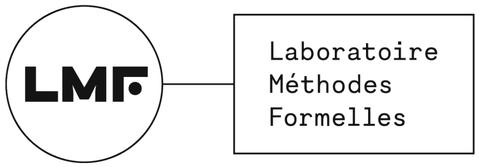


Introduction à la compilation

Polytech'Paris-Saclay– 4ème année –

Une introduction

Burkhart Wolff (& Frederic Voisin)

Contexte Historique (Rappel)

- Des le début de l'informatique après la guerre, la question de (parser, convertir et interpréter) de l'information était une éminence primordiale
- Trouver un moyen de communiquer de manière "humaine" avec les machines de plus en plus puissantes
  - Languages de programmation
    - 1943 : Le "Plankalkül" (Konrad Zuse)
    - 1943 : Le langage de programmation de l'ENIAC
    - 1954 : FORTRAN, le traducteur de formules (FORmula TRANslator), inventé par John Backus et al.
    - 1958 : LISP, spécialisé dans le traitement des listes (LISt Processor), inventé par John McCarthy et al.
    - 1959 : COBOL, spécialisé dans la programmation d'application de gestion (COmmon Business Oriented Language)
    - 1967 : Simula 67, inventé par Nygaard et Dahl comme surcouche d'Algol 60, est le premier langage conçu pour pouvoir intégrer la programmation orientée objet et la simulation par événements discrets.
    - 1969 - 1973 : C, un des premiers langages de programmation système, est développé par Dennis Ritchie et Ken Thompson pour le développement d'Unix aux laboratoires Bell.
    - 1975 : Smalltalk est l'un des premiers langages de programmation à disposer d'un environnement de développement intégré complètement graphique.
    - 1973 : ML (Meta Language) inventé par Robin Milner

Contexte Problématique

- nécessite des langages **adaptés** à des domaines spécifiques
- communiquer de manière “humaine” et pour être
  - “portable” (pre-requisite: abstraction de machines…,)
  - “productive” (pre-requisite: code reutilisable)
- traduire fidèlement des algorithmes complexes
  (pre-requisite: définir “la sémantique d’un language”)
- faciliter la vérification en test et preuve

Concepts des Languages

- Pour répondre a ces défis, un ensemble de concepts a été développé au fil des années:
  - structurer l'algorithmique d'un problème:

```
while B do
   if C then …
         else
           for i=1 to n
              {    …     }
```

nested control flow

```
while B do
    if C then
        … ;break
    else
        … ;return(D)
```

structured exits

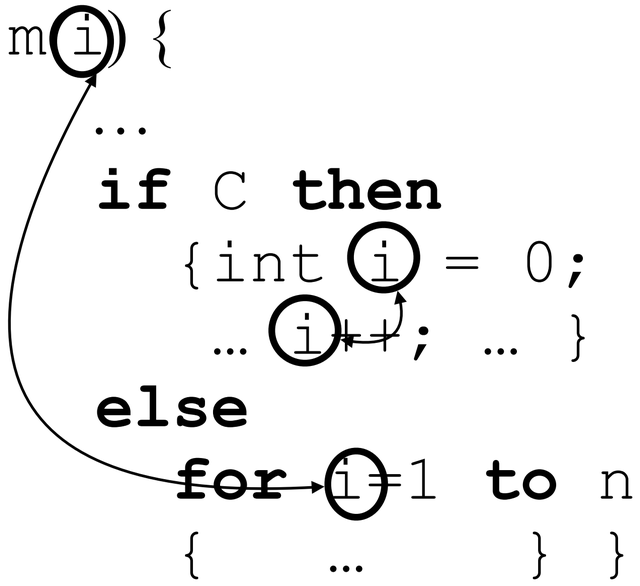
```
method M() {
    … raise E
}

method N{}
  { … ;
    M{};
    catch E do {}
  }
```

exception handling

Concepts des Languages

- Pour répondre à ces défis, un ensemble de concepts a été développé au fil des années:
  - structurer la visibilité des vars, consts et fonctions



nested scopes

```
structure M : sig S :
  begin
     local open M′ in
     fun m () …
     end
  end
…
M.m
```

modules and name spaces

```
use gmplib;
use commonlib;

… sqrt(3.0 + 5)…
```

name spaces and overloading

Concepts des Languages

- Pour répondre a ces défis, un ensemble de concepts a été développé au fil des années:
  - typage de donnés et d'algorithmes:

```
{int a;
 int b[1..10];

 … a = b[4];
}
```

