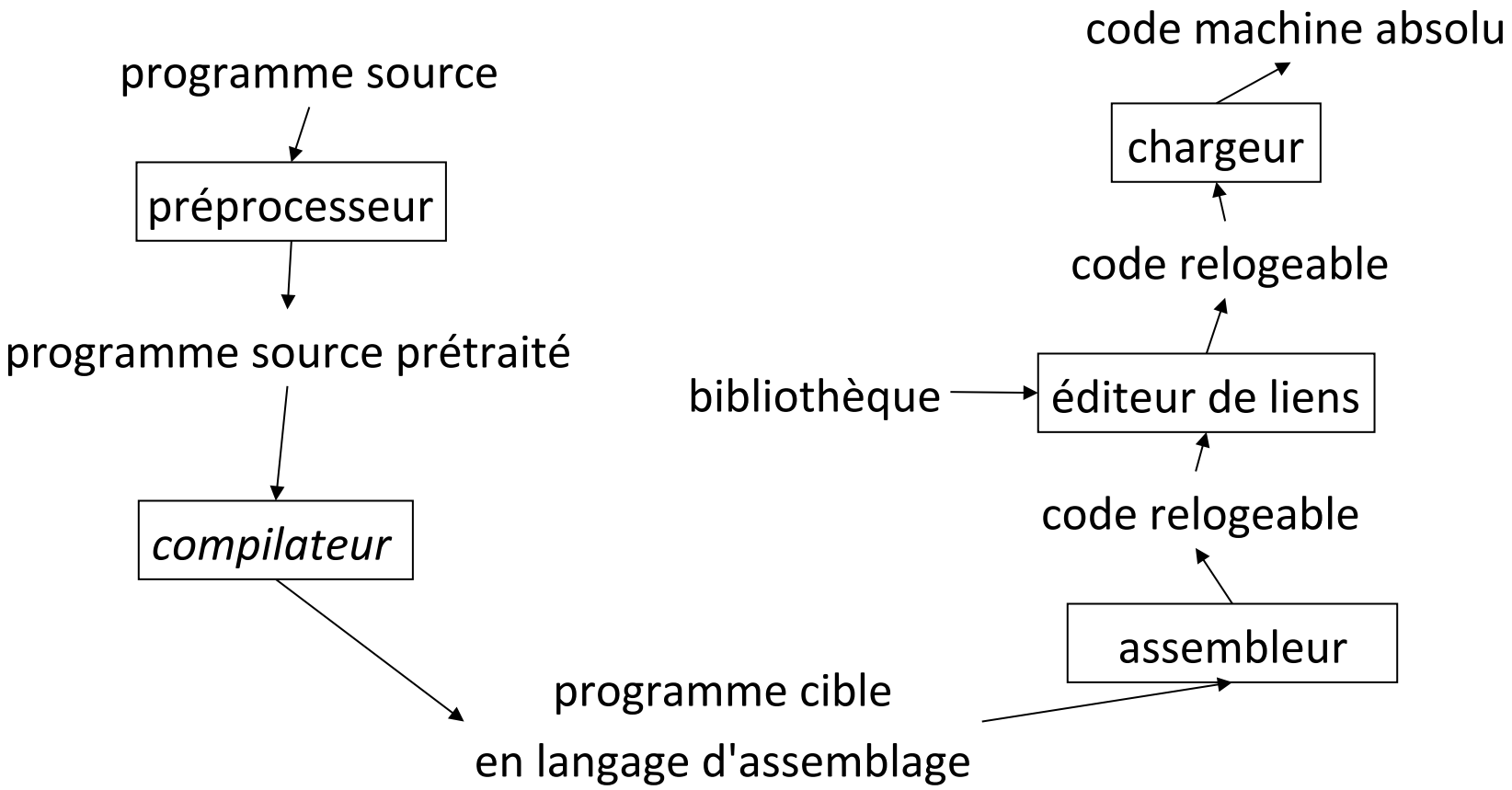
Formateurs de texte

Traduit le code source dans le langage de commande d'une imprimante

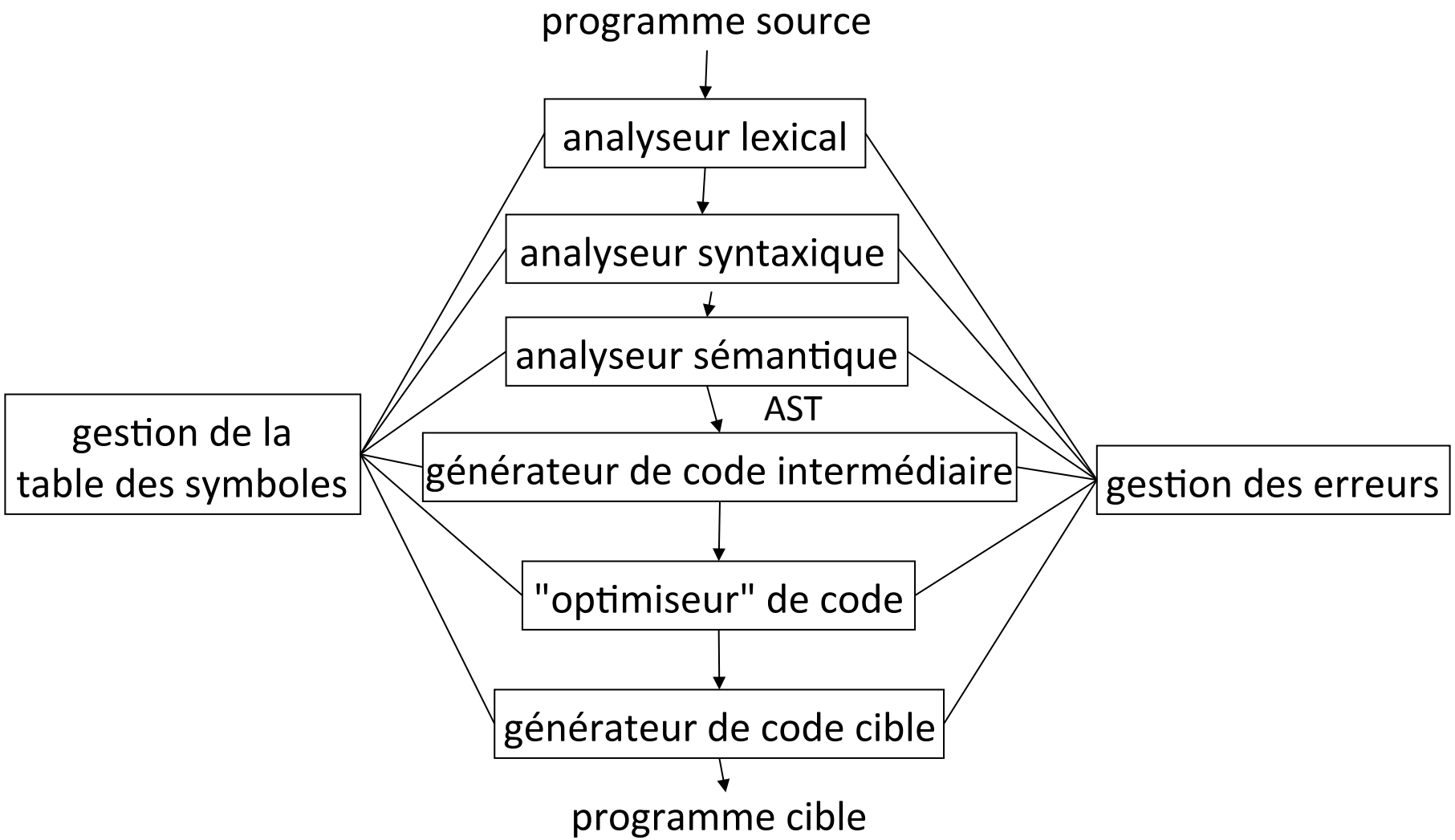
Préprocesseurs

Effectuent des substitutions de définitions, des transformations lexicales, des définitions de macros, etc.

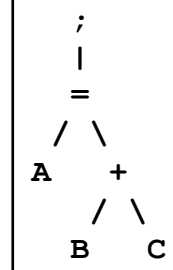
L'environnement d'un compilateur



Les phases de la compilation



Les Phases d'un compilateur

Phase	Sortie	Exemple
Programmeurr (producteur de code source)	Chaîne de car. source	`A=B+C;`
Scanner (fait l'analyse *lexicale*)	Tokens, entités, lexèmes	`'A', '=', 'B', '+', 'C', ';'` et table des *symboles* avec noms
Parser (fait *l'analyse syntaxique*, à partir d'une grammaire du langage)	Parse tree ou abstract syntax tree	
Analyse sémantique (vérification des types, etc)	Parse tree ou abstract syntax tree annotés	

Générateur de code Intermediaire	Code trois adresses	`int2fp B t1`
		`+ t1 C t2`
		`:= t2 A`

