Théorie des Langages

Analyse Sémantique

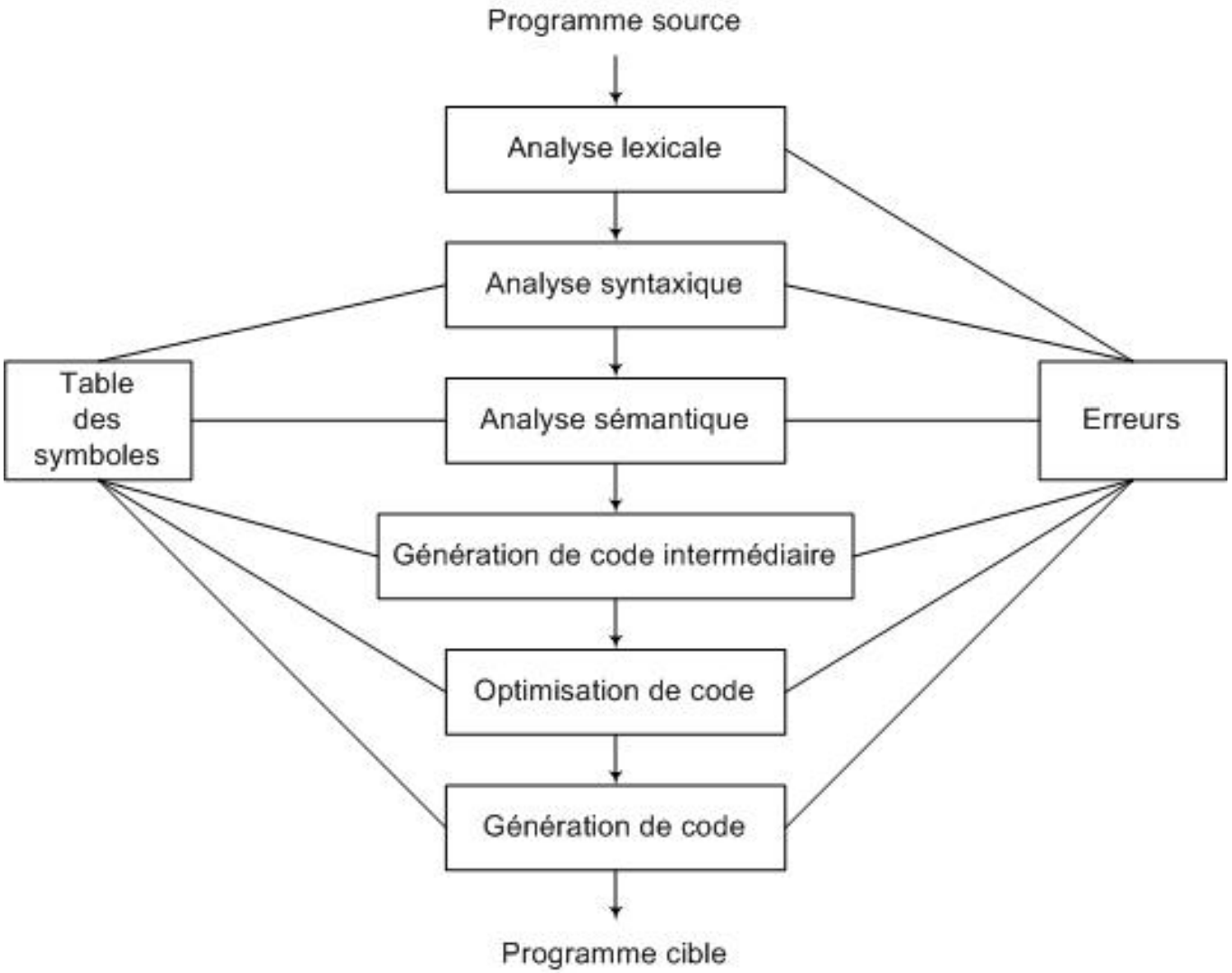
Claude Moulin

Université de Technologie de Compiègne

Printemps 2013

Sommaire

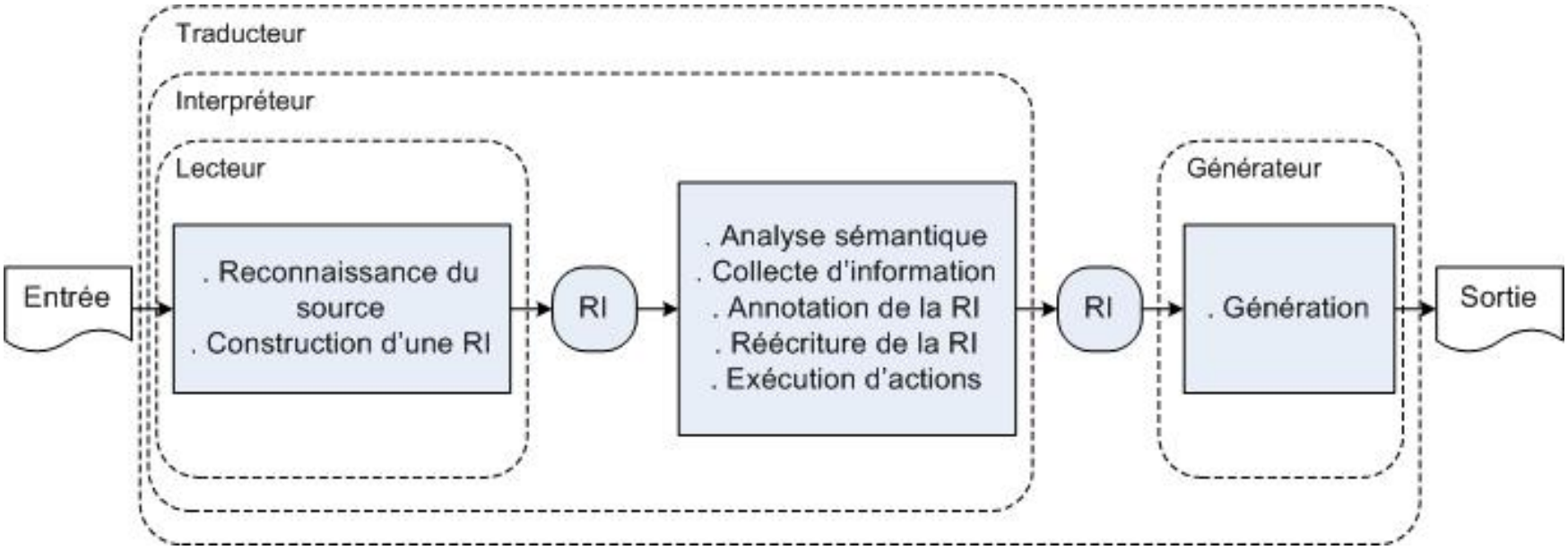
Phases


Programme source
Analyse lexicale
Analyse syntaxique
Table des symboles
Analyse sémantique
Erreurs
Génération de code intermédiaire
Optimisation de code
Génération de code
Programme cible