basic data

```
def map::(α⇒β)
          ⇒ α list
          ⇒ β list

… map(λx.[x])[1,2,3]…
```

higher-order polymorphism

```
def insert:: α EQ :
   α ⇒ α list
where
  ins a = [a]
|ins a (b#S)=
      if a <= b
      then a#b#S
      else b#ins(a#S)
```

class polymorphism

Concepts des Languages

- Pour répondre à ces défis, un ensemble de concepts
  a été développé au fil des années:
  - Surcharge d'operateurs
  - structurer l'execution en **concurrence** (locks, events, threads, processes, …)
  - controler l'acces au **memoire non-standard** (volatile vars, …)
  - l**iaison statique ou dynamique** des appels de fonctions
    pour refléter les mécanismes d'héritage
  - **modules parametrés**, generic classes, …
  - compilation separée, liaison dynamique, …
  - just-in-time compilation, interpretation, …

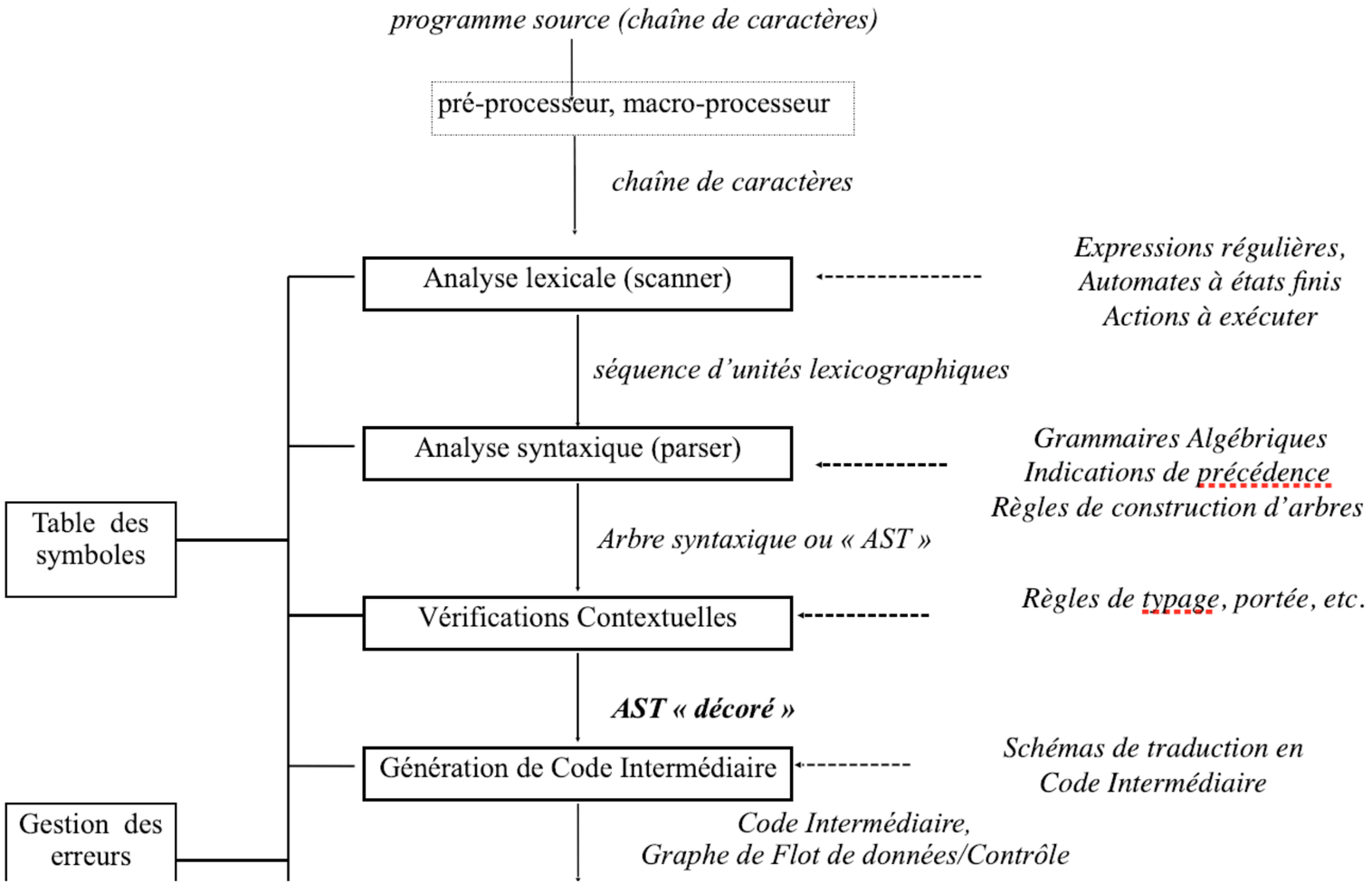
Phases d’un compilateur

- En gros, on a deux phases a distinguer:
  - phase d’analyse et de reconnaissance du programme source
    - analyse lexique (transformation dans une sequence de “mots” (lexemes))
    - analyse syntaxique (structures grammaticales)
    - analyse sémantique (identification de la signification des operators ou appels)
  - phase de synthèse
    - calculé à partir de la représentation intermédiaire le programme cible
    - … conforme a une description de la machine cible
  - ces phases sont elle-même découpée en sous-phases, utilisant des syntaxes abstraites et techniques adaptées et parfois différentes …