Optimiseur	Code trois adresses	`int2fp B t1` `+ t1 #2.3 A`

Générateur de code	Code assembleur	`MOVF r1,#2.3`
		`ADDF2 r2,r1`
		`MOVF A,r2`

Optimiseur local (peep-hole)	Code assembleur	`ADDF2 #2.3,r2` `MOVF r2,A`

Analyse lexicale

Analyse du programme source en entités/constituants minimaux, les **lexèmes**

On passe d'une construction décrite en langage source

```
position = initial + vitesse * 60
```

à

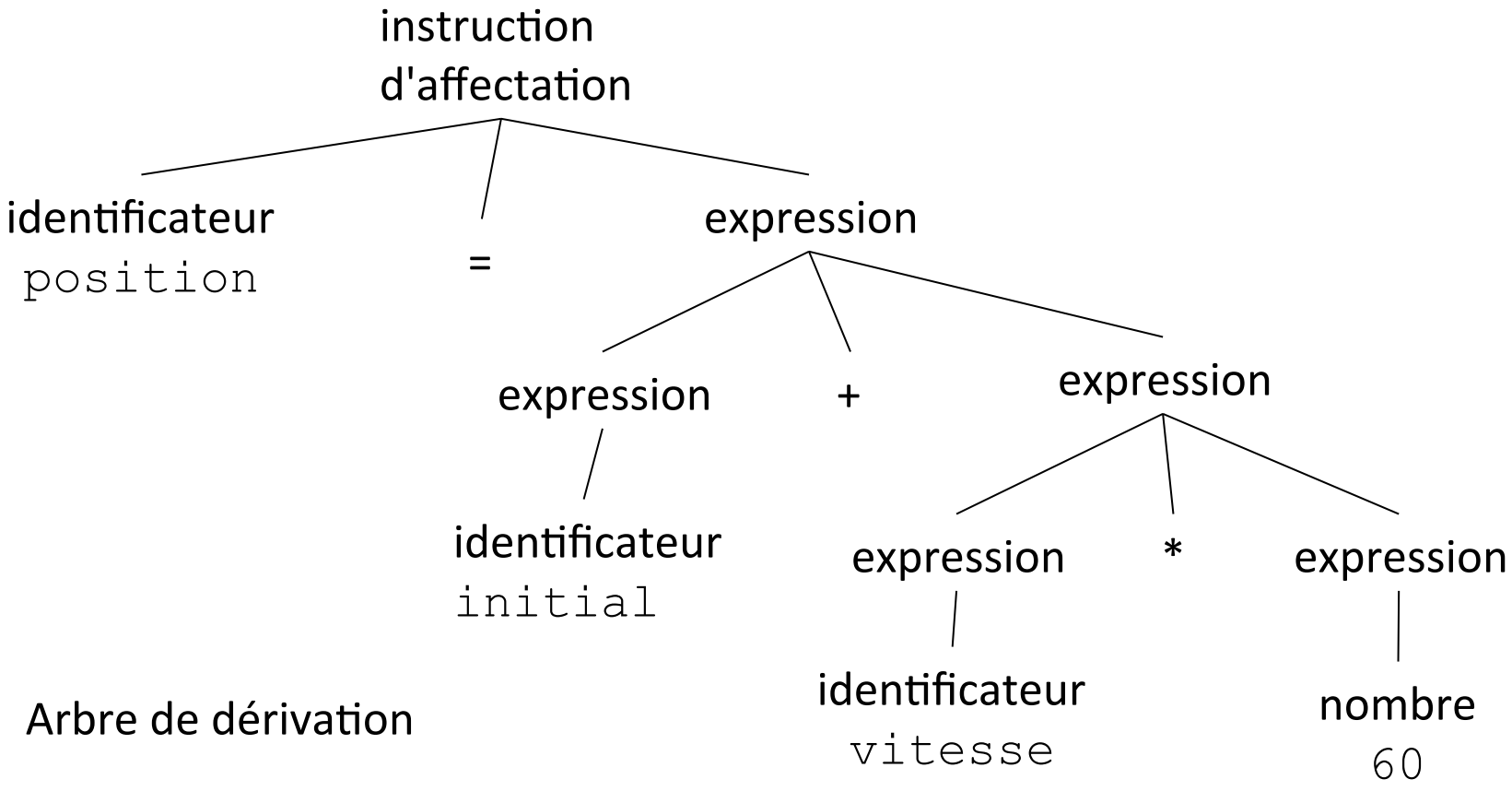
```
[id, 1] [=] [id, 2] [+] [id, 3] [*] [60]
```

