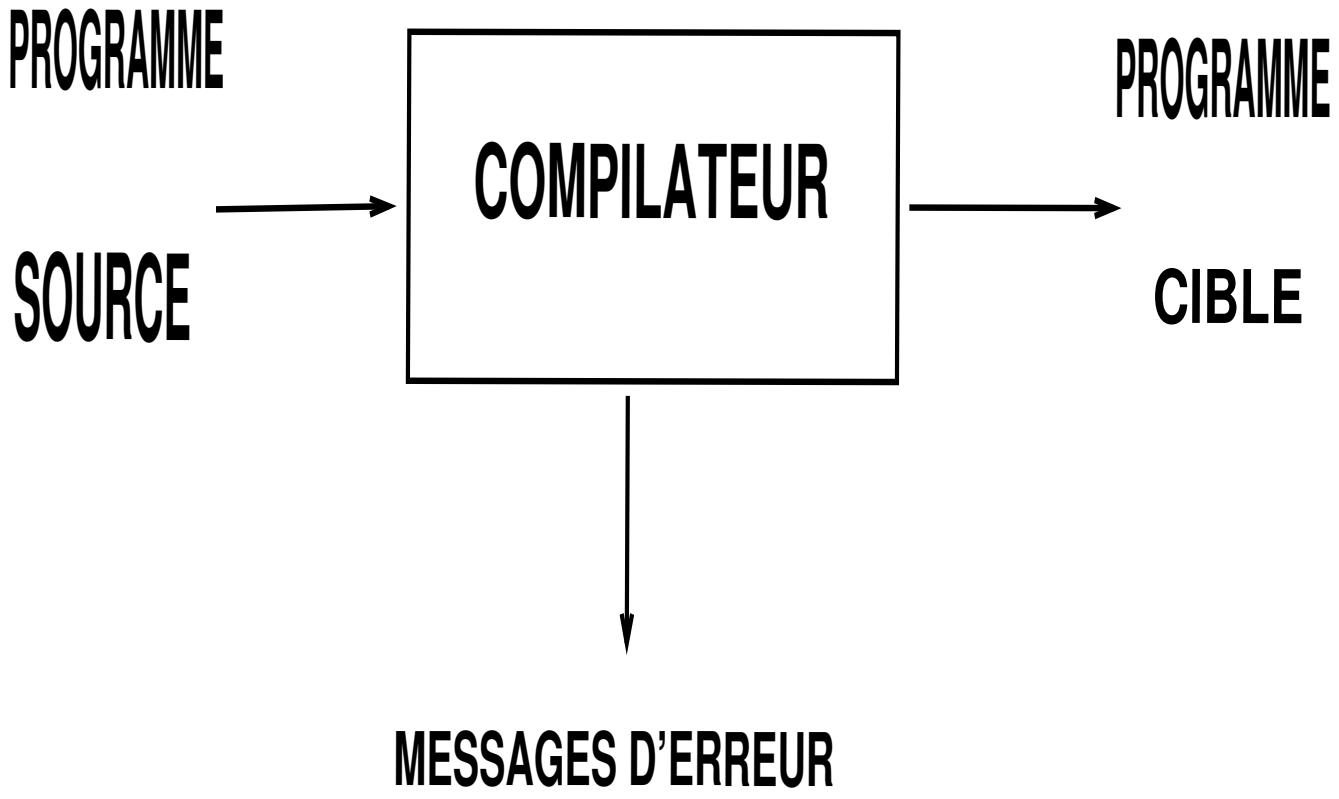
Compilateur ?



Différents compilateurs

- une seule passe
- plusieurs passes
- optimiseurs

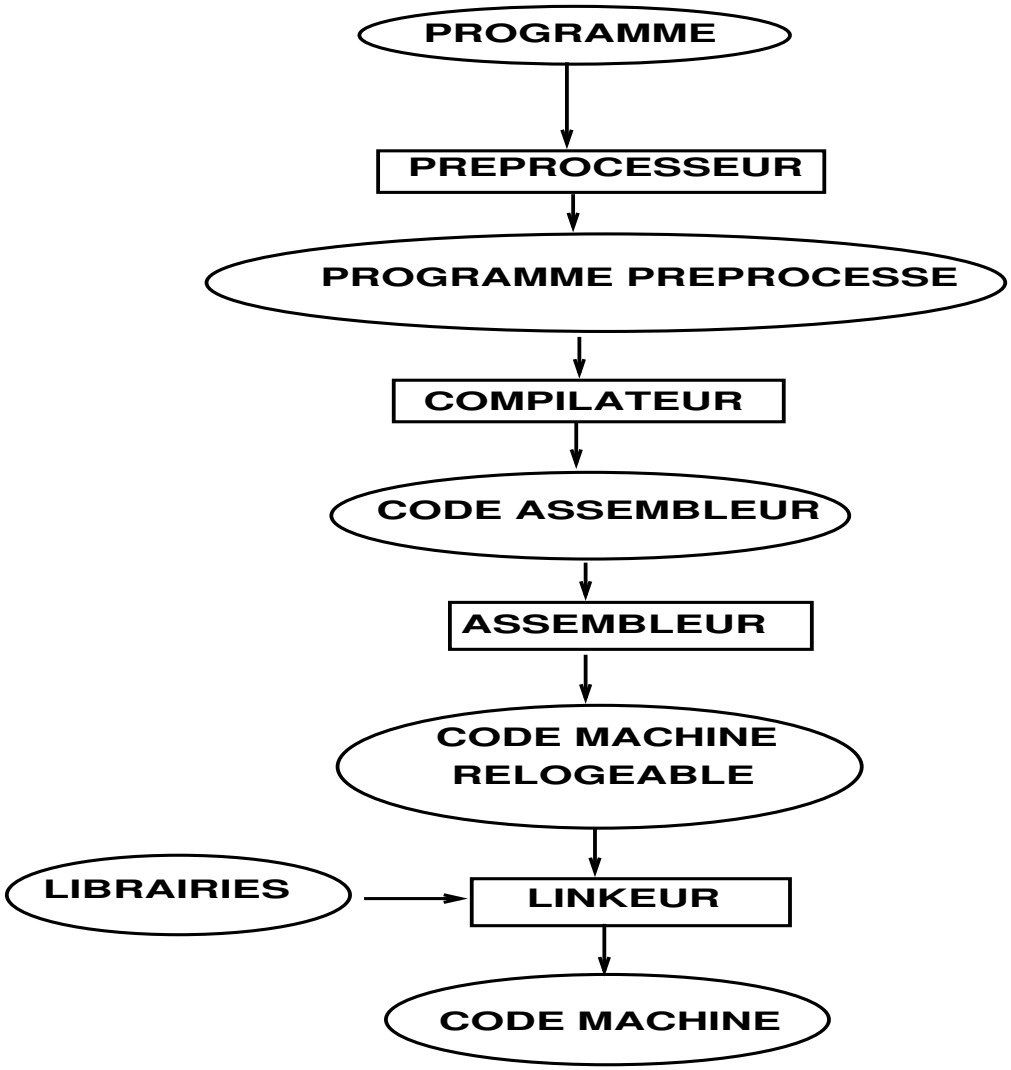
1957 - Le premier compilateur ForTran. 18 Homme-années ....