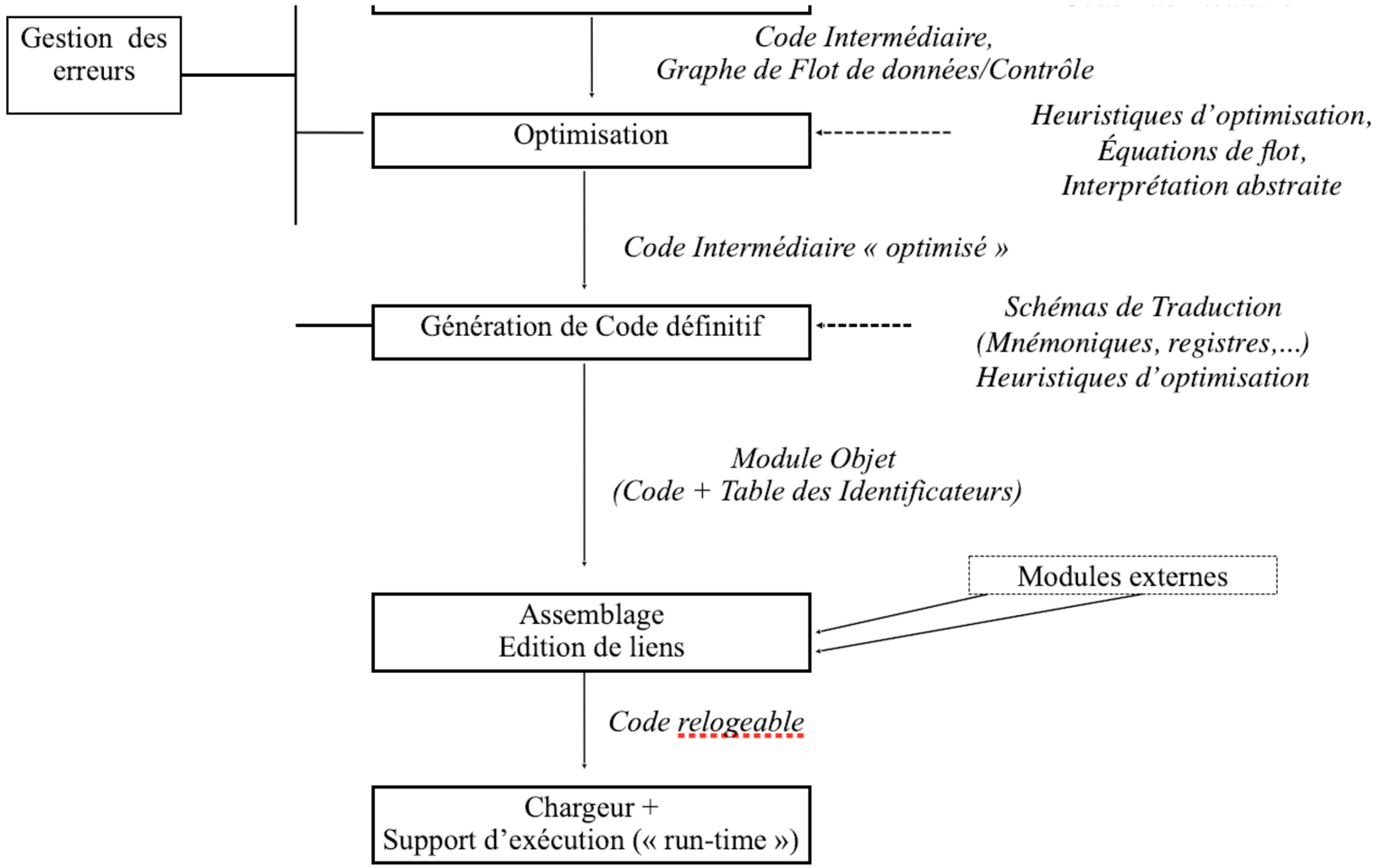
Phases d’un compilateur

- En gros, on a d(eux phases a distinguer:
  - phase d’analyse et de reconnaissance du programme source
    - analyse lexique (transformation dans une sequence de “mots” (lexemes))
    - analyse syntaxique (structures grammaticales)
    - analyse semantique (identification de la signification des operators ou appels)
  - phase de synthèse
    - calculé à partir de la représentation intermédiaire le programme cible
    - … conforme a une description de la machine cible
  - ces phases sont elle-même découpée en sous-phases, utilisant des AST différentes et techniques adaptées …
    - AST décoré, analyse abstraite, execution symbolique, peephole-optimizations,

Phases d'un compilateur



Phases d’un compilateur



Analyse Lexique

- analyse lexique : angl. Scanner, Lexer
- le choix du découpage en unités lexicographiques
  est propre à chaque langage … mais on identifie typiquement:
  - les mots-clefs,
  - les constantes littérales (constantes numériques, chaînes de caractères, etc.),
  - les opérateurs, les symboles (comme `:=` ou `;` ou les parenthèses).
- … et produit des lexemes typiquement comme un triple (**class**, **string**, **pos**)
  - ou **class** est par exemple `ident`, `str`, `numeral`, `relop`, `monop`, `key`, …
  - et **string** est `“i”`, `“m”`, `“ins”` , `“<“`, `“<=“`, `“begin“`, `“if“`, `“while“`