Les identificateurs rencontrés sont placés un par un dans la table des symboles

Les blancs et les commentaires sont éliminés

Analyse syntaxique

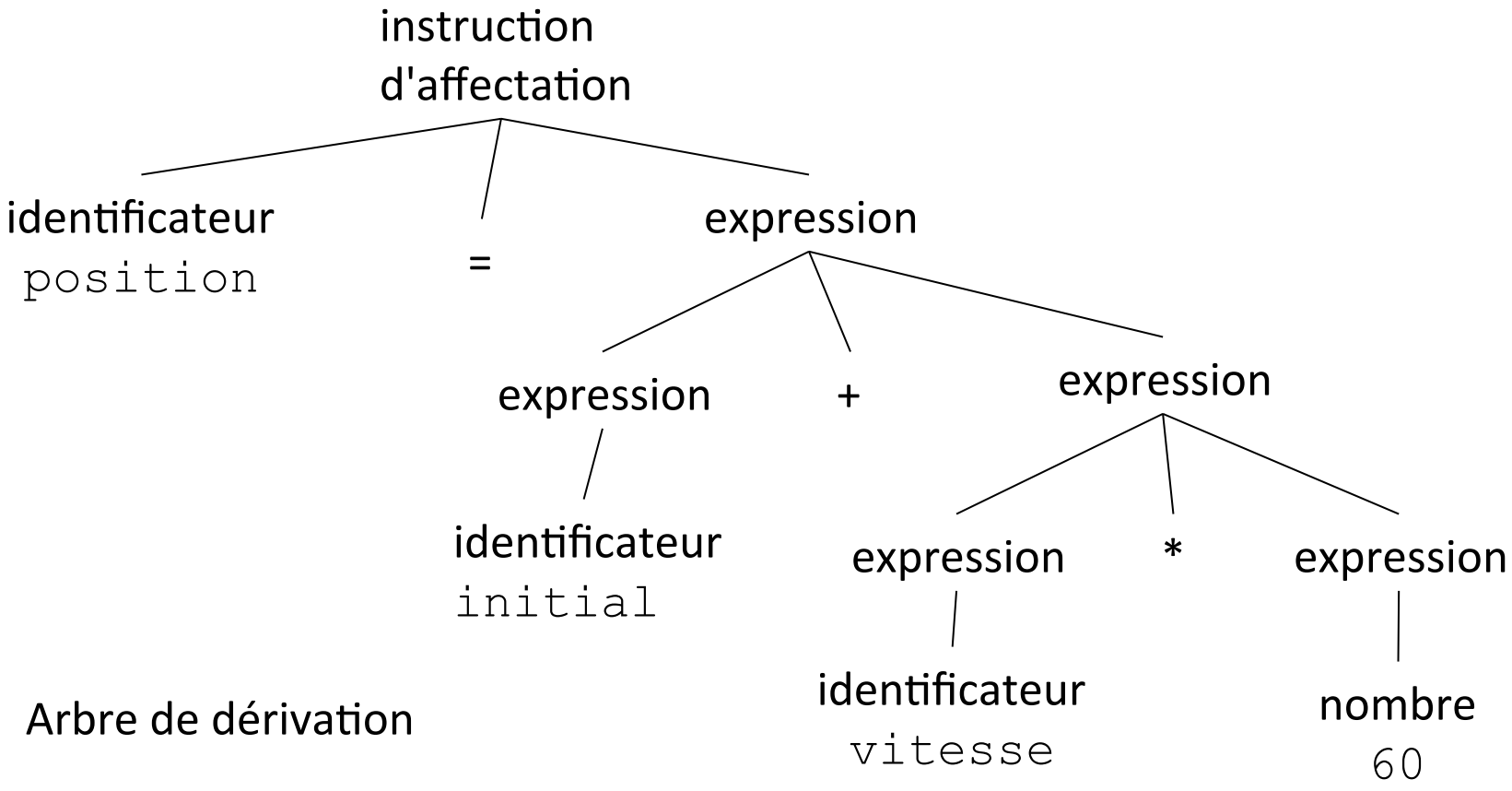
On reconstruit la structure syntaxique de la suite de lexèmes fournie par l'analyseur lexical. On fait cela pour savoir si un texte source donné est conforme à la grammaire du langage.



Arbre de dérivation

Analyse sémantique

En plus de l'analyse syntaxique, on procède à des vérifications sémantiques. Est-ce que le type de la variable affectée (LHS) est compatible avec celui de l'expression droite (RHS) ?



Arbre de dérivation

Génération de code intermédiaire

Programme pour une machine abstraite

Représentations utilisées

- Code à trois adresses

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

- Arbres syntaxiques

"Optimisation" de code

Elimination des opérations inutiles pour produire du code plus efficace.