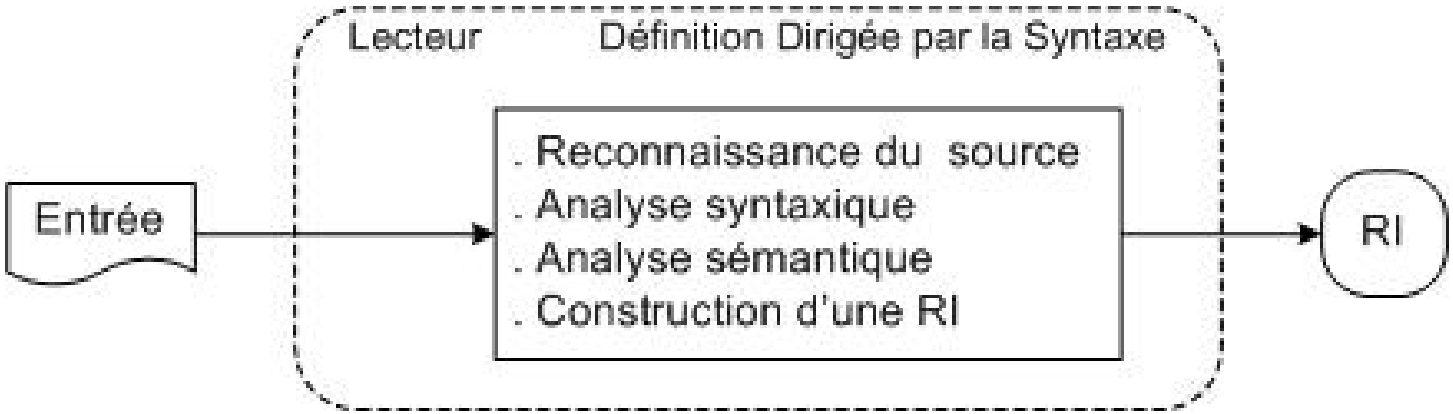
Phase d'analyse sémantique

- Contrôle des cohérences sémantiques.
  - Détermination des instructions, des expressions et des identificateurs.
  - Contrôles des types dans les affectations et les passages de paramètres.
  - Contrôle des types d'indices de tableaux
- Les grammaires hors contexte ne contiennent pas tous les éléments dont l'analyse sémantique a besoin (contexte du programme.)
- Grammaires contextuelles : délicates à manipuler
- Les processus dont on a besoin sont difficilement automatisables

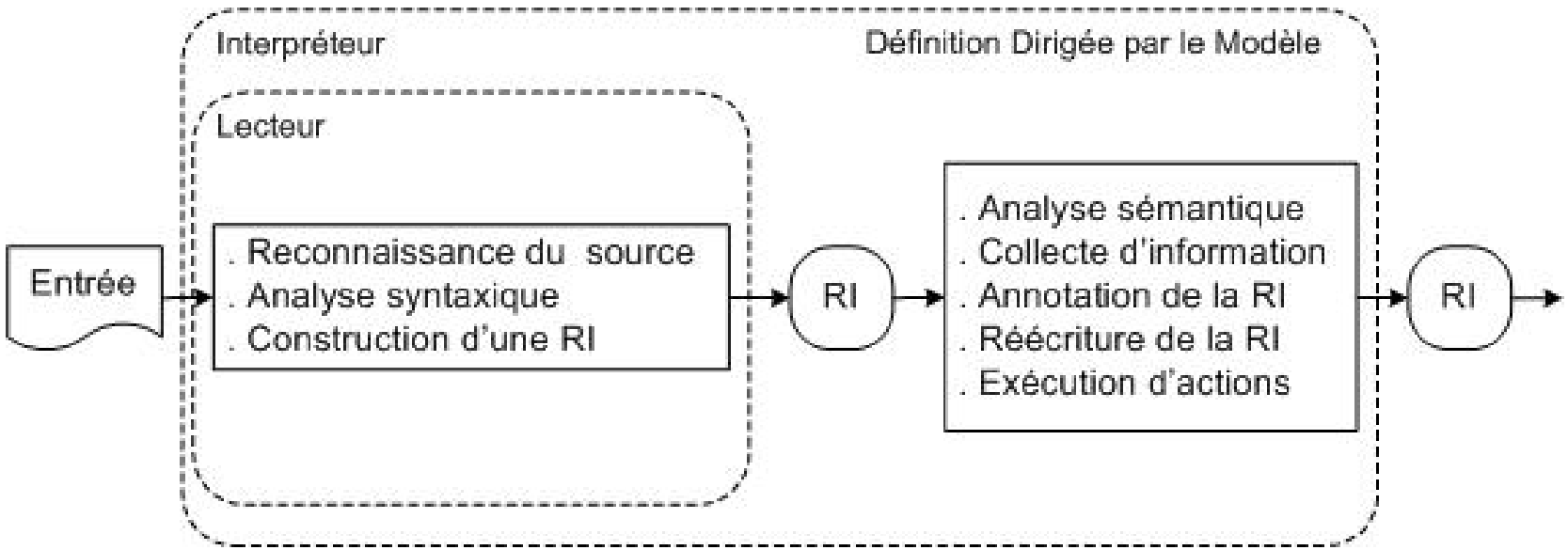
Processus général



Définition Dirigée par la Syntaxe



Définition Dirigée par le Modèle



Sommaire

Grammaires Attribuées - 1

Définition

Une grammaire attribuée est une grammaire hors contexte augmentée d'attributs, de règles sémantiques et de conditions.

- C'est donc un formalisme qui permet d'associer des actions (règles sémantiques) aux règles de production d'une grammaire. On parle de Définition Dirigée par la Syntaxe (modèle).

Grammaires Attribuées - 2

- Chaque symbole de la grammaire possède des attributs.
  - $X.a$ l'attribut a du symbole X.
  - X (partie gauche), X_1, X_2, ..., (partie droite), à partir du plus à gauche.
- Chaque règle de production de la grammaire possède un ensemble d'actions permettant de calculer la valeur des attributs.
- Une règle sémantique est une suite d'instructions algorithmiques.

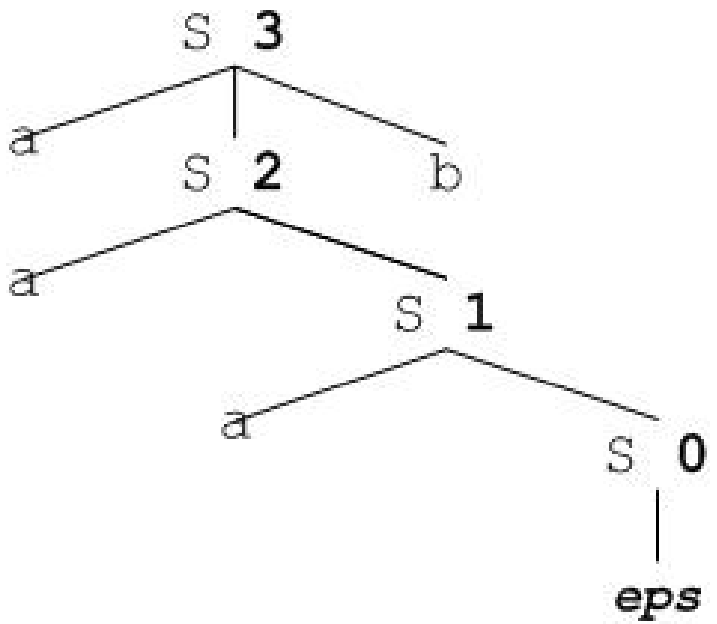
Exemple

G : $S \longrightarrow aSb/aS/\epsilon$
Question : déterminer le nombre de a ?