  - et **pos** est une information positionelle (ex.: fichier, ligne, colonne)

Analyse Lexique

- grammaire typiquement de typ 3
- décrite à l'aide d'***expressions rationnelles***, éventuellement augmentées de mécanismes d'abréviations.
- exemple: format d'un identificateur

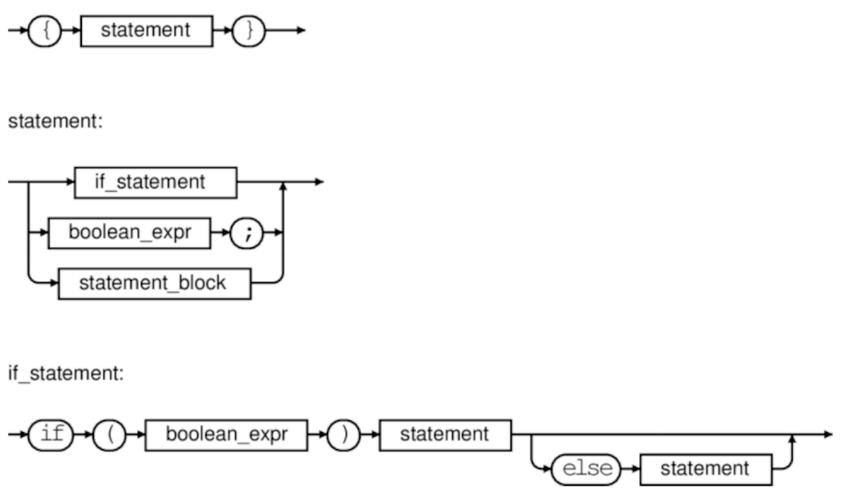
```
lettre   =  [a-zA-Z_]
chiffre  =  [0-9]
ident    =  lettre . (lettre | chiffre)*
```

- dans certains cas, il faut aussi stocker les commentaires
  dans les lexemes (pragmas, antiquotations, …)
- erreurs: typiquement des pb. de codages de caractères (UTF8 , etc)
- generateurs d'analyseurs lexiques: lex, smllex, …

Analyse Syntaxique

- grammaire typiquement de typ 2
- décrite à l'aide des grammaires, evtlmt. présentes sous forme de "railroad diagrams"
- exemple: expressions avec parenthèses

```
E::= E + E
E::= E - E
E::= E * E
E::= E / E
E::= ident
E::= cste
E::= ( E )
```



- Output: arbre syntaxique, typiquement sous forme d
- erreurs: typiquement des pb. structurelles de syntaxe
- generateurs d'analyseurs syntaxiques: Flex, Yacc, Menhir, …

Analyse Contextuelle

- grammaire typiquement de typ 1,
- souvent décrit par des AST décorés
  par attributs (“syntax directed translation”)
- cela couvre:
  - resolution des liaisons,
    - calcul des environnements,
    - analyse des types,
    - construction AST intermédiaires
    - construction du code pour une machine virtuelle