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

La constante est traduite en réel flottant à la compilation, et non à l'exécution

La variable temp3 est éliminée

Génération de code cible

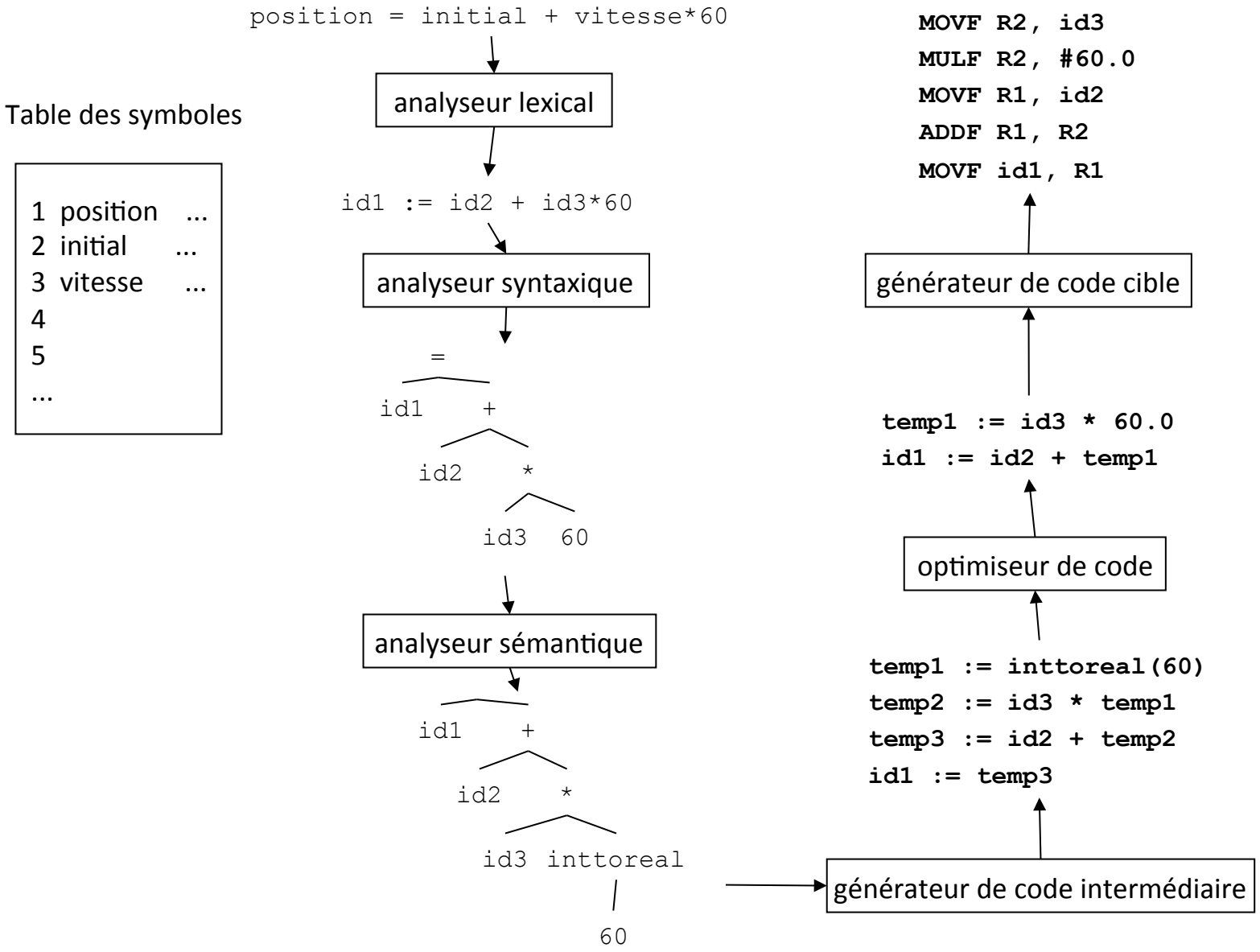
La dernière phase produit du code en langage d'assemblage

Un point important est l'utilisation des registres

```
temp1 := id3 * 60.0        MOVF R2, id3
id1 := id2 + temp1         MULF R2, #60.0
                           MOVF R1, id2
                           ADDF R1, R2
                           MOVF id1, R1
```