Maintenant - premières briques de base en 1 semestre

Voisins des compilateurs

- Editeur de structure
- Pretty-Printer
- Interpréteur
- déboggueur
- Traitement de texte (Latex)
- Interpréteur de requêtes base de données (SQL)

Environnement de compilation



Préprocesseur

- compilateur source à source
- pas de compréhension du langage
- collecter les différents modules
- effectuer les inclusions des fichiers
- remplacement des macros, `define`,..
- extensions du langage

exemple: CPP, M4, RATFOR

```
cc -P x.c   -->  x.i
```

Assembleur

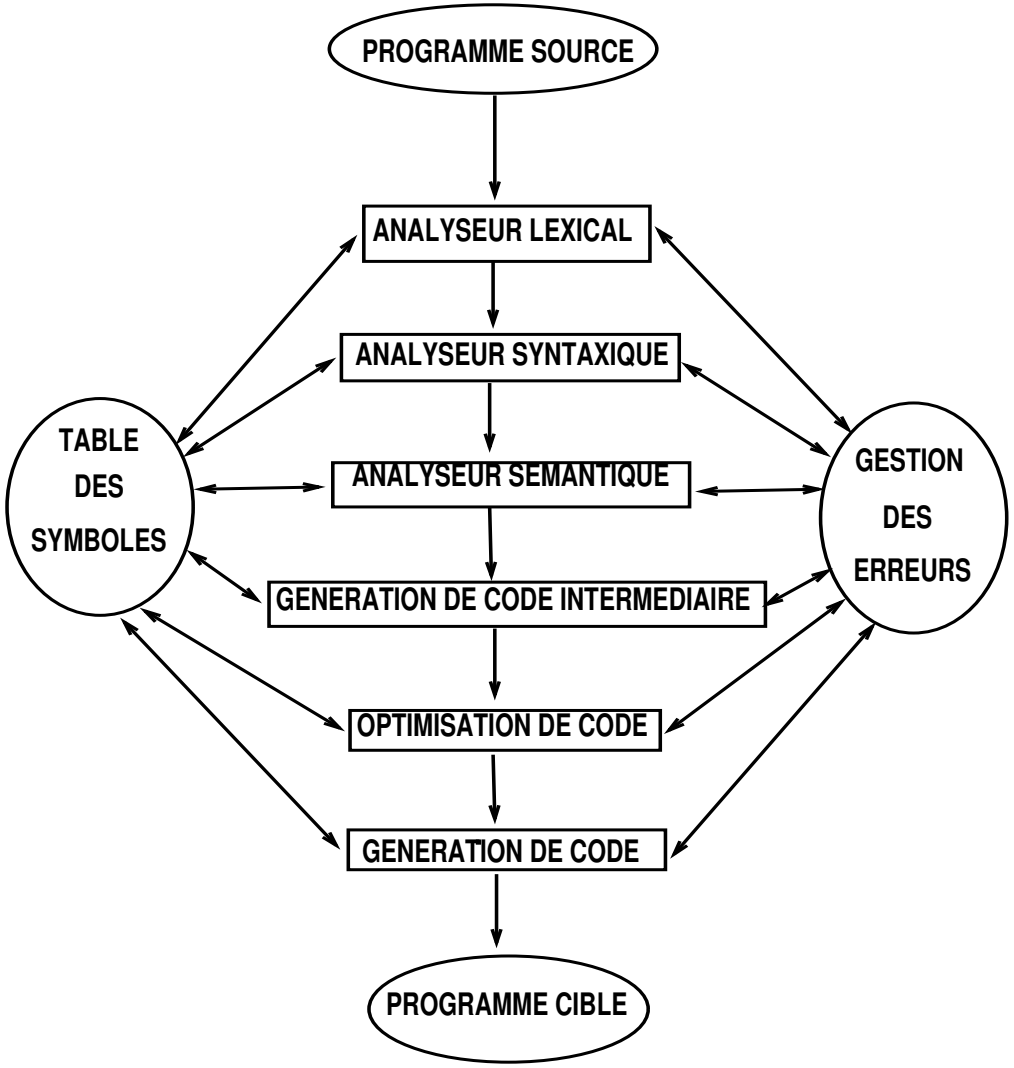
- le code assembleur est une version mnémonique du code machine
- une instruction représente le code binaire d'un opérateur
- les champs représentent les adresses mémoire des opérandes

```
MOV i3, R2
MUL #60, R2
MOV i2, R1
ADD R2, R1
MOV R1, i1
```

