Compilateurs et interpréteurs
L3 informatique

Étienne Payet

Département de mathématiques et d'informatique

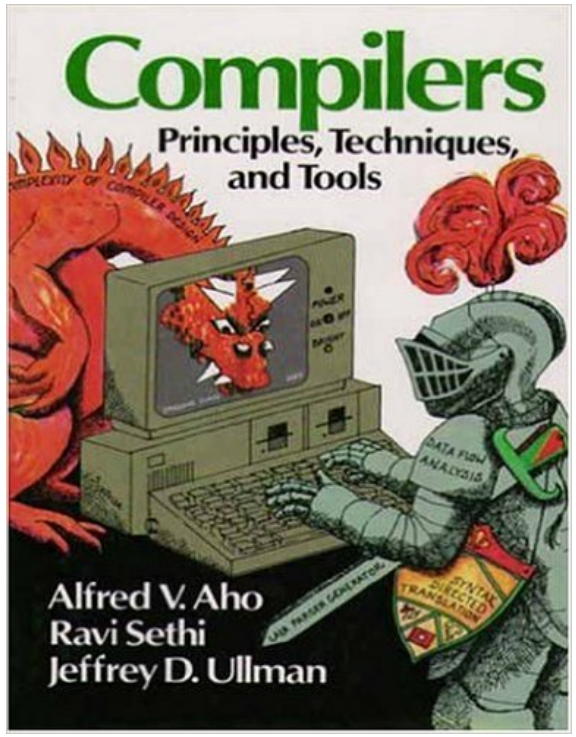
Plan

Informations sur l'UE

- Code : S3IN503
- 20h = 2h CM + 10h TD + 8h TP
- 2 ECTS
- MCC : E(1) TP(1)

Bibliographie

- Beaucoup de livres sur le sujet
- Référence = le « Dragon book »
- Voir la BU et la base d'ebooks en informatique

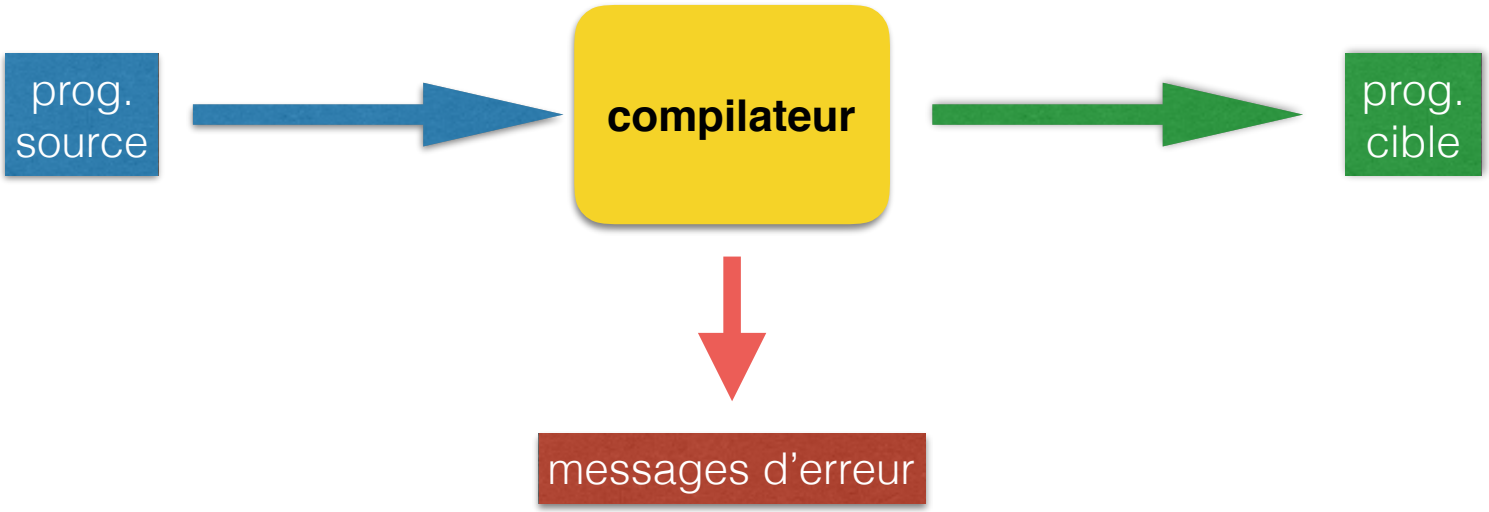


Plan

Compilateur

Définition

*Un **compilateur** est un programme qui lit le code d'un programme écrit dans un **langage source**, vérifie la présence d'erreurs éventuelles et le **traduit** en un programme équivalent écrit dans un langage **langage cible**.*