La première instruction transfère le contenu de id3 dans le registre R2

La seconde multiplie le contenu du registre R2 par la constante 60.0


position = initial + vitesse*60
analyseur lexical
Table des symboles
1 position ...
2 initial ...
3 vitesse ...
4
5
...
id1 := id2 + id3*60
analyseur syntaxique
=
id1 +
id2 *
id3 60
analyseur sémantique
id1 +
id2 *
id3 inttoreal
60
générateur de code intermédiaire
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
optimiseur de code
temp1 := id3 * 60.0
id1 := id2 + temp1
générateur de code cible
MOVF R2, id3
MULF R2, #60.0
MOVF R1, id2
ADDF R1, R2
MOVF id1, R1

Langages d’assemblage

Les langages d’assemblage ou assembleurs sont des versions un peu plus lisibles du code machine avec des noms symboliques pour les opérations et pour les opérandes

```
MOV R1, a
ADD R1, #2
MOV b, R1
```

Chaque processeur a son langage d'assemblage

Les langages d'assemblage d'un même constructeur se ressemblent

Assemblage

L'**assemblage** consiste à traduire ce code « texte » en code binaire

La forme la plus simple est l'**assemblage en deux passes**

Première passe

On crée une table des symboles (distincte de celle du compilateur) pour associer des adresses mémoires aux identificateurs

identificateur	adresse
a	0
b	4

Assemblage

Deuxième passe

On crée du code machine résolu, c'est-à-dire que toutes les étiquettes de code sont déterminées, et toutes les références (sauts, chargements) sont connues

Les instructions précédentes peuvent être traduites en :

```
MOV R1, a      0001 01 00 00000000 *
ADD R1, #2     0011 01 10 00000010
MOV b, R1      0010 01 00 00000100 *
```

La table des symboles de l'assembleur sert aussi à l'édition de liens

Elle permet de remplacer les références à des noms externes contenus dans d'autres fichiers. On ne verra pas cela dans ce cours.

Groupement des phases du compilateur

Partie frontale (*front end*)

Regroupe tout ce qui dépend du langage source plutôt que de la machine cible. On peut utiliser la même partie frontale sans s'intéresser au langage cible

Partie arrière (*back end*)

Regroupe le reste

Passes

Plusieurs phases peuvent être regroupées dans une même passe consistant à lire un fichier et en écrire un autre

Analyse lexicale, syntaxique, sémantique et génération de code intermédiaire peuvent être regroupées en une seule passe

La réduction du nombre de passes accélère le traitement

Outils logiciels disponibles

Outils d'aide à la construction de compilateurs

Analyseurs lexicaux et Générateurs d'analyseurs lexicaux

Engendrent un analyseur lexical (*scanner, lexer*) sous forme d'automate fini à partir d'une spécification sous forme d'expressions rationnelles

Flex, Lex

Analyseurs syntaxiques et Générateurs d'analyseurs syntaxiques

Engendrent un analyseur syntaxique (*parser*) à partir d'une grammaire

Bison, Yacc

Générateurs de traducteurs

Engendrent un traducteur à partir d'un schéma de traduction (grammaire + règles sémantiques)

Bison, Yacc

Compilation d'un programme C

Si on compile par **`gcc -S essai.c`** le fichier suivant :

```
#include <stdio.h>

int main(void) {
  printf("bonjour\n") ; }
```