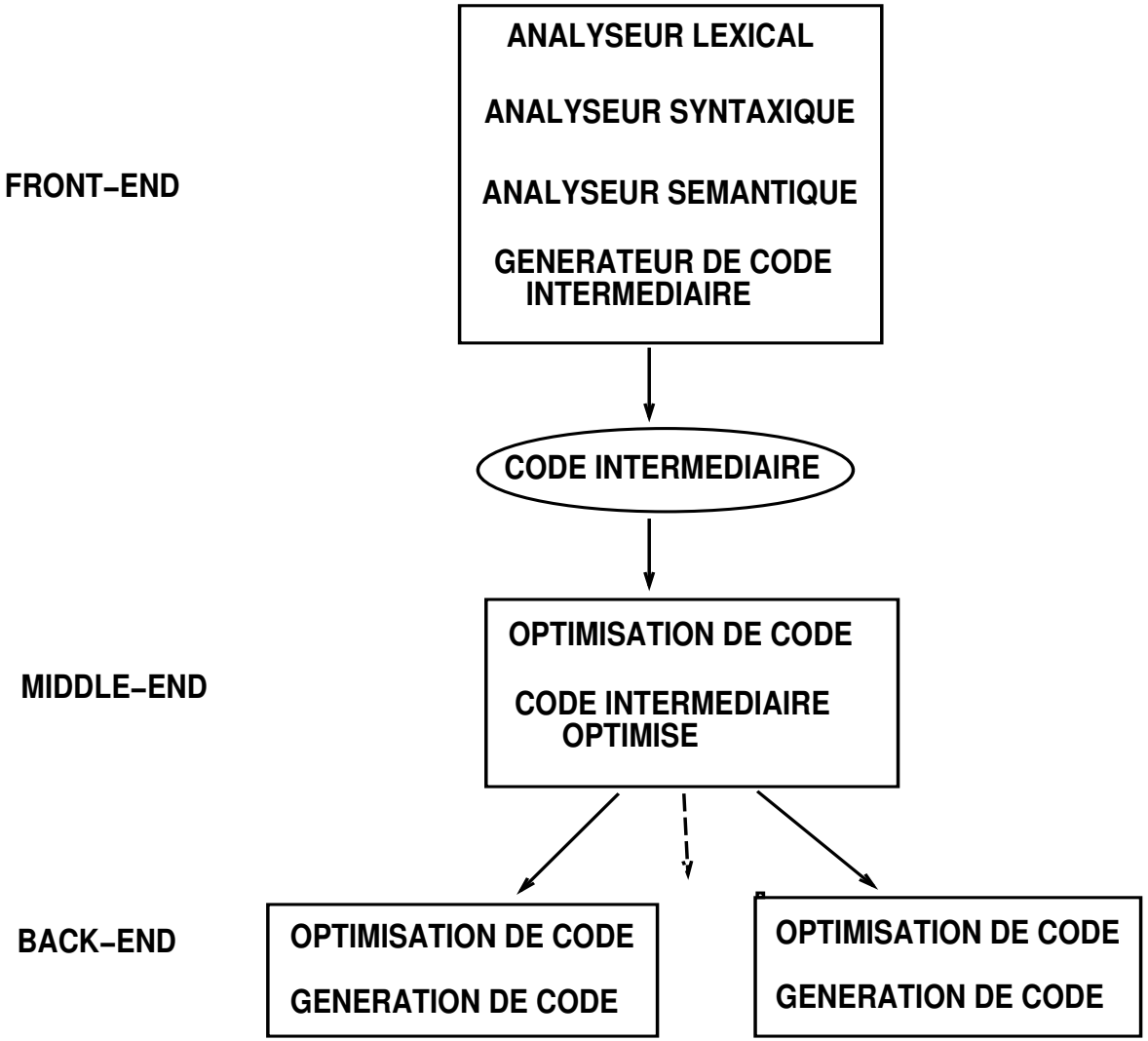
Linkeur

- effectue le lien avec les librairies
- association des codes exécutables
- mise à jour des adresses relogeables
- allocation mémoire des codes

Structure d’un compilateur



Phases du compilateur



Analyse du programme

- identification des mots du langage: *tokens*
- regroupement hiérarchique des *tokens*
  - reconnaissance de phrases grammaticales
- vérification sémantique

Analyse lexicale

- précède l'analyse syntaxique
- associe à chaque entrée un *token*: unité d'analyse lexicale
- reconnaissance des identificateurs et mots clés
  - mots clés contenus dans la table des symbole ou dans une liste spécifique
  - *blanc* caractère spécial

Analyse Syntaxique

- phrases grammaticales
- grammaire hors-contexte
- règles récursives
  - expression : expression opération expression ;
  - expression : nombre ;
  - expression : identificateur ;
- représentation par arbre syntaxique

Analyse Sémantique

- vérification du programme
  - contrôle des types
  - contrôle du flot d'exécution
  - contrôle des déclarations

Code intermédiaire

- programmation d'une machine abstraite (byte code JAVA)
- nombreuses techniques classiques d'optimisation de code intermédiaire
- optimisation de code pour la machine cible
- indépendance des passes
- exemple : code 3 adresses

Optimisations de code

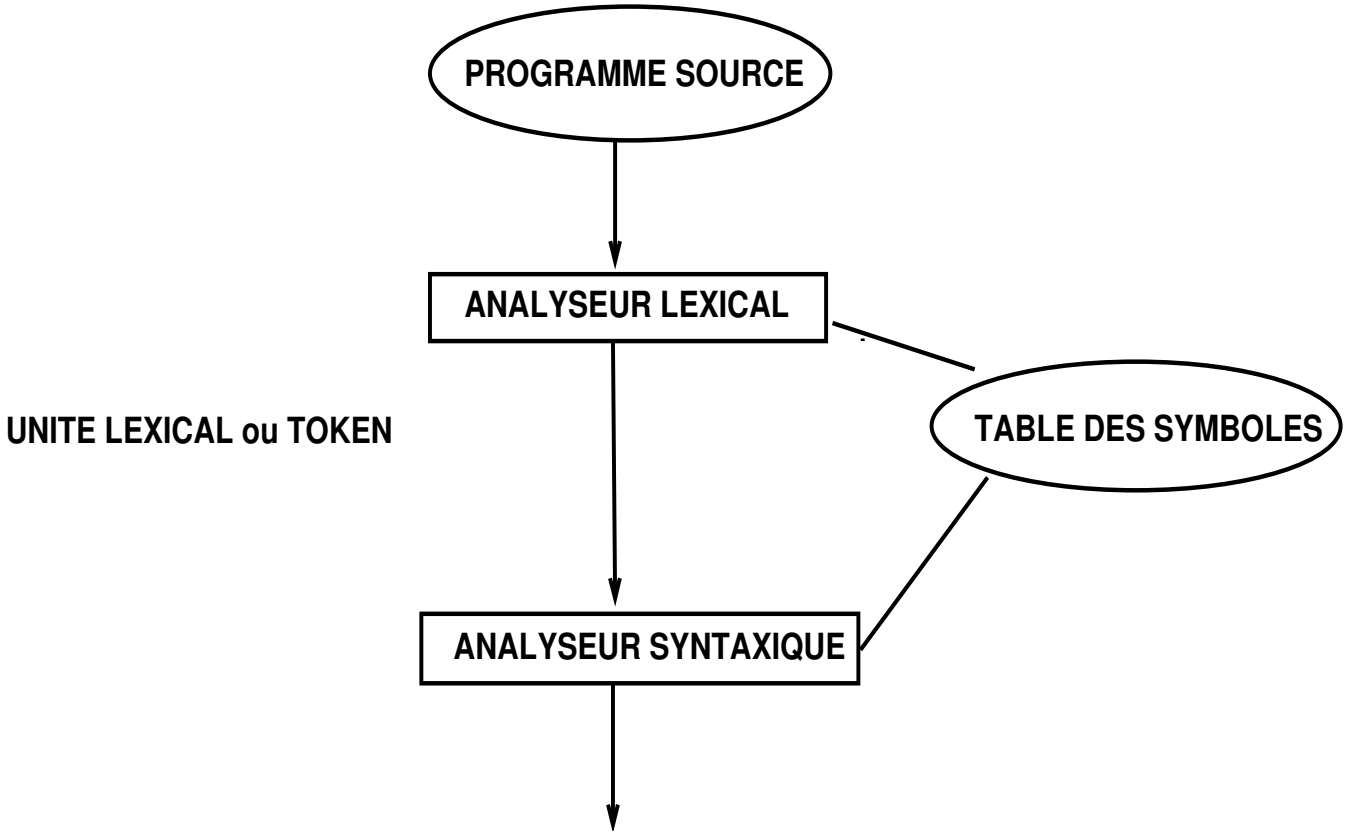
- Améliorer le code intermédiaire
- nombreux critères d'optimisation
  - minimiser l'espace du code compilé
  - augmenter la vitesse d'exécution

```
cc -O3 ...
f77 -qhot -O3 ...
f90 -O 3 -O unroll2 ...
```

Génération de code

- Traduction du code intermédiaire en assembleur ou code machine translatable
- Dépendante de la machine cible et de ses instructions

Analyseur Lexical



Analyse lexicale

- précède l'analyse syntaxique
- associe à chaque entrée un *token*: unité d'analyse lexicale
- reconnaissance des identificateurs et mots clés
  - mots clés contenus dans la table des symbole ou dans une liste spécifique
  - *blanc* caractère spécial

Autres fonctions de l'analyseur lexical

- résolution des ambiguïtés lexicales ?
  - Fortran, espaces non significatifs :

    boucle sur I : `DO 5 I =1,25`

    affectation à une variable `DO 5 I =1.25`
  - chaîne Hollerith de Fortran : $n\mathtt{H}a_1a_2...a_n$
  - PL1, mots clefs non réservés

    ```
    if then then then =else; else else =then;
    ```
- affichage de messages d'erreurs et numéros de ligne

Expressions régulières

- simple
- concise
- facile à comprendre
- efficace à implanter

Analyse Syntaxique

- reconnaissance de phrases grammaticales
- représentation par arbre syntaxique
- règles récursives
  - blocs
  - bloc : instructions ;
  - instruction : expressions ;

Utilisation d'une grammaire

- spécification simple du langage de programmation

  (notation Backus-Naur Form)

- permet la construction parfois automatique d'un parseur (Yacc)

- évolution du langage plus facile lors des ajouts basés sur la description grammaticale

- Grammaire LL : + facile à implanter manuellement

- Grammaire LR : existence d'outils automatiques

  + grande classe de grammaires

Grammaire Hors-contexte

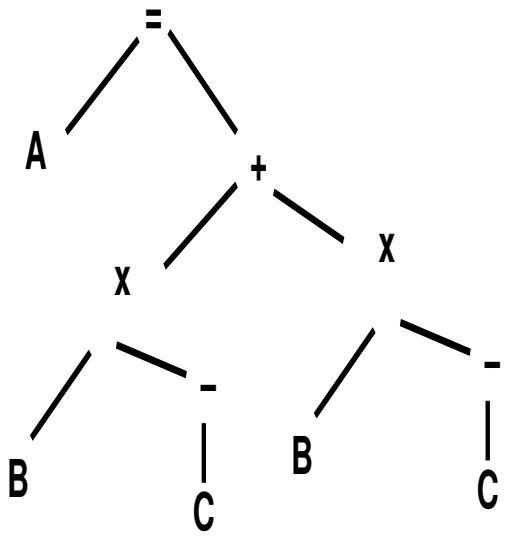
- ensemble de *tokens* = symboles terminaux
- ensemble de non terminaux (donne la hiérarchie)
- ensemble de règles de production
  - non terminal : sequence de *tokens* et non terminaux
- non terminal au début pour marquer le point de départ

Arbre syntaxique

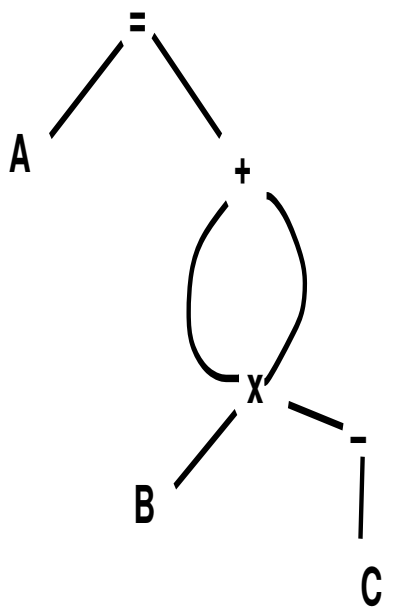
- dérivation à partir du symbole de début des phrases du langage
- racine est labellée par le symbole de début
- chaque feuille est labellée par un *token* ou ϵ
- noeud interne est un non terminal
- est une règle de production

Représentations Graphiques

A = B x - C + B x - C



Arbre abstrait



DAG

Grammaires

list ⟶ list + digit

list ⟶ list – digit

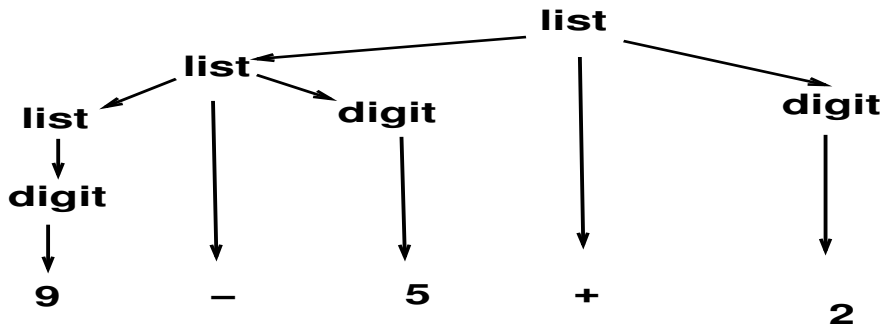
list ⟶ digit

digit ⟶ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

list ⟶ list + digit | list – digit | digit

digit ⟶ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

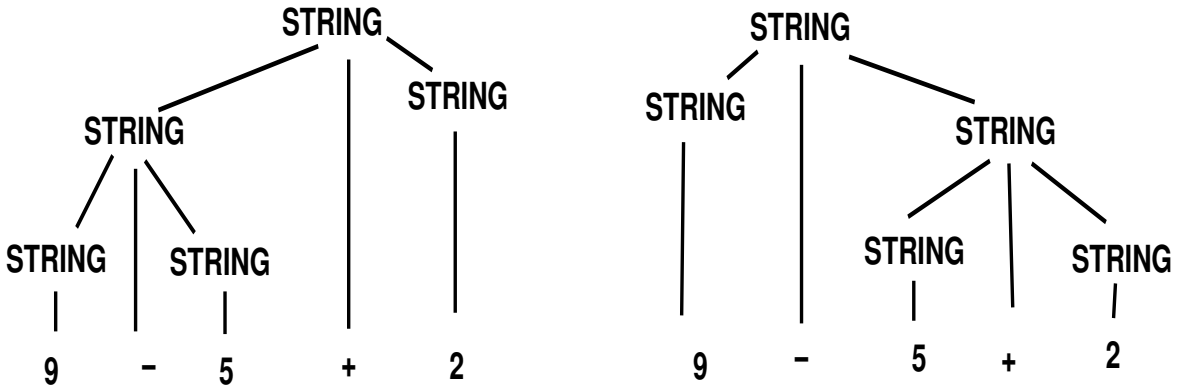
symbols : + – 0 1 2 3 4 5 6 7 8 9



Grammaire Ambiguë

STRING --> STRING + STRING | STRING – STRING | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

ARBRES SYNTAXIQUES



Grammaire Ambiguë

- Plusieurs *arbres syntaxiques* pour une phrase
- Difficultés de choisir une règle de production
- Solutions ?
  - Propriétés des opérateurs
  - Ajout de contexte
  - Modifications des règles de production
  - Grammaire LL ou LR

Propriétés des opérateurs

- ils sont associatifs à gauche par convention
- `+ - / *` sont associatifs à gauche
- `= ^` est associatif à droite
- `*` a une plus forte priorité que `+`

```
expr : expr + term | expr - term | term ;
term : term * factor | term / factor | factor ;
factor : digit | (expr) ;
```

Utilisation d'une grammaire - restriction

- L1 = { $a^n b^m c^n d^m$ }

  nombre de paramètres formels égal nombre de réels

- L2 = { wcw , w = (a | b) * }

Traduction dirigée par la syntaxe

- traduction d'une construction du langage
- utilise une grammaire hors-contexte
- à chaque symbole de la règle de production, on peut associer une règle sémantique
- à chaque symbole grammatical est associé des attributs
  - attributs *synthétiques* calculés à partir des attributs des *fils*
  - attributs *hérités* calculés à partir des attributs des *parents*
- évaluation en profondeur d'abord

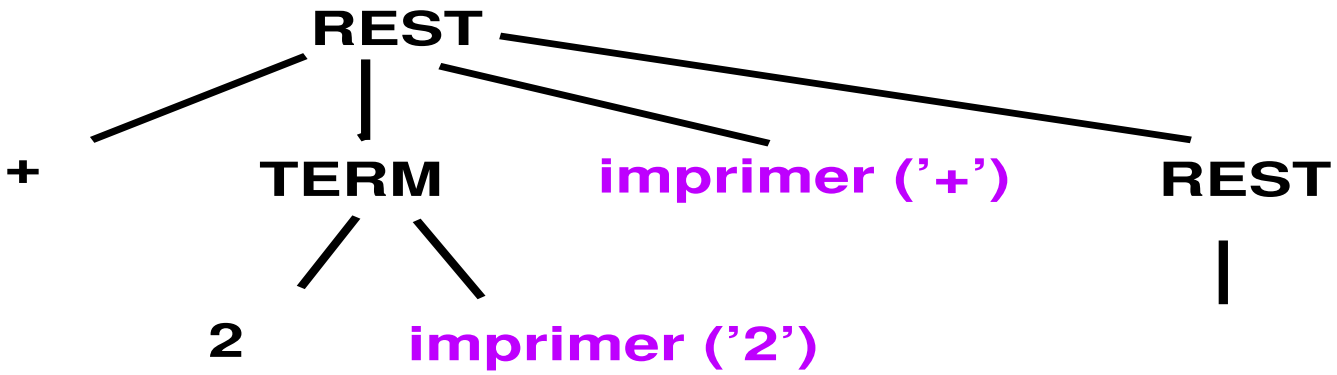
Tables des symboles

- contient des informations sur les diverses construction du langage
- enregistre les identificateurs et leurs caractéristiques
  - types: string, caractère, entier
  - identificateur + types : procédures, labels, variables
  - constantes
  - portée des déclarations

Règles et Actions sémantiques

REST : + TERM imprimer (’+’) REST

TERM : 2 imprimer (’2’)



Approches Top-Down

- Sélectionner une règle de production, construction des fils
- Sélectionner le prochain symbole pour lequel on doit construire l'arbre
- Backtracking quand erreur

Analyse syntaxique prédictive

- le symbole de prévision détermine de manière non ambiguë la procédure à choisir pour chaque non terminal
- descente récursive
- 1) prédiction à l'aide de mécanisme qui permettent de choisir une règle plutôt qu'une autre
  - si conflit `->` erreur
  - ϵ est utilisé quand aucune règle ne peut être utilisée
- 2) la procédure utilise une règle pour la partie droite

bouclage si récursion à gauche `expr : expr - term`

Elimination de la récursion à gauche immédiate

- `A : A x | y`

  doit être transformé en

- `A : y R`
- `R : x R |`ϵ

`expr: expr - term | term` Problème lorsque récursion à gauche par dérivation

- `S : A a | b`
- `A : A c | S d |`ϵ

Factorisation à gauche

- `A : X b1 | X b2 | c`

- `A : X R | c`
- `R : b1 | b2`

Grammaire LL(k)

- adaptée à l'analyse descendante
- *Left to right*
- *Left-most derivation*
- `k` entrées *visibles*
- grammaire LL(1) $->$ ni ambiguë, ni récursive à gauche
- oter la récursion à gauche et factoriser dégrade la lisibilité de la grammaire
- implantation manuelle plus facile

Grammaire LR(k)

- adaptée à l'analyse ascendante
- *Left to right*, *Right-most derivation*
- `k` entrées *visibles*
- utilisée par un grand nombre de constructeurs automatiques d'analyseurs syntaxiques (Yacc, Bison)
- méthode la plus générale d'analyse syntaxique par décalage-réduction sans backtracking
- classe des grammaires LR(k) est un sur-ensemble de celle des grammaires traitées par analyseur prédictif

Tables d'analyse SLR, LR canonique, LALR

- SLR simple mais restriction sur les grammaires
- LR canonique, les plus complètes
- Look-Ahead LR, tables plus petites (rapport 10)

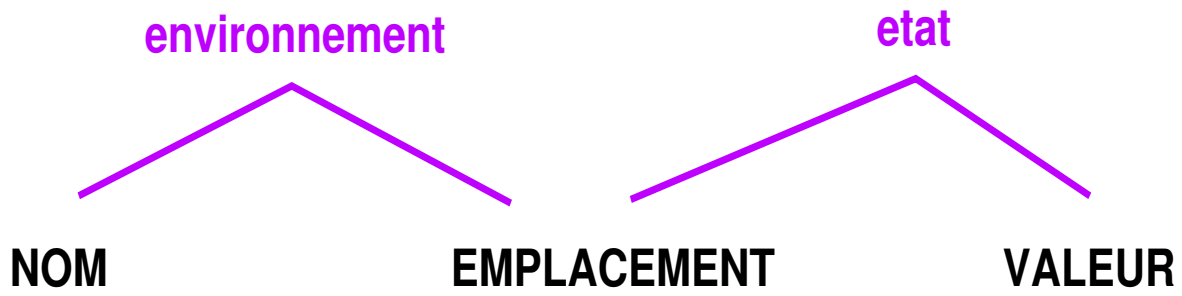
Comparaison LL et LR

- LR reconnaître l'occurrence de la partie droite d'une production à partir de la partie droite dérivée et les k symboles en entrée
- LL reconnaître l'usage de la règle de production à partir des k symboles en entrée
- LR permet de d'écrire une classe plus large de grammaires

Environnements d'exécution

- procédure
- déclaration
- organisation mémoire
- système d'exploitation

Les noms



X = A

Y = X

& X = Z

Déclarations

- explicite en Pascal
- implicite en Fortran
- portée de la déclaration est variable
  - variable globale
  - variable locale

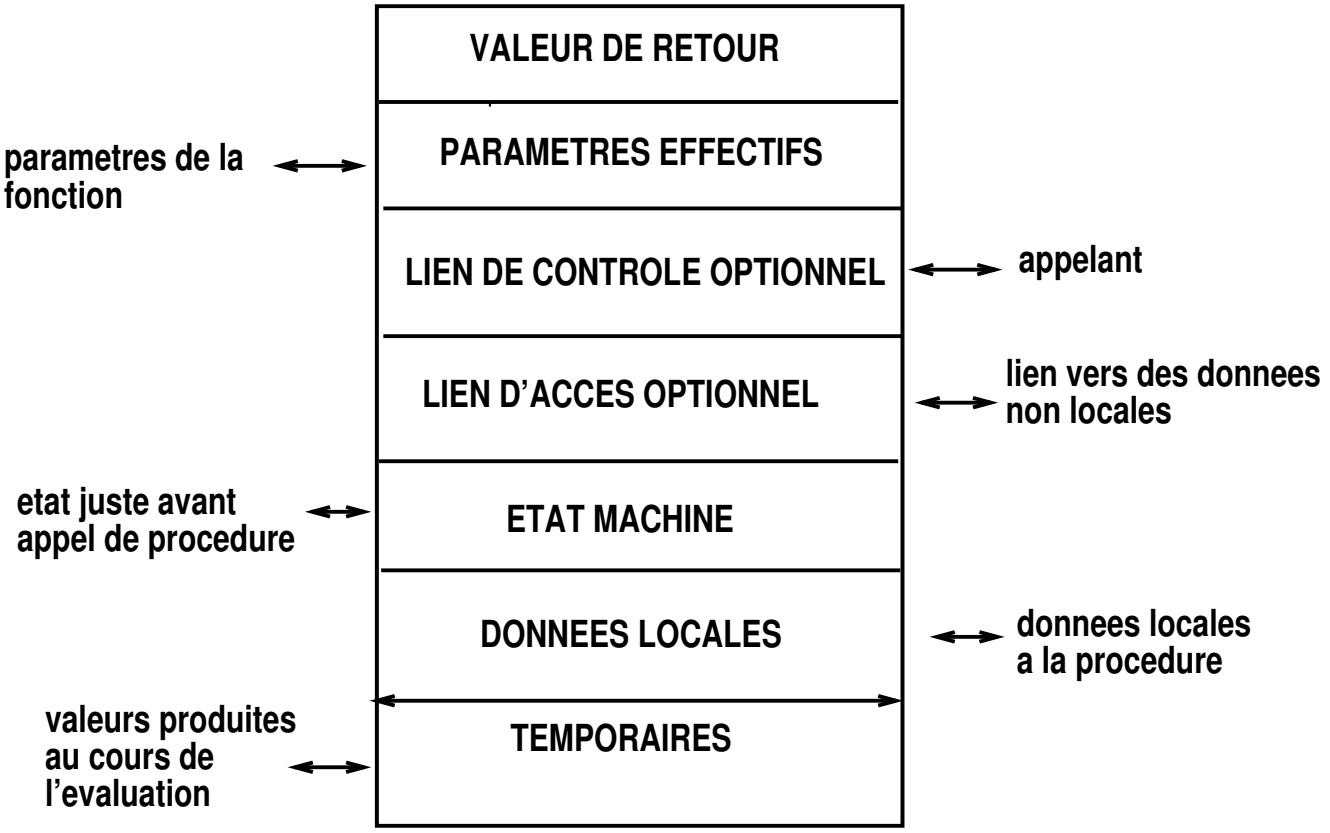
Organisation de la mémoire

- subdivision de la mémoire pour stocker
  - code généré - instructions
  - données statiques
  - pile de contrôle
  - tas

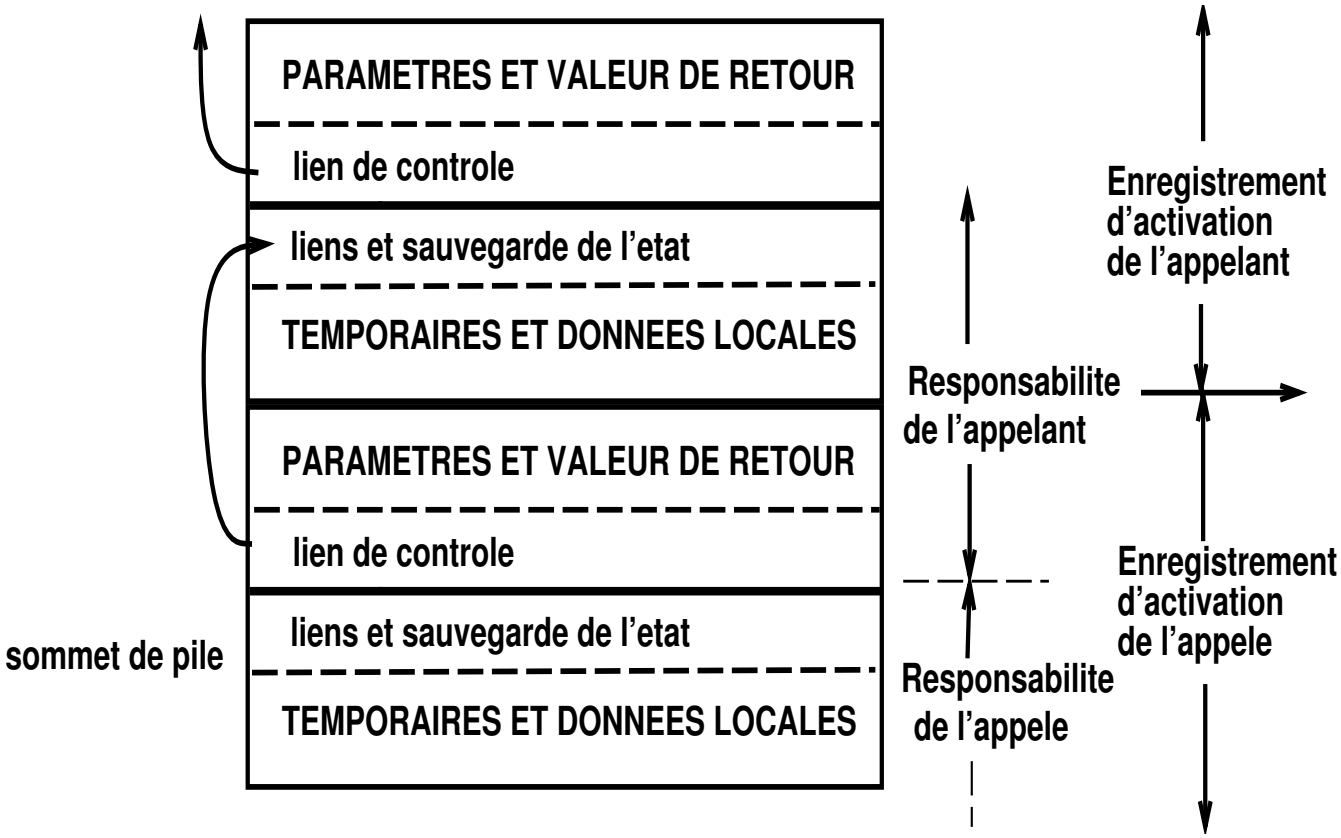
Procédures

- paramètres formels dans la définition
- paramètres réels passés en arguments (par référence/valeur)
- durée de vie d'une procédure
- procédure récursive/non récursive
- pile de contrôle garde trace des activations des procédures

Enregistrement d'activation



Appel de procédure



Passage par paramètres

- passage par valeur
- passage par référence
- passage par nom

Passage par valeur

- C, Pascal
- l'appelant évalue les paramètres réels et les place à l'adresse des paramètres formels

Passage par copie-restauration

- hybride valeur et référence
- les paramètres sont évalués et leur valeur passée
- les adresses sont déterminées avant l'appel

Passage par nom

- procédure vue comme une macro-définition
- correspond à une expansion de procédure
- pb: i = a[i] ; a[i] = t ; `-> a[a[i]] = t`

Génération de code intermédiaire

- utile pour différents front-end et/ou back-end, optimisations
- langage intermédiaire:
  - arbre syntaxique (structure hiérarchique)
  - notation postfixée (représentation linéaire)
  - code 3 adresses : `temp = x op y`

Génération de code - Problèmes

- “Générer un code optimal pour une architecture” est un problème indécidable
- Donnée du générateur de code
- Programmes cibles
- Gestion de mémoire (conversion de type, allocation)
- Sélection des instructions
- Allocation des registres

Donnée du générateur de code

- représentation linéaire (notation postfixée)
- 3 adresses (quadruples, triplets,..)
- représentation machine virtuelle (code machine à pile)
- représentation graphique (arbre syntaxique, dag)

Langages cibles

- langage machine absolu (statique et exécutable)
- langage machine translatable (compilation séparée)
- langage assembleur

Sélection des instructions

- la qualité du code dépend de sa taille et de sa vitesse d'exécution
- Coût des instructions
-
```
MOV b,R0
ADD c,R0
MOV R0,a
MOV a, R0
ADD e,R0
MOV R0,d
```

```
a = b+c
d = a +e
```

Transformation de code

- Elimination des sous-expressions communes
- Elimination du code inutile
- Renommage des variables temporaires
- Echange d'instructions indépendantes

Elimination des sous-expressions communes

```
a = b + c
b = a - d
c = b + c
d = a - d
```

```
a = b + c
b = a - d
c = b + c
d = b
```

Transformations algébriques

- `x = x + 0`
- `x = x * 1`
- `y = x ^2 = x * x`

Allocation des registres

- les registre ont un accès mémoire très rapide
- le nombre de registres est toujours très limité
- “trouver une affectation optimale des registres aux variables” est un problème NP-complet
- autres critères :
  - registres particuliers
  - conventions de noms

Allocation des registres

- conserver les variables les plus utilisées
  - variables internes aux nids de boucles
  - fixer le nombre de registres dédiés à cette tâche
- gain obtenu s'il une valeur reste dans un registre
  - $\sum_{Blocs\ \ B}\ \ Use(x,B) + 2 * Live(x,B)$
- algo. de coloration graphe

Optimisation du code

- Graphe de flot :
  - chaque noeud correspond au DAG d'un bloc de base
  - chaque arc représente le contrôle
- Optimisation des blocs de base
- Utilisation des DAGs
  - détection des sous-expressions communes

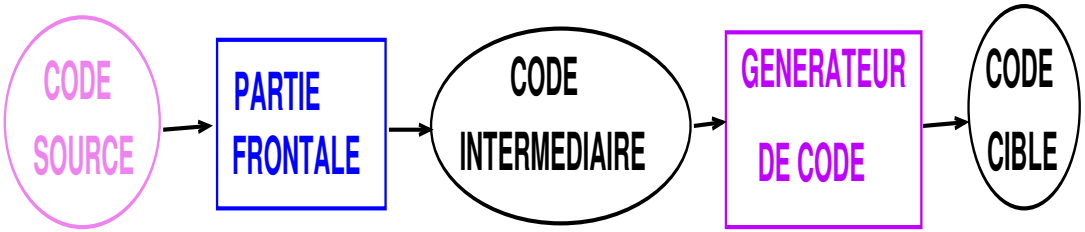
Optimisation à la lucarne

- élimination d'instruction redondante
- optimisation de flot de contrôle
  - `si  x > y goto L1 ; L1: goto L2`
- simplifications algébriques
- utilisation de code spécifique à la machine
- éliminer les `LOAD` et `STORE` redondants
- éliminer le code inutile
  - `if debug = 0 then ... else ..`

Optimisations à plusieurs niveaux

L'utilisateur

- peut profiler le programme
- changer l'algorithme
- transformer les boucles



le compilateur peut

- ameliorer les boucles,

les calculs d'adresses

Le compilateur peut

- utiliser les registres
- selecionner des instructions,
- faire des transformations locales.

Optimisation de code

- produire une code aussi bon qu'à la main ?
- évaluer la série de transformations
- doit préserver le comportement du programme (transformations légales)
- critères d'optimisation
  - minimiser l'espace du code compilé
  - augmenter la vitesse d'exécution
- ne pas dégrader les temps de compilation pour quelques optimisations particulières

Les principales optimisations de code

- détection des sous-expressions communes
- propagation de code
  - après affectation, réutilisation de la variable le plus possible $->$ permet d'éliminer des affectations
- Elimination de code mort
  - `if debug = 0 then ... else ...`
  - affectation inutile (boucle)

- optimisation de boucles
  - déplacement de code vers l'extérieur (réutilisation)
  - inversion de boucles
- élimination des variables d'induction
- optimisation des blocs de base
  - simplifications algébriques
  - évaluation de constantes

Références bibliographiques

- Compilateurs - Principes, techniques et outils

  Alfred Aho, Ravi Sethi, Jeffrey Ullman

- Flex - A fast scanner generator

  Vern Paxson

- Bison - The YACC compatible Parser Generator

  Charles Donnelly, Richard Stallman