Interpréteur

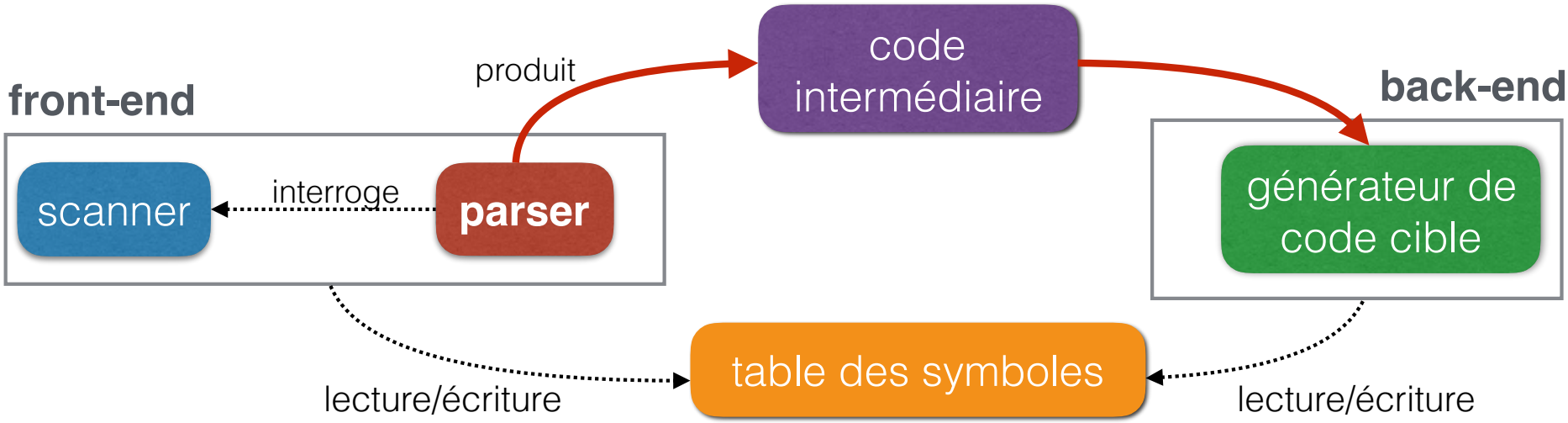
Définition

*Un **interpréteur** est un programme qui lit le code d'un programme écrit dans un **langage source**, vérifie la présence d'erreurs éventuelles et **exécute** les instructions du programme source.*

Exemples

- Compilateurs : Javac, GCC
- Interpréteurs : interpréteurs Python, navigateurs web

Structure d'un compilateur



Structure d'un interpréteur

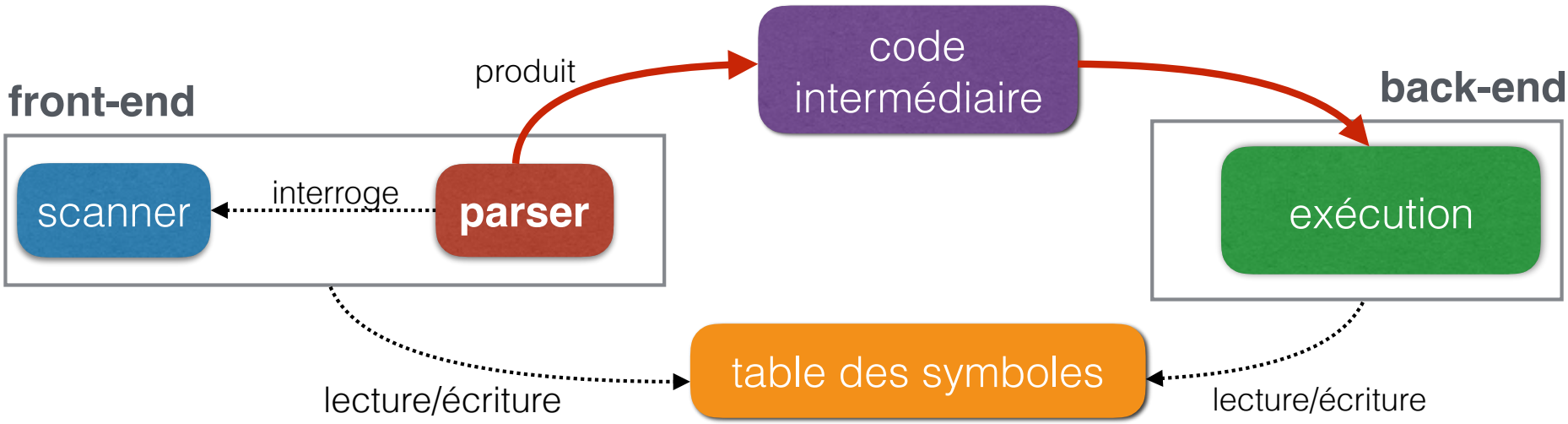


Table des symboles

Structure de données stockant des informations sur les symboles utilisés par le compilateur/interpréteur (ceux du programme source, ceux générés. . .)

Stocke par exemple les identificateurs (noms de variables, de fonctions. . .) et leurs attributs (pour une variable : le type, la portée. . .)

Front-end

Effectue les tâches suivantes :

- **analyse lexicale** (effectuée par le **scanner**)
- **analyse syntaxique** (effectuée par le **parser**)
- **analyse sémantique**

Front-end

Analyse lexicale :

- (entrée) lit le programme source caractère par caractère
- supprime les informations inutiles (espaces, commentaires. . .)
- découpe le programme source en **lexèmes**
- (sortie) produit une suite de **tokens**

Analyse syntaxique :

- (entrée) lit la suite de tokens produite par le scanner
- vérifie la structure de cette suite

  La structure doit être conforme aux règles imposées par le langage source (par exemple en Java, pas de nombre à gauche d'une affectation)

Front-end

Analyse sémantique (*sens* du programme) : par exemple,

- récolte des informations sur les types
- signale les erreurs de typage
- si besoin, effectue des conversions de type

Code intermédiaire

- Code pour une machine abstraite
- Interface entre front-end et back-end
- Intérêt : réutilisabilité

  À partir des compilateurs $C_1 : L_1 \to L'_1$ et $C_2 : L_2 \to L'_2$ on obtient simplement un compilateur $C_{1,2} : L_1 \to L'_2$ en connectant le front-end de C_1 au back-end de C_2
- Exemples : code à trois adresses, SSA...