on obtient de l'assembleur 8086 avec

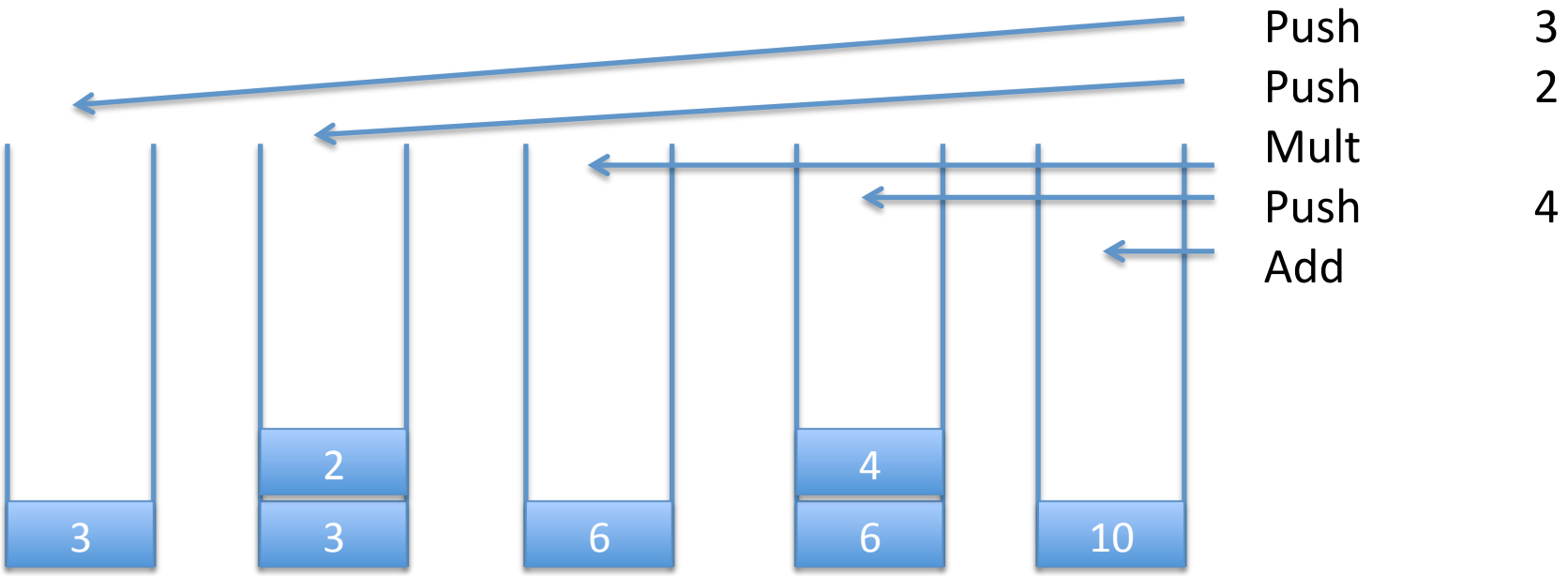
- **`epb`** pour base pointer
- **`esp`** pour stack pointer
- **`pushl`** pour push long

etc.

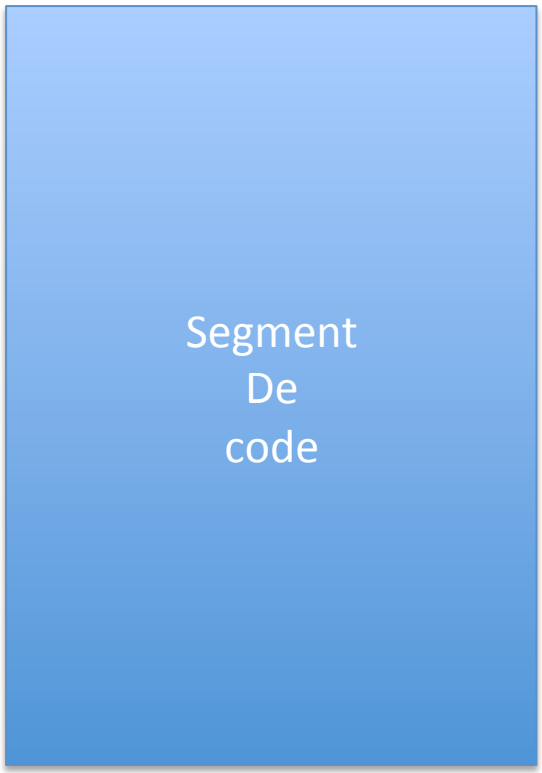
```
        .file   "essai.c"
        .version    "01.01"
gcc2_compiled:
        .section    .rodata
.LC0:
    .string "bonjour\n"
.text
    .align 16
.globl main
    .type   main,@function
main:
    pushl %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    subl    $12, %esp
    pushl   $.LC0
    call    printf
    addl    $16, %esp
    movl    %ebp, %esp
    popl    %ebp
    ret
.Lfe1:
    .size   main,.Lfe1-main
    .ident  "GCC: (GNU) 2.96 20000731 (Linux-Man
```

Bytecode pour machine virtuelle

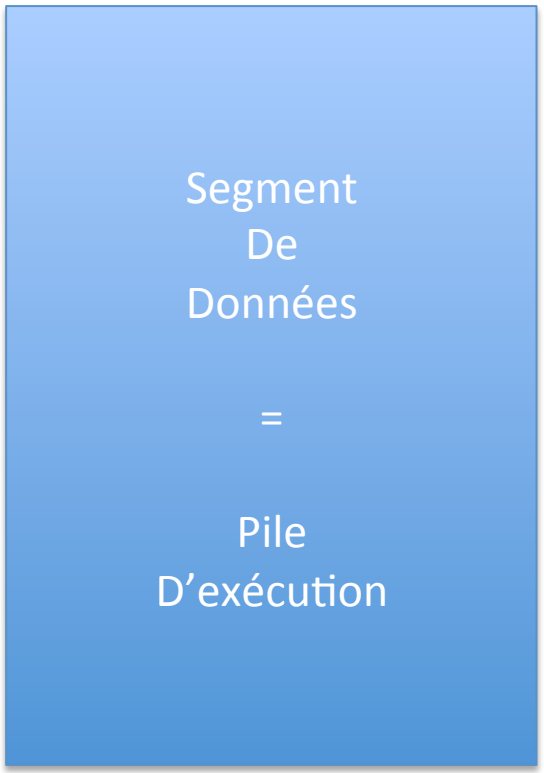
- Le bytecode est un ensemble d'instructions générées par le compilateur qui vont produire à l'exécution des mouvements de données sur une pile d'exécution