Production	Règle sémantique
$S \longrightarrow aSb$	S.nba := S_1.nba + 1
$S \longrightarrow aS$	S.nba := S_1.nba + 1
$S \longrightarrow \epsilon$	S.nba := 0
$S' \longrightarrow S\$$	// S.nba contient la valeur

Arbre Décoré

Chaîne : *aaab*



Types d'attributs

Attribut synthétisé

Un attribut synthétisé est attaché au symbole en partie gauche et se calcule en fonction des attributs des symboles de la partie droite.

- Un attribut synthétisé est attaché à un nœud et se calcule en fonction des attributs de ses fils.

Attribut hérité

Un attribut est hérité lorsqu'il est calculé à partir des attributs du non terminal de la partie gauche et des attributs des autres symboles de la partie droite.

- Dans l'arbre décoré, un attribut hérité dépend des attributs du nœud père et des attributs des nœuds frères.

Déclaration de variables

Déclaration des variables en C :

$D \longrightarrow T\ L$
$L \longrightarrow id\ R$
$R \longrightarrow ,\ id\ R$
$R \longrightarrow \epsilon$
$T \longrightarrow$ int
$T \longrightarrow$ real

Déclaration : **real** x, y, z

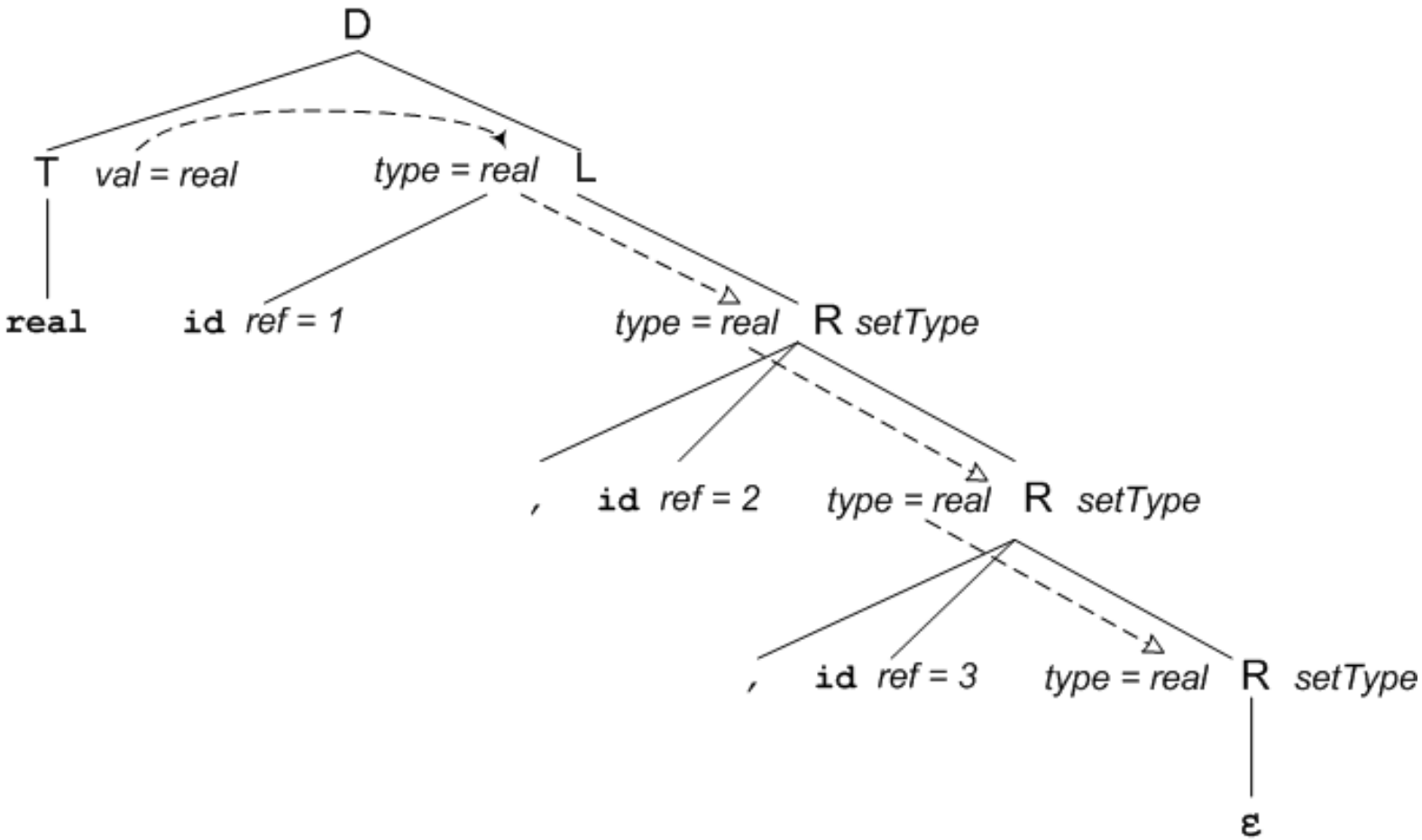
Déclaration de variables

Déclaration des variables en C :

Production	Règle sémantique
$D \longrightarrow T\ L$	L.type = T.val
$L \longrightarrow id\ R$	$R.type = L.type$ setType(id.ref,L.type)
$R \longrightarrow ,\ id\ R$	$R_1.type = R.type$ setType(id.ref,R.type)
$R \longrightarrow \epsilon$	
$T \longrightarrow$ int	T.val = integer
$T \longrightarrow$ real	T.val = real

Déclaration : **real** x, y, z

Graphe de dépendances



Définitions S et L attribuées

Définition S-attribuée

Une définition dirigée par la syntaxe n'ayant que des attributs synthétisés est appelée définition S-attribuée.

Définition L-attribuée

Une définition dirigée par la syntaxe est L-attribuée si tout attribut hérité d'un symbole de la partie droite d'une production ne dépend que :

- des attributs hérités du symbole en partie gauche et
- des attributs des symboles le précédant dans la production.

Sommaire

Schéma de traduction

Schéma de traduction

Un schéma de traduction est une définition dans laquelle l'ordre d'exécution des actions sémantiques est imposé.

- $A \longrightarrow \alpha X \{action\} Y \beta$: l'action est exécutée après que le sous-arbre issu de X a été construit et parcouru et avant que celui issu de Y ne le soit.
- On peut évaluer les attributs en même tant que l'on effectue l'analyse syntaxique (usage d'une pile).
- L'ordre d'évaluation des attributs est tributaire de l'ordre dans lequel les noeuds de l'arbre sont créés (méthode ascendante, méthode descendante).
- L'arbre de dérivation n'est pas réellement construit lors de l'analyse syntaxique.

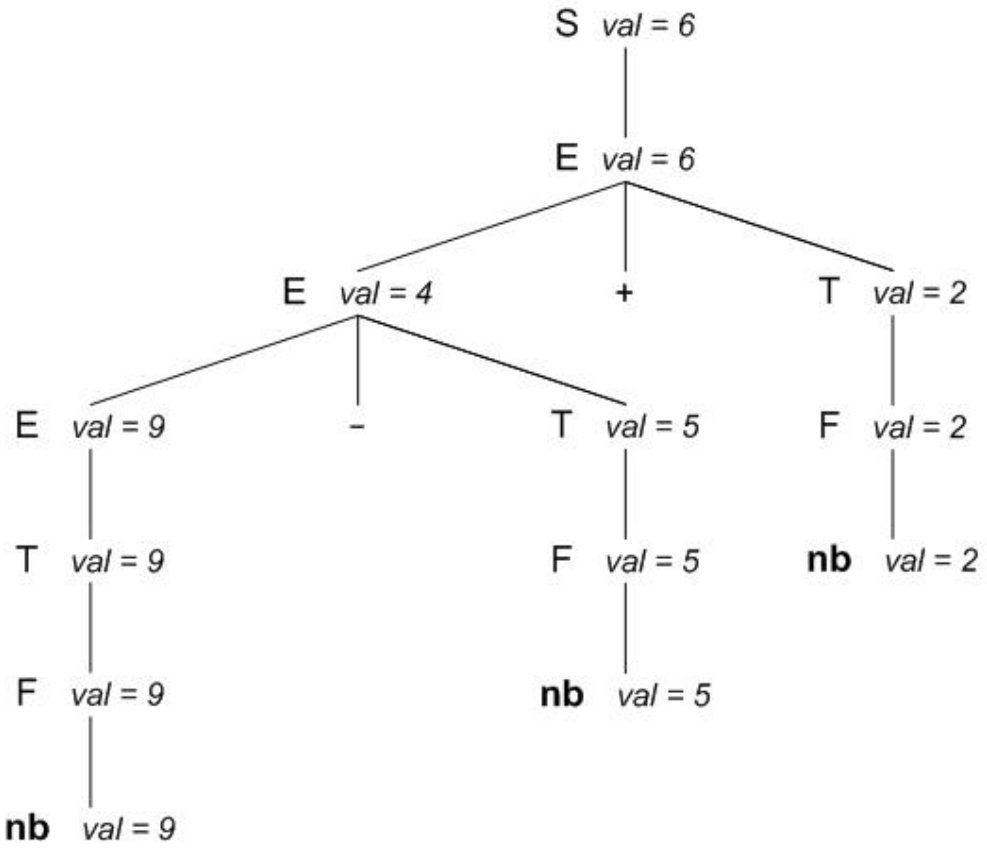
Analyse ascendante - 1

Production	Règle sémantique	Traduction
$S \longrightarrow E\$$	S.val := E.val	écrire p.depiler()
$E \longrightarrow E + T$	E.val := E_1.val+T.val	tmpT = p.depiler() tmpE = p.depiler() p.empiler(tmpE + tmpT)
$E \longrightarrow E - T$	E.val := E_1.val-T.val	tmpT = p.depiler() tmpE = p.depiler() p.empiler(tmpE - tmpT)
$E \longrightarrow T$	E.val := T.val	

Analyse ascendante - 2

Production	Règle sémantique	Traduction
$E \longrightarrow T$	E.val := T.val	
$T \longrightarrow T * F$	T.val := T_1.val*F.val	tmpF = p.depiler() tmpT = p.depiler() p.empiler(tmpT x tmpF)
$T \longrightarrow F$	T.val := F.val	
$F \longrightarrow (E)$	F.val := E.val	
$F \longrightarrow nb$	F.val := nb.val	p.empiler(nb)

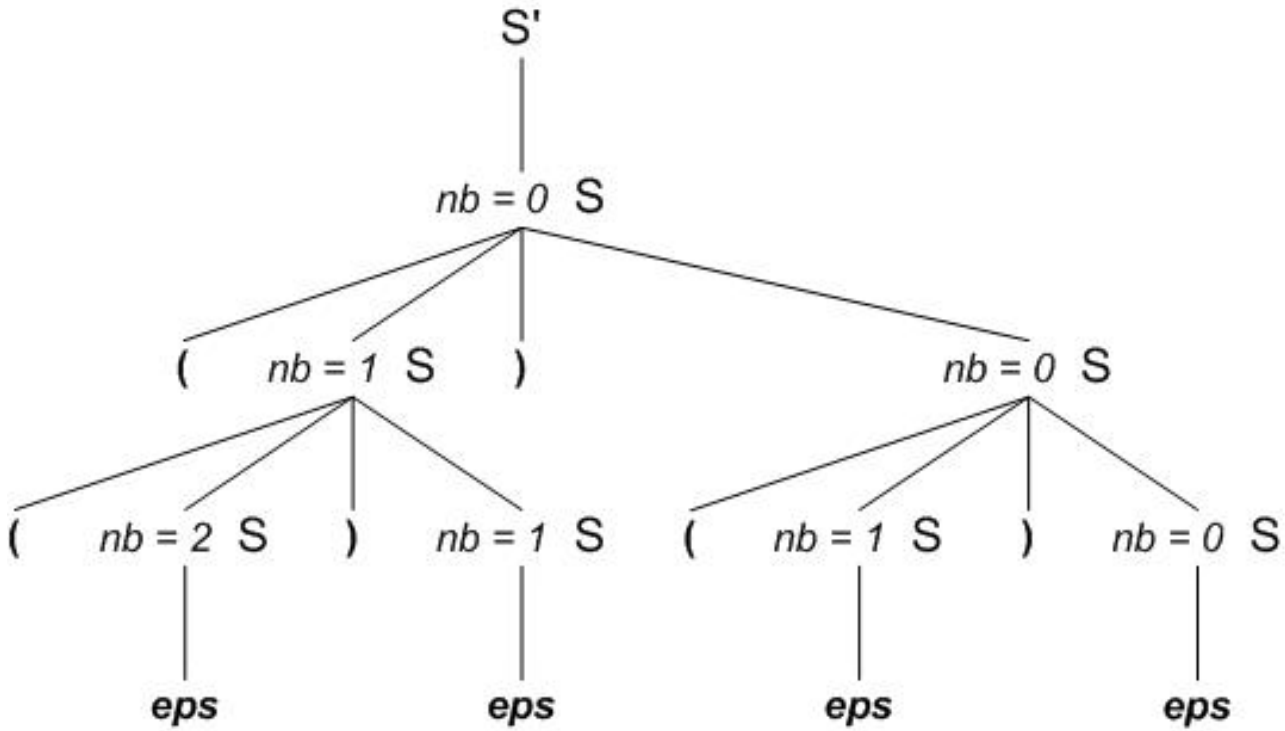
Analyse de 9 - 5 + 2

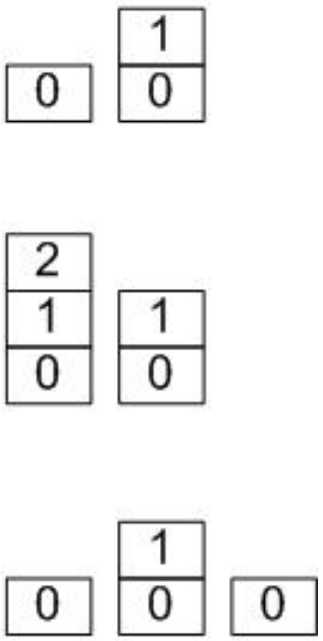


Analyse descendante

Production	Règle sémantique	Traduction
$S' \longrightarrow S\$$	S.nb := 0	p.empiler(0)
$S \longrightarrow (S)S$	S_1.nb := S.nb + 1 S_2.nb := S.nb	tmp = p.depiler() p.empiler(tmp) p.empiler(tmp + 1)
$S \longrightarrow \epsilon$	écrire S.nb	p.depiler()

Analyse de (( )) ( )


S'
nb = 0 S
(nb = 1 S)
nb = 0 S
(nb = 2 S)
nb = 1 S
(nb = 1 S)
nb = 0 S
eps
eps
eps
eps


1
0
0
2
1
0
1
0
0
1
0
0
0

Conclusion

- L'analyse descendante se prête bien à la traduction de définitions L-attribuées n'ayant que des attributs hérités puisque l'arbre syntaxique est créé de la racine vers les feuilles. Il suffit dans certains cas d'empiler et de dépiler les attributs hérités.
- L'analyse ascendante se prête bien à la traduction de définitions S-attribuées puisque l'arbre syntaxique est créé des feuilles vers la racine. Il suffit dans certains cas d'empiler et de dépiler les attributs synthétisés.

Principe

On insère les actions dans les règles de production comme suit :

- Un attribut hérité attaché à une variable en partie droite de règle est calculé par une action intervenant avant cette variable.
- Un attribut synthétisé attaché à la variable en partie gauche est calculé après que tous les arguments dont il dépend ont été calculés. L'action est en général placée en fin de règle.
- Dans les définitions L-attribuées une action ne peut faire référence à un attribut d'un symbole placé à sa droite.

Définition L-attribuée - 1

Production	Règle sémantique	Traduction
$E \longrightarrow T$ R	R.he := T.val E.val := R.s	rhe = t e = rs
$R \longrightarrow +T$ R	R_1.he := R.he + T.val R.s := R_1.s	rhe = rhe + t
$R \longrightarrow -T$ R	R_1.he := R.he - T.val R.s := R_1.s	rhe = rhe - t
$R \longrightarrow \epsilon$	R.s := R.he	rs = rhe
$T \longrightarrow nb$	T.val := nb.val	t = n

Schéma :

$E \longrightarrow T$ {rhe = t ;} R {e = rs ;}

$R \longrightarrow +T$ {rhe = rhe + t ; } R

$R \longrightarrow -T$ {rhe = rhe - t ; } R

$R \longrightarrow \epsilon$ {rs = rhe ; }

$T \longrightarrow nb$ {t = n ;}

Définition L-attribuée - 2

Production	Schéma de Traduction
$E \longrightarrow TR$	$E \longrightarrow T$ {rhe = t ;} R {e = rs ; }
$R \longrightarrow +TR$	$R \longrightarrow +T$ {rhe = rhe + t ; } R
$R \longrightarrow -TR$	$R \longrightarrow -T$ {rhe = rhe - t ;} R
$R \longrightarrow \epsilon$	$R \longrightarrow \epsilon$ {rs = rhe ;}
$T \longrightarrow nb$	$T \longrightarrow nb$ {t = n ;}

Ordre des actions lors de l'analyse de : 10 - 4 + 5 ?

Ordre

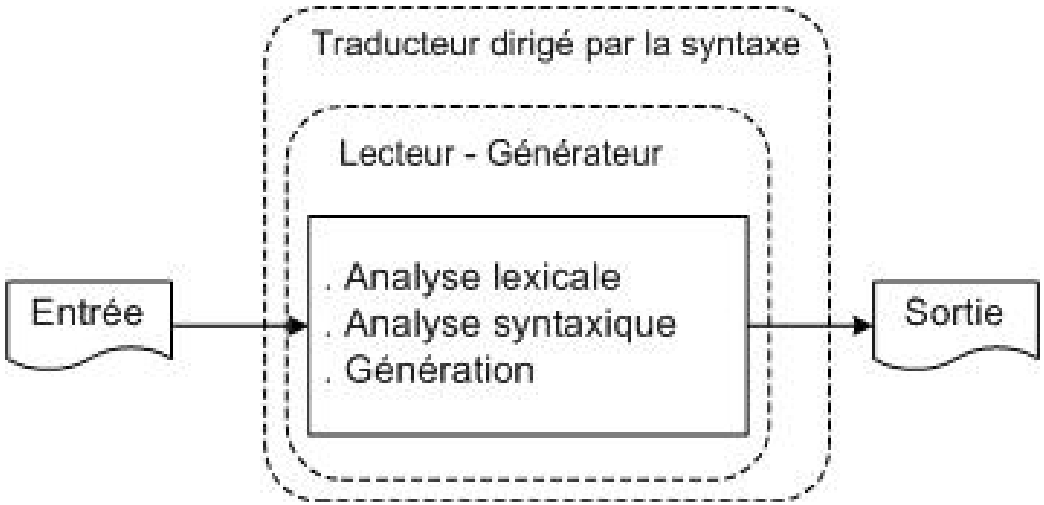
t ⟵ nb (10)
rhe ⟵ t
t ⟵ nb (4)
rhe ⟵ rhe - t
t ⟵ nb (5)
rhe ⟵ rhe + t
rs ⟵ rhe
e ⟵ rs

Sommaire

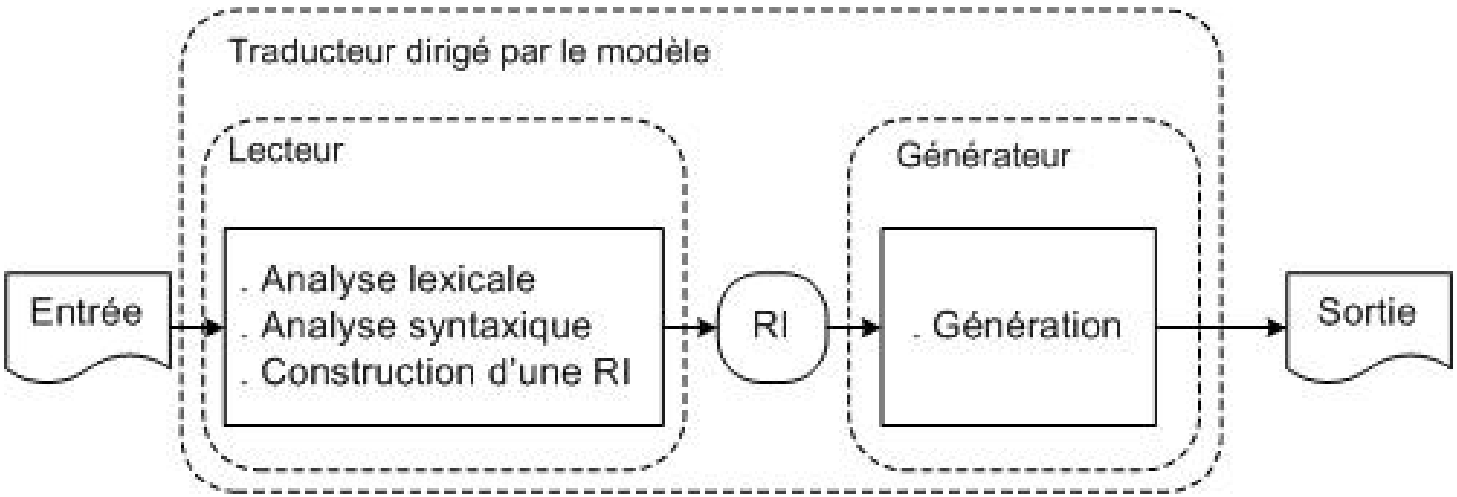
Utilisations

- Transformation d'arbres : il s'agit de construire un nouvel arbre à partir d'un autre.
  - Analyse formelle : Dérivation
  - Un analyseur d'arbre (AST walker) peut construire un autre arbre AST.
- Transformation de formalismes :
  - Formalismes ayant le même pouvoir de représentation
  - Traduction dirigée par la syntaxe.
  - Traduction dirigée par le modèle.
- Traitement des erreurs

Traduction dirigée par la syntaxe



Traduction dirigée par le modèle



Dérivation

Expression initiale : $2x^3 + 4x^2 + 9x + 21$

Expression dérivée : $6x^2 + 8x + 9$

Arbre initial :
(+ (+ (+ (* 2 (^ x 3)) (* 4 (^ x 2))) (* 9 x)) 21)

Arbre obtenu :
(+ (+ (+ (* 6 (^ x 2)) (* 8 (^ x 1))) 9) 0)

Arbre simplifié :
(+ (+ (* 6 (^ x 2)) (* 8 1)) 9)

Sommaire

Introduction

- La reconnaissance des erreurs par un parser traitant un programme intervient soit :
  - durant le développement de la grammaire et les messages d'erreurs servent à la mettre au point ;
  - après le développement de la grammaire et les messages sont destinés aux utilisateurs du parser.
- Les libellés des messages doivent être informatifs et précis ;
- Ils doivent indiquer une solution possible au problème décelé ;
- La quantité des messages doit être mesurée.

Principes

- Un parser ne doit pas s'arrêter à la première erreur sur le texte entré ;
- en cas d'erreur de syntaxe, le parser doit déterminer une action lui permettant de continuer son processus ;
- la récupération d'erreurs est le processus consistant, en cas d'erreur de syntaxe, à modifier le texte entré (en ajoutant ou en retirant des caractères) jusqu'à ce que le parser puisse reprendre l'analyse dans un état connu.

Type d'erreurs

- Erreurs de syntaxe habituelles :
  - 1. erreur ou omission de tokens ;
  - 2. aucune alternative d'une règle satisfaite
  - 3. sortie prématurée d'une boucle (...)+

Exemple

```
ID : ('a'..'z' |'A'..'Z' )+ ;
INT : '0'..'9' + ;
prog : stat+ ;
stat :  expr ';' | assign ';' ;
assign : ID '=' expr ';' ;
expr : multExpr (('+' |'-' ) multExpr)* ;
multExpr : atom ('*' atom)* ;
atom:
  INT
| '(' expr ')'
;
```

Erreur - 1

Entrée : (3) ; (7) ;
=> syntaxe correcte

Entrée : (3 ; 4+5 ;
=> syntaxe incorrecte : trouvé ’ ;’ manque ’)’ - Cas 1

Le parser signale l’erreur, continue l’analyse et reconnaît la seconde expression.
Le parser reconnaît (3 et 4+5 ;

Erreur - 2

Entrée : (3 (3+10) ;
=> syntaxe incorrecte : trouvé ’(’ manque ’)’ - Cas 1

Il manque aussi ’ ;’
Le parser signale une erreur et non pas deux, continue l’analyse après les erreurs et reconnaît la seconde expression.
Le parser reconnaît (3+10) ;

Erreur - 3

Entrée : 25 + ;

=> syntaxe incorrecte : aucune alternative dans la règle *atom* n'est satisfaite - Cas 2

Le parser récupère l'erreur en cherchant le prochain symbole qui peut suivre la règle *atom* ou une règle qui l'a invoquée (*expr*).

' ;' est un symbole viable après la règle *atom* . Le parser ne consomme pas de tokens et sort simplement de la règle.

Erreur - 4

Entrée : fichier vide

=> syntaxe incorrecte : la boucle stat+ n'est pas satisfaite - Cas 3

Erreur dans le lexer - 1

Entrée : (103) ;
=> syntaxe correcte

Entrée : (1a3) ;
=> syntaxe incorrecte
1 est reconnu par le lexer comme un INT ;
a est reconnu comme ID mais provoque l'erreur ')' attendu et 'a' trouvé.
3 est reconnu comme INT mais provoque l'erreur '=' attendu et '3' trouvé.
erreur ' ;' attendu et ')' trouvé.

Si le caractère erroné peut être compris dans une autre règle lexicale, l'erreur produite est une erreur syntaxique et non une erreur lexicale.

Erreur dans le lexer - 2

Un caractère est invalide si aucune alternative dans un règle lexicale ne peut être satisfaite.
Entrée : (1&3) ; => caractère invalide '&' indiquant une erreur lexicale et non syntaxique.

3 est reconnu comme INT mais provoque l'erreur ')' attendu et '3' trouvé.

Erreur dans le lexer - 3

Entre les tokens corrects, le lexer récupère l'erreur en supprimant le caractère erroné et en recherchant le prochain token valide. Une fois l'erreur récupérée, le lexer envoie le prochain token au parser.

L'erreur lexicale peut générer une erreur syntaxique immédiate.

=> 1&3 a été entré à la place d'un nombre tel que 103. Le lexer envoie deux tokens INT, l'un pour 1 et l'autre pour 3. Le second token est syntaxiquement mal placé.

Récupération automatique d'erreurs

- Si possible, un parser insère ou supprime un symbole lorsqu'il rencontre des erreurs.
- Sinon, il avale les symboles jusqu'à ce qu'il trouve un symbole de l'ensemble de resynchronisation et sort alors de la règle.
- L'ensemble de resynchronisation est l'ensemble des symboles qui peuvent logiquement suivre la variable en tête de la règle courante ou de toute règle l'invoquant.
- Il est important de n'émettre qu'un seul message d'erreurs et d'éviter des messages en cascade ou faux. A travers l'usage d'une simple variable booléenne le parser peut éviter d'émettre de tels messages jusqu'à ce qu'un symbole soit reconnu dans une syntaxe correcte.

Exemple

```
INT : '-'?'0'..'9'+ ;
WS : ' ' {skip();} ;
ID : 'a'..'z' ;
stat :( expr ';' | assign ';')+ ;
assign : ID '=' expr ';' ;
expr : multExpr (('+' |'-' ) multExpr)* ;
multExpr : atom ('*' atom)* ;
atom:
    INT
| '(' expr ')'
;
```

Erreur Lexer - Parser

- 10 - 5 ; est reconnu.
- 7-5 ; provoque l'erreur : il manque un ' ;' et -5 est trouvé.
  - Les lexèmes 7 et -5 sont reconnus comme INT et le parser reçoit INT INT.
  - Les règles INT et WS interdisent une bonne lecture.

Solution - 1

- On ne peut laisser le lexer (règle INT) englober les nombres négatifs. Sinon il faudrait imposer la prise en compte des séparateurs dans l'écriture des règles.
- Il faut ajouter une règle au parser capable de reconnaître les moins unaires en plus de la reconnaissance de la soustraction.

Solution - 2

- La règle du parser concernée est celle reconnaissant les nombres : atom.
- Ele doit permettre de reconnaître : 10 - 5 ; 10 - -4 ; (-2) + (-3)

```
atom:
    INT
|   '-' INT
| '(' expr ')'
```

Sommaire

Exemple.
Examen 2008.

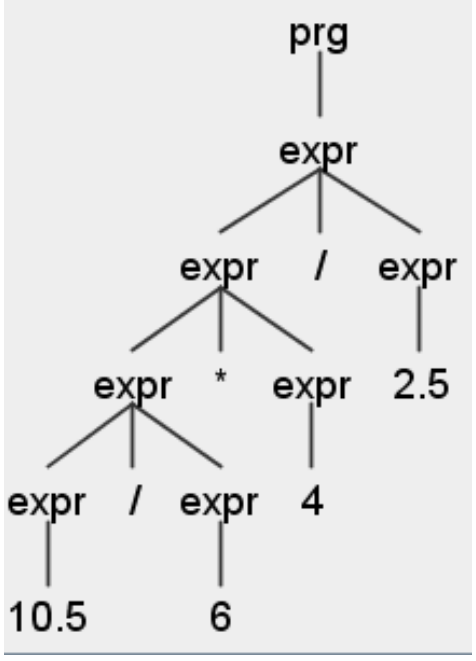
- La grammaire suivante permet d'écrire des expressions comprenant des suites de produits et de quotients, associatives à gauche telles que 10.5 / 6 * 4 / 2.5

```
FLOAT : '0'..'9'+ ('.' '0'..'9'+) ? ;
prg:
  expr
  ;
expr:
      expr '^' expr                 #exp
  |   expr ('*' | '/' ) expr        #mult
  |   expr ('+' | '-' ) expr        #sum
  |   FLOAT                         #float
  |   '(' expr ')'                  #parent
  ;
```

Arbre de dérivation

Analyse descendante.

Expression : 10.5 / 6 * 4 / 2.5 – Calcul : (((10.5 / 6) * 4) / 2.5)



Analyse
Analyse descendante.

Expression : 10.5 / 6 * 4 / 2.5 – Calcul : (((10.5 / 6) * 4) / 2.5)

- Chaque nœud `expr` synthétise la valeur de l'expression à partir du FLOAT pour les nœuds les plus profonds et à partir des sous-expressions sinon

exp	synthétise
FLOAT	10.5 et 6
mult	10.5 / 6 soit 1.75
FLOAT	4
mult	1.75 * 4 soit 7
FLOAT	2.5
mult	7 / 2.5 soit 2.8

Représentation des attributs

- Solution pour représenter les attributs :
  - Une map dont les clés sont des mots clés pour les attributs hérités.
  - Une map dont les clés sont les nœuds de l'arbre pour les attributs synthétisés (plusieurs attributs possibles).

Calcul des attributs

- Les attributs sont calculés par la méthode du visiteur lors de l'exploration d'un nœud.
- Les attributs synthétisés sont calculés après l'appel de la règle à laquelle ils sont attachés.
- Les attributs hérités sont calculés avant l'appel de la règle à laquelle ils sont attachés.

Sommaire

Identificateurs

- Une table des symboles est une structure regroupant les informations sémantiques des identificateurs d'un programme :
  - Variable.
  - Type, Classe.
  - Méthode, fonction.
- Il est possible de construire plusieurs tables de symboles.
- Un dictionnaire de symboles est en général implémenté comme une table de hachage dont les clés sont les noms des symboles et les valeurs sont des structures déterminant les propriétés du symbole.
- La recherche d'un symbole est plus rapide dans une table de hachage.

Propriétés

- Les symboles ont au moins en commun les propriétés suivantes :
  - Un nom ; en général composé de lettres, de chiffres et quelques symboles particuliers comme _. Dans certains cas on peut avoir des signes d'opérateurs (+).
  - Une catégorie : variable, fonction, label, classe, etc.
  - Un type : il peut être défini dynamiquement (Python) ou statiquement (Java, C++).
- On peut représenter chaque catégorie par une classe séparée ayant comme attribut commun le nom et le type.

Portées

- La portée est une région du code très bien délimitée qui groupe des définitions de symboles.
- Par exemple une portée de type fonction groupe les paramètres et les variables locales.
- Portée lexicale ou statique : une portée limitée par une paire de symboles particuliers : { . . . }

Particularité des portées

- Portée statique (la plupart des langages) vs portée dynamique (lisp).
  - portée dynamique : une fonction peut partager les variables de la fonction qui l'a invoquée.
- Beaucoup de portées ont des noms : fonction, classe.
- Les portées sont en général imbriquées : blocs de code.
- Certaines portées permettent des déclarations, des instructions.
- Les symboles dans une portée peuvent être visibles ou non à partir d'autres sections : les champs d'une classe ont des modificateurs de visibilité.

Portées imbriquées

```
int x;          // portée globale
void f() {      // f dans portée globale
  int j;        // j dans la portée f
  {int i;}      // i dans une portée imbriquée
  {int k;}      // k dans une autre portée imbriquée
}
void g() {      // g dans protée globale
  int i;        // i dans la portée g
}
```

Contexte

- Au moment du parsing :
  - on représente une portée par une structure qui contient le dictionnaire des symboles qui y sont déclarés.
  - la définition d'un symbole dans la portée revient à ajouter ce sybole dans le dictionnaire.
  - représenter des portées imbriquées revient à empiler les structures représentant les portées.
- Le contexte d'un symbole correspond à la portée dans laquelle il se trouve utilisé plus éventuellement les portées où elle est imbriquéee.

Opération sur un contexte

- Empilement d'une portée.
- La portée courante est la portée en sommet de pile.
- Définition d'un symbole dans la portée courante.
- Dépilement de la portée courante.

Portées imbriquées - 1

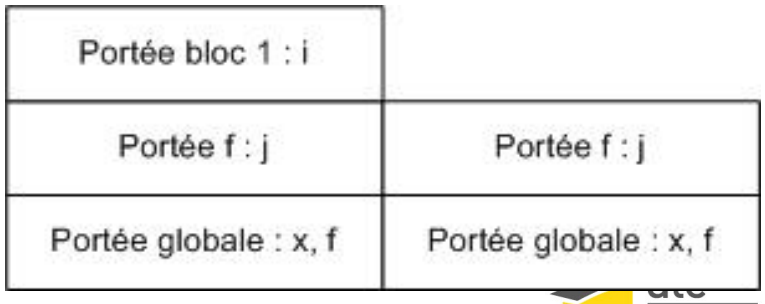
```
int x;        // 1. portée globale
void f() {    // 2. f dans portée globale
  int j;      // 3. j dans la portée f
  {int i;}    // 4. i dans une portée imbriquée 1
  {int k;}    // 5. k dans une portée imbriquée 2
}
void g() {    // 6. g dans protée globale
  int i;      // 7. i dans la portée g
}
```

- 1. : Empilement de la portée globale
- 1.1 : Définition de x dans la portée courante
- 2 : Définition de f dans la portée courante
- 2.1 : Empilement de la portée f

Portées imbriquées - 2

```
int x;        // 1. portée globale
void f() {    // 2. f dans portée globale
  int j;      // 3. j dans la portée f
  {int i;}    // 4. i dans une portée imbriquée 1
  {int k;}    // 5. k dans une portée imbriquée 2
}
void g() {    // 6. g dans la portée globale
  int i;      // 7. i dans la portée g
}
```

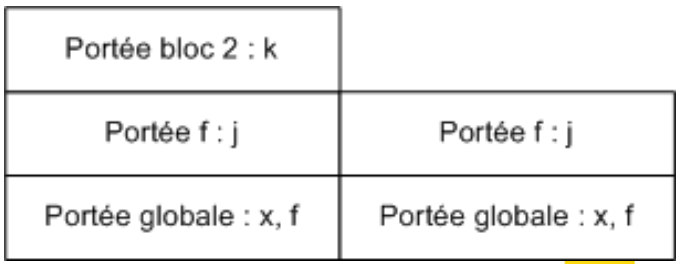
- 3. : Définition de j dans la portée courante (f)
- 4. : Empilement de la portée bloc 1
- 4.1 : Définition de i dans la portée courante
- 4.2 : Dépilement de la portée courante



Portées imbriquées - 3

```
int x;        // 1. portée globale
void f() {    // 2. f dans portée globale
  int j;      // 3. j dans la portée f
  {int i;}    // 4. i dans une portée imbriquée 1
  {int k;}    // 5. k dans une portée imbriquée 2
}
void g() {    // 6. g dans portée globale
  int i;      // 7. i dans la portée g
}
```

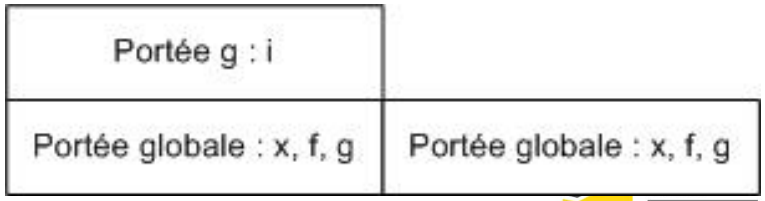
- 5. : Empilement de la portée bloc 2
- 5.1 : Définition de j dans la portée courante
- 5.2 : Dépilement de la portée courante



Portées imbriquées - 4

```
int x;        // 1. portée globale
void f() {    // 2. f dans portée globale
  int j;      // 3. j dans la portée f
  {int i;}    // 4. i dans une portée imbriquée 1
  {int k;}    // 5. k dans une autre portée imbriquée 2
}
void g() {    // 6. g dans portée globale
  int i;      // 7. i dans la portée g
}
```

- 5.3 : Dépilement de la portée courante
- 6. : Définition de g dans la portée courante
- 6.1 : Empilement de la portée g
- 7. : Définition de i dans la portée courante (g)



Recherche d'un identificateur dans un contexte

- Un contexte possède une pile de portées.
- L'insertion d'un identificateur se fait dans le dictionnaire de la portée en sommet de pile.
- La recherche d'un identificateur se fait d'abord dans la portée courante.
- Si l'identificateur n'est pas trouvé, la recherche continue dans la portée sous le sommet de la pile et ainsi de suite jusque la portée où l'identificateur est trouvé.
- Si l'identificateur n'est trouvé dans aucune portée, l'identificateur n'a pas été défini.
- Si l'identificateur a été défini plusieurs fois, la recherche donne un résultat dans la portée la plus récente.

Recherche de x pour printf(x)

```
int x;        // 1. portée globale
void f() {    // 2. f dans portée globale
  int j;      // 3. j dans la portée f
  {int i;     // 4. i dans une portée imbriquée 1
   printf(x);
  }
  {int k;}    // 5. k dans une autre portée imbriquée 2
}
void g() {    // 6. g dans la portée globale
  int i;      // 7. i dans la portée g
}
```

Sommaire

Notion de type

- En général un langage définit des Types Simples :
- entier, booléen, flottant, caractère, void, ...
- et des Types Construits : énumération, structure, tableau, fonction, pointeur, ...

Expression de type - 1

- Si T est une expression de type, alors Tableau(I,T) est une expression de type.
- Tableau est un constructeur de type.
- Tableau(I,T) dénote un tableau dont les éléments sont de type T et les indices de type I (I peut être un intervalle d'entiers).

Expression de type - 2

- Si T est une expression de type, alors Pointeur(T) est une expression de type.
- Pointeur est un constructeur de type.
- Exemple : int *x ;
- déclare un objet x de type "Pointeur(entier)".

Table des types

- On peut rassembler les types d'un programme dans une table de types.
- Une entrée de type contient : constructeur, taille et alignement de ses variables, liste des champs (une structure), dimensions (tableau), etc.
- La table des symboles contient le nom du type ("Pointeur(entier)") qui renvoie à l'index (type_5) de la table des types.

Contrôle de type

- Il peut se faire statiquement par le compilateur, ou dynamiquement lors de l'exécution du programme.
- Le système de typage devrait être capable de dire si deux types sont équivalents, c'est-à-dire s'ils dénotent le même ensemble de valeurs.

Système de typage simple

Déclaration des variables en C :

Production	Règle sémantique
$D \longrightarrow L$ id ;	setType(id.ref,L.type)
$L \longrightarrow L$, id	$L.type = L_1.type$ setType(id.ref,L.type)
$L \longrightarrow$ T	L.type = T.val
$T \longrightarrow$ int	T.val = integer
$T \longrightarrow$ real	T.val = real

Déclaration : **real** x, y, z

Contrôle du type d'une expression

- Ces règles sémantiques permettent de contrôler et d'affecter des types aux expressions arithmétiques.
- Règle : Si les deux opérandes des opérateurs arithmétiques d'addition, de soustraction et de multiplication sont de type entier (resp de type flottant), alors l'expression est de type entier (resp flottant).
- Lorque la règle échoue, la situation peut entraîner des conversions de type.

Production	Règle sémantique
$E \longrightarrow id$	E.type = rechercherType(id.ref)
$E \longrightarrow E + T$	E.type = T.type

- En général on ne donne pas de construction de type pour une instruction.

Equivalence de type

- Le problème est de déterminer avec exactitude si deux types sont équivalents.
- Deux types sont équivalents si l'on peut substituer un des deux types par l'autre sans altérer le résultat du programme.
- L'équivalence de types se fait par nom ou par structure (en pratique n'est plus utilisée).

Equivalence par nom

- Si deux types ont le même nom, ils sont équivalents.
- Le nom d'un type peut être donné par le programmeur ou par le compilateur.
  - type t1 = tableau [int] de int ;
  - type t2 = tableau [int] de int ;
  - type t3 = t2 ;
- t1, t2, t3 sont équivalents.

Equivalence par structure

- Dans le cas général, le problème d'équivalence structurelle est plus difficile car
  - des expressions de types peuvent contenir des noms de types qui ne sont pas des types de base ;
  - l'utilisation des pointeurs peut rendre cyclique la représentation d'un type.

Equivalence structurelle

```
type t5 = structure{
 a: int;
 p: pointeur sur t5;
}

type t6 = structure{
 a: int;
 p: pointeur sur structure {
  a: int;
  p: pointeur sur t5;
 }
}
```

Conversion de type

- La conversion de type s'effectue car les compilateurs essayent de générer des codes effectuant des calculs entre données de types différents.
- Quand l'équivalence de type échoue, le compilateur tente de corriger l'erreur et continue d'analyser le programme.

Conversion implicite

- Les règles sont définies par le langage
  - Ex : flottant + entier = flottant.
  - Un flottant est construit à partir de l'entier pour réaliser l'opération.
- La conversion implicite ne doit pas entraîner de perte d'information.

Conversion explicite

- C'est un mécanisme prévu par certains langages pour indiquer (forcer) le type d'une expression dans un programme.
- Forme habituelle : (type) expr
- Contrairement à la conversion implicite, le transtypage ne conduit pas à une transformation de la donnée.