Back-end

- Compilateur : optimisation du code intermédiaire et génération du code cible

  Un aspect crucial peut être la gestion de la mémoire (par exemple si la cible est un langage de bas niveau comme l'assembleur ou le langage machine)

- Interpréteur : exécution des instructions du code intermédiaire (ou directement du programme source dans des cas particuliers comme le langage Basic)

Exemple

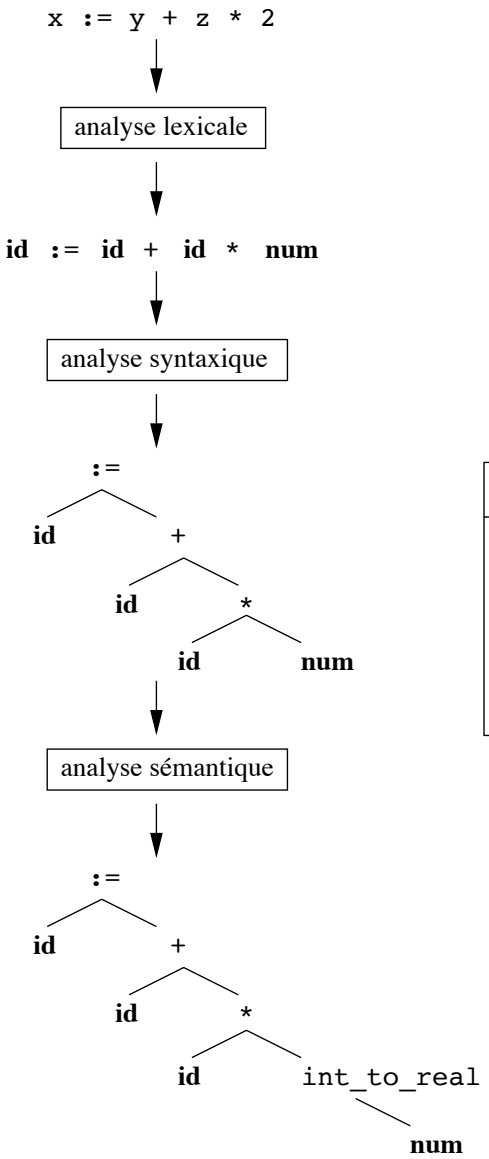


table des symboles

@	lexème	token	type
1	x	**id**	real
2	y	**id**	real
3	z	**id**	real
4	2	**num**	int

Exemple (suite)

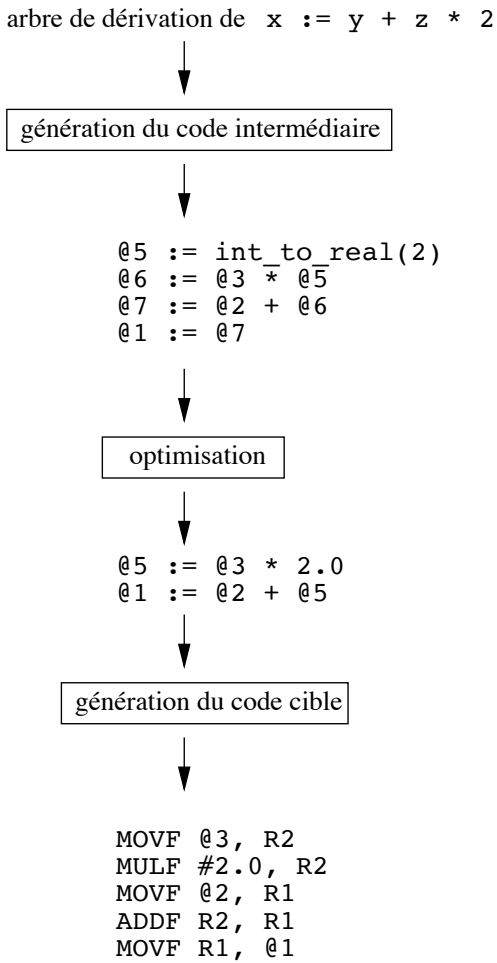


table des symboles

@	lexème	token	type
1	x	**id**	real
2	y	**id**	real
3	z	**id**	real
4	2	**num**	int
5	t1	**id**	real
6	t2	**id**	real
7	t3	**id**	real

Plan

Analyse lexicale

Effectuée par le scanner qui lit le programme source, le découpe en **lexèmes** et produit des **tokens**

Définition

*Un **lexème** est un mot appartenant au vocabulaire du langage source (par exemple l'instruction `if` ou la variable `x` dans un programme Java).*

Définition

*Un **token** est un mot qui représente un ensemble de lexèmes ayant une signification commune (par exemple* **id** *représente tous les noms de variables,* **num** *représente toutes les constantes numériques).*

Erreurs lexicales

- Lorsque des caractères lus ne correspondent à aucun token connu (lecture d'un symbole non admis dans le langage, identificateur trop long. . .)
- Signalées par le scanner

Attribut d'un token

Le scanner

- enregistre des informations concernant les tokens dans la table des symboles
- fournit des couples (*token*, *attribut*) au parser où *attribut* est
  - soit une unique information concernant *token*
  - soit un pointeur sur l'entrée de la table des symboles où les informations sur *token* sont stockées
  - soit le pointeur `NULL` si aucune information sur *token* n'est nécessaire

Attribut d'un token : exemple

Table des symboles				
@	lexème	token	type	num. de ligne
1	x	**id**	int	3
2	y	**id**	real	5

L'instruction `x := y + 2` est traduite en

(**id**,`@1`), (**assign**,`NULL`), (**id**,`@2`), (**add**,`NULL`), (**num**,2)

Définition et reconnaissance des tokens

L'ensemble des lexèmes correspondant à un token est défini au moyen d'une **expression régulière**

Un **automate fini déterministe (AFD)** correspondant à l'expression régulière est utilisé pour reconnaitre le token

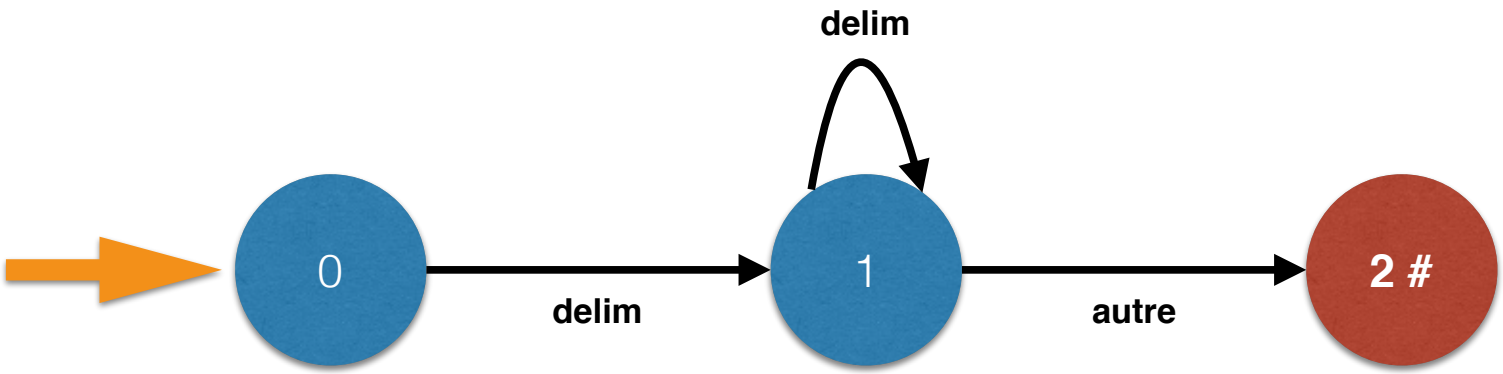
Définition des tokens : exemple

token	=	expression régulière
delim	=	**blanc** | **tab** | **nl**
espace	=	**delim**$^+$
lettre	=	`a` | `b` | ... | `z` | `A` | `B` | ... | `Z`
chiffre	=	`0` | `1` | ... | `9`
id	=	**lettre**(**lettre** | **chiffre**)*
num	=	**chiffre**$^+$(.**chiffre**$^+$)?

- '|' = 'ou bien'
- '$*$' = zéro, une ou plusieurs occurrences
- '$+$' = une ou plusieurs occurrences
- '?' = zéro ou une occurrence

Reconnaissance des tokens : exemple

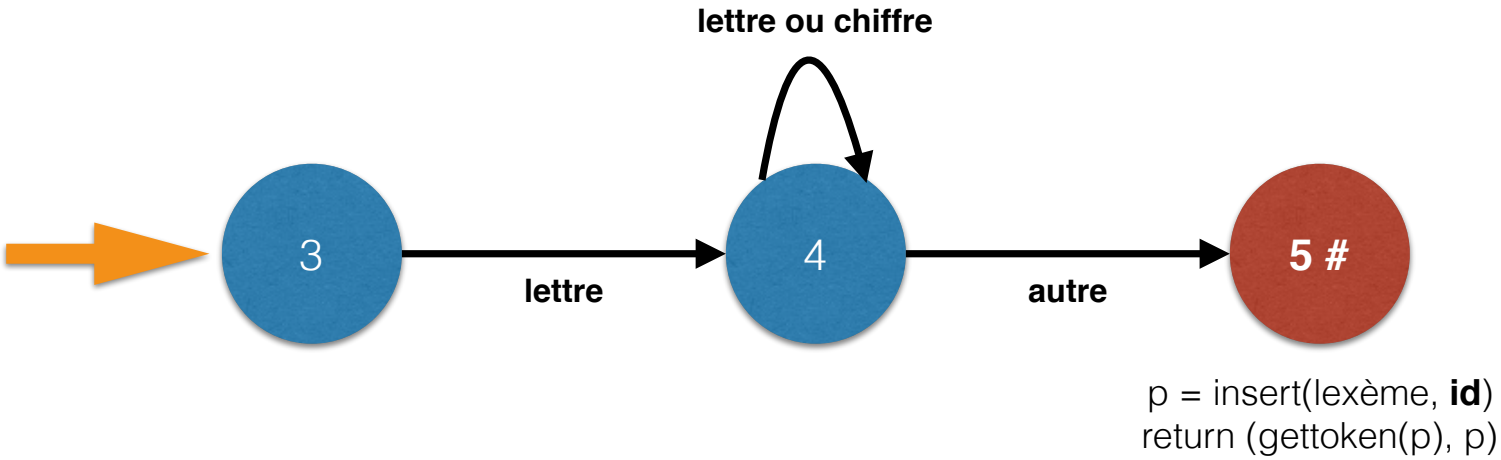
AFD pour le token **espace** :



signifie « reculer la tête de lecture »

Reconnaissance des tokens : exemple

AFD pour le token **id** :



Reconnaissance des tokens

Les fonctions *insert* et *gettoken* assurent la communication avec la table des symboles :

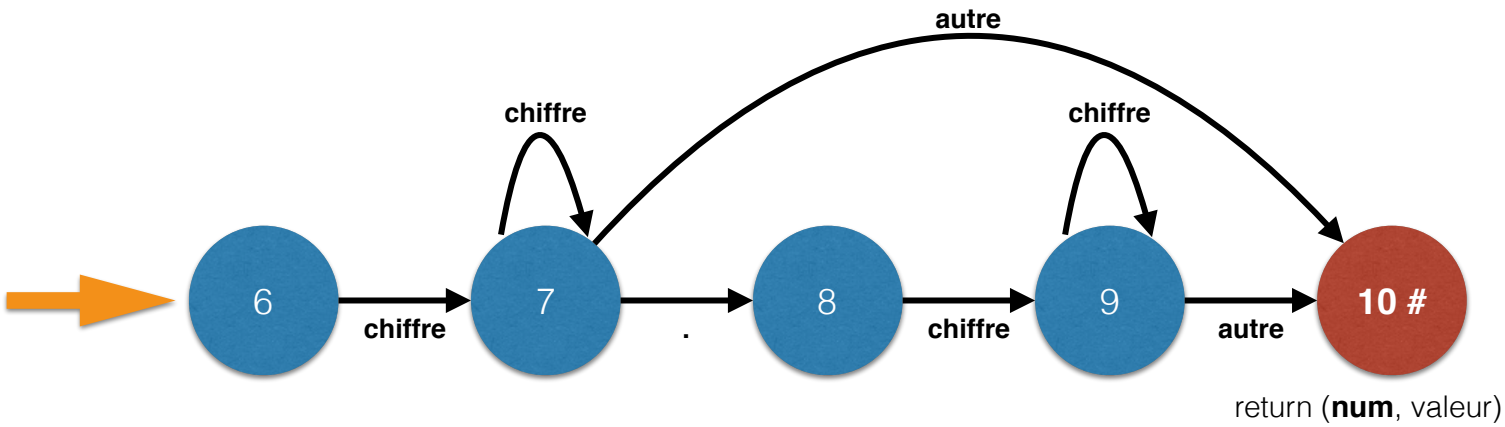
- *insert*(*lexème*, *token*) insère le couple (*lexème*, *token*) dans la table des symboles et renvoie la ligne p du couple dans la table

  Si *lexème* est déja stocké dans la table, alors n'insère rien et renvoie la ligne p où *lexème* est stocké

- *gettoken*(p) renvoie le token stocké à la ligne p

Reconnaissance des tokens : exemple

AFD pour le token **num** :



Et les mots-clés du langage source ?

token	=	lexème (mot-clé)
if	=	`if`
else	=	`else`
for	=	`for`
	⋮	

- Initialiser la table des symboles en y insérant tous les mots-clés
- Utiliser l'AFD défini pour **id**

Plan

Grammaire algébrique

Définition

C'est un quadruplet $(\mathcal{V}, \mathcal{T}, S, \mathcal{P})$ *où :*

- $\mathcal{V}$ *ensemble fini de* **symboles non-terminaux**
- $\mathcal{T}$ *ensemble fini de* **symboles terminaux**, *disjoint de* $\mathcal{V}$
- $S \in \mathcal{V}$ *est le* **symbole de départ**
- $\mathcal{P}$ *ensemble fini de* **productions** *ayant la forme*

  $A \to \alpha$ *où* $A \in \mathcal{V}$ *et* $\alpha \in (\mathcal{V} \cup \mathcal{T})^*$

Les productions indiquent les remplacements possibles

Une **dérivation** est une suite de remplacements successifs

Langage engendré par une grammaire algébrique

Définition

C'est l'ensemble de tous les mots, formés de symboles terminaux uniquement, que l'on obtient en utilisant les productions à partir du symbole de départ.

On note $\mathcal{L}(G)$ le langage engendré par une grammaire algébrique G

Grammaire algébrique : exemple

$$G = \left(\{S\}, \{a, b\}, S, \{S \xrightarrow{1} aSb, S \xrightarrow{2} \varepsilon\}\right) \text{ où } \varepsilon = \text{mot vide}$$

$$\begin{array}{ccccccccc} \underline{S} & \xrightarrow{1} & a\underline{S}b & \xrightarrow{1} & aa\underline{S}bb & \xrightarrow{1} & aaa\underline{S}bbb & \xrightarrow{1} & aaaa\underline{S}bbbb \quad \cdots \\ \downarrow 2 & & \downarrow 2 & & \downarrow 2 & & \downarrow 2 & & \downarrow 2 \\ \varepsilon & & ab & & aabb & & aaabbb & & aaaabbbb \end{array}$$

$$\mathcal{L}(G) = \{\varepsilon, ab, aabb, \ldots\} = \{a^n b^n \mid n \in \mathbb{N}\}$$

Grammaire algébrique ambiguë

Définition

*G est dite **ambiguë** lorsqu'il existe un mot de $\mathcal{L}(G)$ ayant deux dérivations distinctes à partir du symbole de départ.*

Exemple : $G = \left(\{I\}, \{\textbf{if}, \textbf{cond}, \textbf{else}, \textbf{autre}\}, I, \mathcal{P}\right)$ où

$$\mathcal{P} = \left\{ I \xrightarrow{1} \textbf{if cond } I,\ I \xrightarrow{2} \textbf{if cond } I \textbf{ else } I,\ I \xrightarrow{3} \textbf{autre} \right\}$$

Le mot **if cond if cond autre else autre** a deux dérivations distinctes

Lesquelles ?

Grammaire algébrique récursive à gauche

Définition

*$G = (\mathcal{V}, \mathcal{T}, S, \mathcal{P})$ est dite **récursive à gauche** s'il existe une dérivation de la forme $A \to \cdots \to A\alpha$ où $A \in \mathcal{V}$ et $\alpha \in (\mathcal{V} \cup \mathcal{T})^*$.*

Théorème

Pour toute grammaire algébrique G récursive à gauche, on peut construire une grammaire algébrique G' non-récursive à gauche telle que $\mathcal{L}(G) = \mathcal{L}(G')$.

Voir par exemple ici ou là pour un algorithme de construction de G'

Récursivité à gauche : exemples

Désormais, on donne les exemples de grammaires sous la forme :

$$G_1 = \begin{cases} S \to (L) \mid a \\ L \to L, S \mid S \end{cases} \qquad G_2 = \begin{cases} E \to E + T \mid T \\ T \to T * F \mid F \\ F \to (E) \mid \mathbf{id} \end{cases}$$

Symboles non-terminaux = les lettres majuscules
Symboles terminaux = les autres symboles
Symbole de départ = le symbole à gauche de la première production

- Quels sont les langages engendrés par G_1 et G_2 ?
- Pourquoi G_1 et G_2 sont-elles récursives à gauche ?

Premiers symboles

$G = (\mathcal{V}, \mathcal{T}, S, \mathcal{P})$ grammaire algébrique

$\alpha \in (\mathcal{V} \cup \mathcal{T})^*$

Définition

premiers(α) *est l'ensemble des symboles terminaux qui apparaissent en première position dans un mot dérivable depuis* α.

premiers(α) *contient le mot vide* ε *si celui-ci est dérivable depuis* α.

Premiers symboles : exemple

$$G = \begin{cases} A \rightarrow BCa \\ B \rightarrow b \mid \varepsilon \\ C \rightarrow c \mid \varepsilon \end{cases}$$

- $premiers(C) = ?$
- $premiers(B) = ?$
- $premiers(A) = ?$

Plan

Analyse syntaxique

La syntaxe d'un langage de programmation peut souvent être décrite par une grammaire algébrique

L'analyse syntaxique est effectuée par un parser qui

- vérifie que la suite de tokens produite par le scanner est un mot du langage engendré par cette grammaire
- signale les **erreurs syntaxiques** : correspondent au cas où la suite de tokens n'est pas un mot du langage de la grammaire

Méthodes d'analyse syntaxique

Idée : construire une dérivation de la suite de tokens

- **Méthodes descendantes (top-down)** : depuis le symbole de départ elles essaient de « descendre » vers la suite de tokens
- **Méthodes ascendantes (bottom-up)** : depuis la suite de tokens elles essaient de « remonter » vers le symbole de départ

La suite de tokens est lue de gauche à droite, un token à la fois

Définition

*Le **lookahead** est le token sous la tête de lecture à un moment donné.*

Analyse descendante récursive prédictive

Idée

*En fonction du lookahead, **choisir** une production et l'**appliquer** dans le « bon » sens.*

(Implémentation des productions) pour chaque non-terminal A, construire une **procédure** qui choisit une production de la forme $A \to \alpha$ et l'applique

(Initialisation) placer la tête de lecture sous le premier token et exécuter la procédure associée au symbole de départ

Implémentation des productions

La **procédure** $A()$ **pour** $A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$:

- **choisit** quelle production appliquer en fonction du lookahead
  - si lookahead $\in$ *premiers*(α_i) alors $A \rightarrow \alpha_i$ est choisie
  - une production de la forme $A \rightarrow \alpha_i$ avec $\alpha_i = \varepsilon$ est choisie lorsque lookahead $\notin$ *premiers*(α_j) pour tout $j \neq i$
  - si pour tout i on a $\alpha_i \neq \varepsilon$ et lookahead $\notin$ *premiers*(α_i) alors **erreur**
- **applique** $A \rightarrow \alpha_i$ en lisant α_i de gauche à droite
  - lecture d'un non-terminal $B \Rightarrow$ appel de la procédure $B()$ associée
  - lecture d'un terminal égal au lookahead $\Rightarrow$ lecture du prochain token
  - lecture d'un terminal différent du lookahead $\Rightarrow$ **erreur**

Analyse descendante récursive prédictive : exemple

$$G = \begin{cases} S \to (S + F) \mid F \\ F \to \mathbf{num} \mid \mathbf{id} \end{cases}$$

Productions $S \to \cdots$	Productions $F \to \cdots$
$premiers((S + F)) = \{(\}$	$premiers(\mathbf{num}) = \{\mathbf{num}\}$
$premiers(F) = \{\mathbf{num}, \mathbf{id}\}$	$premiers(\mathbf{id}) = \{\mathbf{id}\}$

Analyse descendante récursive prédictive : exemple

```
def S():
  if lookahead == '(':
    match('(')
    S()
    match('+')
    F()
    match(')')
  else:
    F()

def F():
  if lookahead in [NUM,ID]:
    match(lookahead)
  else:
    raise SyntaxError
```

```
def match(token):
  if lookahead == token:
    lookahead, attr = scanner.nexttoken()
  else:
    raise SyntaxError

def start():
  lookahead, attr = scanner.nexttoken()
  S()
  if lookahead != DONE:
    raise SyntaxError
```

Conditions d'application de cette technique

La grammaire algébrique doit vérifier les propriétés suivantes :

- non récursive à gauche
- $\mathit{premiers}(\alpha_i) \cap \mathit{premiers}(\alpha_j) = \varnothing$ pour $A \to \alpha_1 \mid \ldots \mid \alpha_n$ et $i \neq j$

Pourquoi, exemples ?

Conditions d'application de cette technique

La condition $premiers(\alpha_i) \cap premiers(\alpha_j) = \varnothing$ peut être assouplie

$$G = \begin{cases} E \to T + E \mid T \\ T \to F * T \mid F \\ F \to (E) \mid \textbf{num} \mid \textbf{id} \end{cases}$$

Ici on peut s'en sortir même si

$$premiers(T + E) \cap premiers(T) \neq \varnothing \quad \text{et}$$
$$premiers(F * T) \cap premiers(F) \neq \varnothing$$

Pourquoi ?

Schéma de traduction

Définition

C'est une grammaire algébrique $G = (\mathcal{V}, \mathcal{T}, S, \mathcal{P})$ « enrichie » :

- *un ensemble d'**attributs** est associé à chaque symbole de $\mathcal{V} \cup \mathcal{T}$*
- *dans les parties droites des productions, des **actions sémantiques** délimitées par des accolades sont insérées entre les symboles.*

Idée : effectuer des tâches supplémentaires durant l'analyse syntaxique

Schéma de traduction : exemple

$$
\begin{array}{lcl}
L & \rightarrow & E\ ;\ \{print(E.val)\}\ L \ \mid\ \varepsilon \\
E & \rightarrow & T + E_1\ \{E.val := T.val + E_1.val\} \mid T\ \{E.val := T.val\} \\
T & \rightarrow & F * T_1\ \{T.val := F.val * T_1.val\} \mid F\ \{T.val := F.val\} \\
F & \rightarrow & (E)\ \{F.val := E.val\} \mid \mathbf{num}\ \{F.val := \mathbf{num}.val\}
\end{array}
$$

- L'attribut *val* stocke la valeur liée au symbole associé
- Les indices (ex. dans E_1) servent à différencier des occurrences d'un même symbole dans une production

Schéma de traduction : implémentation

La **fonction** $A()$ **pour** $A \to \alpha_1 \mid \ldots \mid \alpha_n$:

- a pour **variables locales** les attributs des symboles de $\alpha_1, \ldots, \alpha_n$
- **choisit** la production $A \to \alpha_i$ à appliquer comme précédemment
- **applique** $A \to \alpha_i$ en lisant α_i de gauche à droite
  - terminal $a \Rightarrow$ sauver la valeur des attributs de a dans les variables locales correspondantes et exécuter `match`(a)
  - non-terminal $B \Rightarrow$ effectuer l'affectation $(b_1, \ldots, b_m) := B()$ où les b_j sont les variables locales associées aux attributs de B
  - action sémantique $\Rightarrow$ exécuter l'action
- **retourne** la valeur des attributs de A

Schéma de traduction : suite de l'exemple

```
def L():
  if lookahead in [NUM, '(']: # Pourquoi ?
    # production L -> E ; L
    E_val = E()
    match(';')
    print E_val
    L()
  else:
    # production L -> epsilon : ne rien faire
    pass
```

Exercice 5.1

- Écrire un programme qui prend en entrée un ficher texte et lui applique un parser basé sur le schéma de traduction précédent (vérifier les erreurs lexicales et syntaxiques)
- Que fait ce parser ?
- Ajouter la possibilité d'écrire des commentaires commençant par le caractère # et s'étendant jusqu'à la fin de la ligne (comme en Python)

Exercice 5.2

Écrire un programme qui

- prend en entrée un fichier texte contenant une suite d'affectations $id := E;$ où id est un identificateur (nom de variable) et E est une expression arithmétique construite à partir des opérateurs +, -, *, /, des parenthèses, de contantes numériques et d'identificateurs
- calcule la valeur des expressions arithmétiques
- affiche la nouvelle valeur de id après chaque affectation $id := E;$
- vérifie les erreurs lexicales et syntaxiques et les variables sans valeur

Exemples :

- la suite $n := (2 + 3) * 4;\ m := n + 4;$ produit $n = 20; m = 24;$
- la suite $n := (2 + 3) * 4;\ m := n + 4 + \underline{p};$ produit une erreur

Exercice 5.3

Écrire un programme qui

- prend en entrée un fichier texte contenant une suite d'expressions arithmétiques infixées terminées par un point-virgule et construites à partir des opérateurs +, -, *, /, des parenthèses, de contantes numériques et d'identificateurs (noms de variables)
- traduit ces expressions en expressions postfixées équivalentes
- vérifie les erreurs lexicales et syntaxiques

Exemple : la suite $(x + 3) * 4;\ x + y + 4;$ est traduite en $x\ 3\ +\ 4\ *;\ x\ y\ +\ 4\ +;$

Plan

Analyse sémantique

Vérification de propriétés liées au sens du programme, par exemple :

- (S) types
- (S) portée des variables
- (FC) initialisation des variables
- (FC) portions de code inaccessibles
- (FC) l'exécution de toute fonction se termine par un `return`

où

(S) : vérification durant l'analyse syntaxique (assez simple)

(FC) : dépendent du flot de contrôle (plus compliqué)

Vérification des types via un schéma de traduction

Table des symboles :

- on ajoute la colonne **type** à la table des symboles

Fonctions :

- *settype*(*adr*, *t*) insère le type *t* à la ligne *adr* de la table des symboles
- *gettype*(*adr*) renvoie le type stocké à la ligne *adr* de la table des symboles

Schéma de traduction :

- voir plus bas

Plan

Génération de code intermédiaire

Scanner :

- lorsque le scanner lit une constante numérique, il l'insère (via la fonction *insert*) dans la table des symboles et renvoie (**num**, *adr*) où *adr* est le numéro de ligne où la constante a été insérée

Table des symboles :

- on ajoute la colonne **valeur** à la table des symboles

Fonctions :

- *setvalue*(*adr*, *v*) insère la valeur *v* à la ligne *adr* de la table des symboles
- *getvalue*(*adr*) renvoie la valeur stockée à la ligne *adr* de la table des symboles

Code à 3 adresses

Instruction	Représentation (quadruplet)
`x := y op z`	`(op, y, z, x)`
`x := y`	`(:=, y, _, x)`
`goto L`	`(goto, _, _, L)`
`if x relop y goto L`	`(relop, x, y, L)`
`print x`	`(print, _, _, x)`
`halt`	`(halt, _, _, _)`

Code à 3 adresses

- Les quadruplets ont la forme

  `(opérateur, argument1, argument2, résultat)`

- Les quadruplets sont stockés dans le tableau **Icode**
- `x`, `y`, `z` sont des numéros de ligne (adresse) dans la table des symboles
- `L` est un numéro de ligne (adresse) dans le tableau Icode
- `_` correspond à un champ vide
- `op` est un opérateur arithmétique binaire (+, -, ...)
- `relop` est un opérateur relationnel (<, <=, ...)

Exercice 7.1

Écrire un programme en code à 3 adresses qui calcule 10! au moyen d'une boucle puis affiche ce nombre et s'arrête

Écrire un programme en code à 3 adresses qui calcule $pgcd(20, 12)$ au moyen de l'algorithme d'Euclide puis affiche ce nombre et s'arrête

Exercice 7.2

Écrire une classe `Icode` permettant de gérer et d'exécuter du code à 3 adresses

Tester cette classe sur les programmes de l'exercice 7.1

Génération de code à 3 adresses via un schéma de traduction

Attribut *adresse* :

- **id**.*adresse* = numéro de ligne de la table des symboles où est stocké l'identificateur
- **num**.*adresse* = numéro de ligne de la table des symboles où est stockée la constante numérique
- $E.adresse$ et $F.adresse$ = numéro de ligne de la table des symboles où est stocké l'identificateur associé à E ou F

Fonctions :

- *newid*() crée un nouvel identificateur, l'insère dans la table des symboles et renvoie le numéro de ligne où il est stocké
- $emit(op, arg_1, arg_2, res)$ insère le quadruplet donné à la fin de Icode

Génération de code à 3 adresses : affectations

$$I \rightarrow \textbf{id} := E \ \{\text{si } gettype(\textbf{id}.adresse) \neq gettype(E.adresse) \text{ alors } \textbf{erreur} \text{ sinon } emit(:=, E.adresse, \_, \textbf{id}.adresse)\}$$

$$E \rightarrow F + E_1 \ \{E.adresse := newid() \text{ si } gettype(F.adresse) = gettype(E_1.adresse) \text{ alors } settype(E.adresse, gettype(F.adresse)) \text{ sinon } settype(E.adresse, flottant) \ emit(+, F.adresse, E_1.adresse, E.adresse)\}$$

$$\mid \quad F \ \{E.adresse := F.adresse\}$$

$$F \rightarrow \textbf{id} \ \{\text{si } gettype(\textbf{id}.adresse) = \texttt{null} \text{ alors } \textbf{erreur} \text{ sinon } F.adresse := \textbf{id}.adresse\}$$

$$\mid \quad \textbf{num} \ \{F.adresse := \textbf{num}.adresse\}$$

Exercice 7.3

$$
\begin{array}{lcl}
P & \rightarrow & \textbf{program id } D \textbf{ begin } L \textbf{ end .} \\
D & \rightarrow & \textbf{var } D' \mid \varepsilon \\
D' & \rightarrow & \textbf{id } : Y \ ; \ D' \mid \varepsilon \qquad\qquad Y \rightarrow \textbf{int} \mid \textbf{float} \\
L & \rightarrow & I \ ; L \mid \varepsilon \\
I & \rightarrow & \textbf{id } := E \mid \textbf{print}(E) \\
E & \rightarrow & T + E \mid T - E \mid T \\
T & \rightarrow & F * T \mid F \ / \ T \mid F \textbf{ mod } T \mid F \\
F & \rightarrow & (E) \mid \textbf{id} \mid \textbf{num}
\end{array}
$$

Exercice 7.3

Écrivez un interpréteur pour les programmes dont la syntaxe obéit aux règles de cette grammaire

Votre interpréteur devra

- produire du code à 3 adresses puis exécuter ce code
- signaler les erreurs lexicales et syntaxiques
- vérifier le typage et signaler les erreurs

Exercice 7.3

Un exemple de programme dont la syntaxe est correcte :

```
program Exemple
var a: int; b: int; r: float;
begin
  a := 1;
  b := 4;
  r := 3.0*(a+b);
  print(r);
end.
```

Génération de code à 3 adresses : relop

Idée : générer des instructions `goto` incomplètes (manque l'adresse cible) puis les compléter dès que possible

Attributs :

- $B.truelist$ = liste d'adresses des `goto` à exécuter quand B est vraie
- $B.falselist$ = liste d'adresses des `goto` à exécuter quand B est fausse
- **relop**.op = opérateur relationnel correspondant à **relop**

Fonctions :

- $fillgoto(l, adr)$ complète les `goto` de la liste l par l'adresse adr
- $nextquad()$ renvoie l'adresse de la prochaine instruction qui sera générée

Génération de code à 3 adresses : relop

$$
\begin{aligned}
B \quad \rightarrow \quad E_1 \ \textbf{relop} \ E_2 \ \{ & B.truelist := [nextquad()] \\
& B.falselist := [nextquad() + 1] \\
& emit(\textbf{relop}.op, E_1.adresse, E_2.adresse, \_) \\
& emit(\texttt{goto}, \_, \_, \_) \}
\end{aligned}
$$

Génération de code à 3 adresses : instructions

$$
\begin{array}{lcl}
L & \rightarrow & I \ ; \ M \ L \ \mid \ \varepsilon \\
I & \rightarrow & \textbf{if } B \textbf{ then } M_1 \ I_1 \ N \textbf{ else } M_2 \ I_2 \\
 & \mid & \textbf{while } M_1 \ B \textbf{ do } M_2 \ I_1 \\
 & \mid & \textbf{begin } L \textbf{ end} \\
 & \mid & \textbf{id} := E \\
 & \mid & \textbf{print}(E) \\
M & \rightarrow & \varepsilon \\
N & \rightarrow & \varepsilon
\end{array}
$$

M et N sont des non-terminaux servant de **marqueurs**

Génération de code à 3 adresses : instructions

Idée : comme précédemment, générer des instructions `goto` incomplètes (manque l'adresse cible) puis les compléter dès que possible

Attributs :

- *M.quad* = adresse de l'instruction à exécuter quand M a été traité
- *nextlist* pour les symboles L, I et N = liste d'adresses des `goto` à exécuter quand L, I ou N est terminé

Génération de code à 3 adresses : instructions

$$L \rightarrow I\ ;\ M\ L_1 \left\{ fillgoto(I.nextlist, M.quad),\ L.nextlist := L_1.nextlist \right\}$$

$$\mid\ \varepsilon \left\{ L.nextlist := [\ ] \right\}$$

$$I \rightarrow \textbf{if}\ B\ \textbf{then}\ M_1\ I_1\ N\ \textbf{else}\ M_2\ I_2$$
$$\left\{ fillgoto(B.truelist, M_1.quad)\right.$$
$$fillgoto(B.falselist, M_2.quad)$$
$$\left. I.nextlist := merge(I_1.nextlist, N.nextlist, I_2.nextlist) \right\}$$

$$\mid\ \textbf{print}(E) \left\{ emit(\texttt{print}, \_, \_, E.adresse),\ I.nextlist := [\ ] \right\}$$

$$M \rightarrow \varepsilon \left\{ M.quad := nextquad() \right\}$$

$$N \rightarrow \varepsilon \left\{ N.nextlist := [nextquad()],\ emit(\texttt{goto}, \_, \_, \_) \right\}$$

Exercice 7.4

On ajoute les productions suivantes à la grammaire de l'exercice 7.3 :

$$I \rightarrow \textbf{begin } L \textbf{ end} \quad | \quad \textbf{if } B \textbf{ then } I \textbf{ else } I \quad | \quad \textbf{while } B \textbf{ do } I$$

Écrivez un interpréteur pour les programmes dont la syntaxe obéit aux règles de la grammaire obtenue

Votre interpréteur devra

- produire du code à 3 adresses puis exécuter ce code
- signaler les erreurs lexicales, syntaxiques et de typage