Machine virtuelle à pile (JVM)

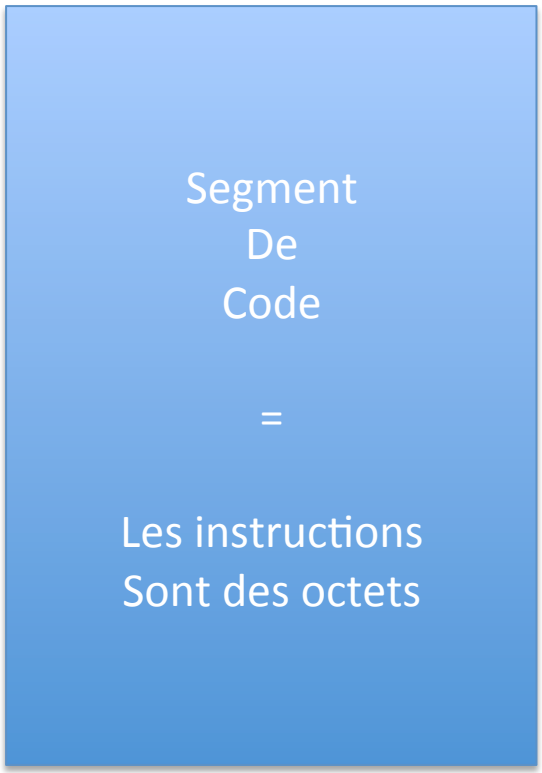


Contient les instructions



Contient les données

Machine virtuelle à pile (JVM)



0

Code

3 registres
- PC program counter
- SP stack pointer
- BP base pointer

Pile d'exécution, contient des
Nombres flottants (float)

Machine virtuelle EISE = EVM

- EVM contient plusieurs groupes d'instructions
  - Instructions arithmétiques et logiques
  - Instructions de manipulation de pile
  - Instructions de sauts, conditionnels et inconditionnels
  - Instructions d'entrée/sortie
  - Instructions spéciales

Boucle principale de l'interpréteur

```
int code[MAXCODE_SIZE];
float pile[MAXSTACK_SIZE];
int pc, sp, bp;

while code[pc]!=OP_HALT
{
        switch (code[pc])
        {
                case OP_ADD:
                        pile[sp-1]=pile[sp-1] + pile[sp] ; sp-- ; pc++ ; break ;
                case OP_SUB:
                        pile[sp-1]=pile[sp-1] - pile[sp] ; sp-- ; pc++ ; break ;
                …
        }
}
```

Arithmétique (pc++)

- ADD: pile[sp-1]=pile[sp-1]+pile[sp]
- SUB: pile[sp-1]=pile[sp-1]-pile[sp]
- MULT: pile[sp-1]=pile[sp-1]*pile[sp]
- DIV: pile[sp-1]=pile[sp-1]/pile[sp], erreur
- DIVI: pile[sp-1]=pile[sp-1]/pile[sp], en entier, erreur
- NEG: pile[sp-1]=-pile[sp-1]

Logique (pc++)

- AND: pile[sp-1]=pile[sp-1] and pile[sp]
- OR: pile[sp-1]=pile[sp-1] or pile[sp]
- XOR: pile[sp-1]=pile[sp-1] xor pile[sp]
- NOT: pile[sp-1]=not pile[sp-1]
- EQ: pile[sp-1]=1 si pile[sp-1]=pile[sp] sinon 0
- LS: pile[sp-1]=1 si pile[sp-1]<pile[sp] sinon 0
- GT: pile[sp-1]=1 si pile[sp-1]>pile[sp] sinon 0

Manipulation de pile (pc+=2)

- DEC op: sp = sp – op
- INC op: sp = sp + op
- PUSH op: pile[++sp] = op
- PUSHR op: pile[++sp]=atof(op) pc+=len(op)
- DUPL: duplication du sommet de pile
- LIBP op: