

## Plan de l'UE

### Introduction

1. Analyse lexicale
  2. Grammaires algébriques
  3. **Analyse syntaxique (descendante, ascendante)**
  4. Analyse sémantique
  5. Production de code
- Conclusion - bilan

1

## Analyseurs syntaxiques ascendants

- Type de grammaire : LR(k)
  - Toujours sous forme d'une grammaire augmentée (par ajout de la règle  $S' \rightarrow S$ )
- Parser LR(k) :
  - Left-to-right scanning : on lit l'entrée de gauche à droite
  - Rightmost derivation in reverse : on construit une dérivation droite en partant du mot à analyser
  - La prédiction utilise k lettres du mot d'entrée
- Différentes analyses ascendantes :
  - Analyse LR(0)
  - Analyse SLR(1) - « Simple LR(1) »
  - Analyse LR(1)**
  - Analyse LALR(1) – optimisation de LR(1)**
  - Résumé des analyses LR(k)

2

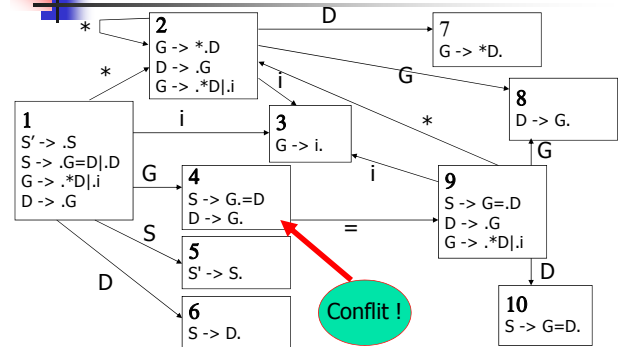
## G2 est-elle LR(0), SLR(1) ?

Soit la grammaire augmentée G2 :

$S' \rightarrow S$	Suivant( $S'$ )={ $\$$ }
$S \rightarrow G = D \mid D$	Suivant( $S$ )={ $\$$ }
$G \rightarrow * D \mid i$	Suivant( $G$ )={ $=$ , $\$$ }
$D \rightarrow G$	Suivant( $D$ )={ $\$$ , $=$ }

3

## Automate LR(0) pour G2



4

## Table Action SLR(1) pour grammaire G2

	=	*	i	\$
1	erreur	d 2	d 3	erreur
2	erreur	d 2	d 3	erreur
3	r G->i	erreur	erreur	r G->i
4	<b>d 9</b> <b>r D-&gt;G</b>	erreur	erreur	r D->G
5	erreur	erreur	erreur	<b>accepter</b>
6	erreur	erreur	erreur	r S->D
7	r G->*D	erreur	erreur	r G->*D
8	r D->G	erreur	erreur	r D->G
9	erreur	d 2	d 3	erreur
10	erreur	erreur	erreur	r S->G=D

Conflit !

5

## Conflits SLR(1)

- La grammaire G2 n'est ni LR(0), ni SLR(1), car le conflit décalage/réduction de l'état 4 se retrouve dans la table Action SLR(1).
- Les états de l'automate LR(0) ne contiennent donc pas assez d'informations pour remarquer ce fait et l'information sur les *Suivant* est également insuffisante pour résoudre le conflit.

-> Pour résoudre ce type de conflit, l'idée est d'enrichir les règles pointées des états avec des symboles de prévision (terminaux ou  $\$$ ) -> *analyse LR(1)*.

6

## Analyse LR(1)

Quand l'analyse SLR(1) n'est pas suffisante  
(présence de conflits)

➡ Analyse LR(1)

**Principe :** enrichir les règles pointées des états  
avec un symbole de prévision (terminal ou \$).

7

## Automate LR(1) : définitions préliminaires

**Définition :** On appelle *règle pointée étendue* toute règle pointée accompagnée d'un symbole de prévision  $a \in (V_T \cup \{\$, \epsilon\})$ , de la forme :  $X \rightarrow \alpha.\beta.a$

On notera  $X \rightarrow \alpha.\beta.a|b$   
lorsque, dans un même état, on a  $X \rightarrow \alpha.\beta.a$  et  $X \rightarrow \alpha.\beta.b$

**Définition :** Un ensemble de règles pointées étendues  $E$  est dit *saturé* si, lorsque  $X \rightarrow \beta.Y.a$  appartient à  $E$  et  $Y \in V_N$ , alors pour toute règle  $Y \rightarrow \alpha$  de la grammaire,  $Y \rightarrow \alpha.b$  avec  $b \in \text{Premier}(Ya)$  appartient à  $E$ .

**Définition :** Pour un ensemble de règles pointées étendues  $E$ , on note  $\text{Saturation}(E)$  le plus petit ensemble saturé  $E'$  contenant  $E$ .

8

## Automate LR(1) : définition

### Etats de l'automate :

Les états  $Q$  de l'automate LR(1) sont des ensembles saturés de règles pointées étendues. Ils sont souvent appelés *collections LR(1)* (ou *LR(1)-items*).

L'état initial de l'automate est  $\text{Saturation}(\{S' \rightarrow .S, \$\})$ .

### Transitions de l'automate :

Pour un état (ensemble saturé de règles pointées étendues)  $E$  et  $x \in (V_T \cup V_N)$ , on définit l'état  $\Delta(E, x)$  comme :

$\text{Saturation}(\{X \rightarrow \beta.x.\gamma.a \mid X \rightarrow \beta.x.\gamma.a \in E\})$ .

9

## Exemple de saturation LR(1) sur grammaire G2

### Saturation( $\{S' \rightarrow .S, \$\}$ ) :

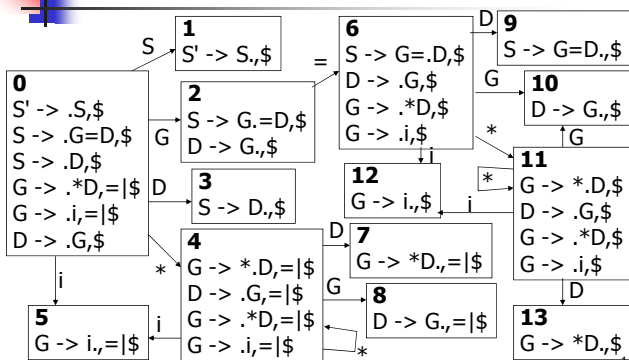
$S' \rightarrow S$   
 $S \rightarrow G=D \mid D$   
 $G \rightarrow *D \mid i$   
 $D \rightarrow G$

$\text{Premier}(G) = \{*, i\}$   
 $\text{Premier}(D) = \text{Premier}(G) = \{*, i\}$   
 $\text{Premier}(S) = \text{Premier}(S') = \{*, i\}$

$S' \rightarrow .S, \$$   
 $S \rightarrow .G=D, \$$   
 $S \rightarrow .D, \$$   
 $G \rightarrow .*D, =$   
 $G \rightarrow .i, =$   
 $D \rightarrow .G, \$$   
 $G \rightarrow .*D, \$$   
 $G \rightarrow .i, \$$

10

## Automate LR(1) pour G2



## Tables LR(1)

- Table Successeur LR(1) :** pour tout  $q \in Q$  et  $X \in V_N$ , si  $\Delta(q, X) = q'$  alors mettre  $q'$  dans la case  $(q, X)$ . (idem LR(0))
- Table Action LR(1) :**
  - pour tout  $a \in V_T$ ,  $q \in Q$ , si  $\Delta(q, a) = q'$  alors mettre "d q'" dans la case  $(q, a)$ ,
  - pour toute règle pointée complète étendue  $X \rightarrow \beta..a$  de  $q \in Q$ , avec  $X \neq S'$ , mettre "r  $X \rightarrow \beta$ " dans la case  $(q, a)$ ,
  - mettre "accepter" dans la case  $(q, \$)$  où  $q$  est l'état contenant  $S' \rightarrow .S, \$$ ,
  - mettre "erreur" dans toutes les cases encore vides.

-> Si la table Action contient exactement une action par case, alors la grammaire est LR(1).

12

## Table Action LR(1) pour G2

	*	i	=	\$
0	d4	d5		
1				accepter
2			d6	r D->G
3				r S->D
4	d4	d5		
5			r G->i	r G->i
6	d11	d12		
7			r G->*D	r G->*D
8			r D->G	r D->G
9				r S->G=D
10				r D->G
11	d11	d12		
12				r G->i
13				r G->*D

13

## Grammaires SLR(1), LR(1) et...

- Toute grammaire SLR(1) est LR(1)
  - La méthode LR(1) nécessite un très grand nombre d'états et donc consomme beaucoup d'espace mémoire
  - La méthode SLR(1) nécessite moins d'états mais limite trop la forme de la grammaire utilisable
- > Une méthode plus puissante que SLR(1) mais dont le nombre d'états de l'automate est voisin de SLR(1) ?  
-> *analyse LALR(1)*.

14

## Analyse LALR(1) : principe

**Remarque :** l'explosion du nombre d'états de l'automate LR(1) provient de la prise en compte du symbole de prévision (différents symboles possibles pour une même règle pointée).

**Idee de l'analyse LALR(1) "Look-Ahead LR(1)" :** regrouper les états qui sont identiques modulo les symboles de prévision.

**Exemple :** on peut regrouper les états 4 et 11, ainsi que 8 et 10 de l'automate LR(1) pour G2.

15

## Analyse LALR(1) : définitions

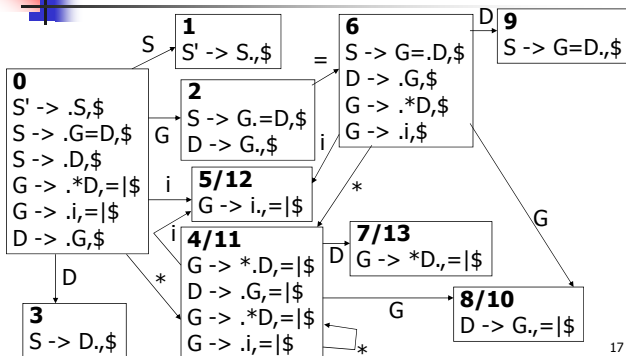
**Définition :** Lorsque deux états de l'automate LR(1) ne diffèrent que sur les caractères de prévision, on dit qu'ils ont le même cœur LR(0).

**Définition :** L'automate LALR(1) est construit à partir de l'automate LR(1) en fusionnant les états ayant le même cœur LR(0). En fusionnant, on crée un état dont les règles sont l'union des règles des deux états fusionnés. Les transitions de l'automate sont ensuite simplement mises à jour en fonction des fusions effectuées.

**Remarque :** Dans l'automate LR(1), les cœurs LR(0) des états sont précisément les états de l'automates LR(0). L'automate LALR(1) a donc exactement le même nombre d'états que l'automate LR(0).

16

## Automate LALR(1) pour G2



17

## Tables LALR(1)

- **Table Successeur LALR(1) :** pour tout  $q \in Q$  et  $X \in V_N$ , si  $\Delta(q,X) = q'$  alors mettre  $q'$  dans la case  $(q,X)$ . (idem LR(1) mais à partir de l'automate LALR(1))
  - **Table Action LALR(1) :**
    - pour tout  $a \in V_T$ ,  $q \in Q$ , si  $\Delta(q,a) = q'$  alors mettre "d  $q'$ " dans la case  $(q,a)$ ,
    - pour toute règle pointée complète étendue  $X \rightarrow \beta$ ,  $a$  de  $q \in Q$ , avec  $X \neq S'$ , mettre "r  $X \rightarrow \beta$ " dans la case  $(q,a)$ ,
    - mettre "accepter" dans la case  $(q,\$)$  où  $q$  est l'état contenant  $S'$  ->  $S, \$$ ,
    - mettre "erreur" dans toutes les cases encore vides.
- (idem LR(1) mais à partir de l'automate LALR(1))
- > Si la table Action contient exactement une action par case, alors la grammaire est LALR(1).

18

## Table Action LALR(1) pour G2

	*	i	=	\$
0	d4/11	d5/12		
1				<b>accepter</b>
2			d6	r D->G
3				r S->D
4/11	d4/11	d5/12		
5/12			r G->i	r G->i
6	d4/11	d5/12		
7/13			r G->*D	r G->*D
8/10			r D->G	r D->G
9				r S->G=D

19

## Grammaires LR(1) et LALR(1)

- Toute grammaire LALR(1) est LR(1)
- Il existe des grammaires LR(1) qui ne sont pas LALR(1) !
- Toute grammaire LR(1), ou a fortiori LALR(1), est non-ambiguë
- L'analyse LALR(1) est plus efficace que l'analyse LR(1), avec une table LALR(1) de la taille de la table LR(0). Notons qu'il existe une méthode directe permettant de construire l'automate et les tables LALR(1) (sans construire l'automate LR(1))

20

## Résumé Analyses LR(k)

- Toute grammaire LR(0) est SLR(1)
- Toute grammaire SLR(1) est LR(1)
- Toute grammaire LALR(1) est LR(1)
- Toute grammaire LR(k), SLR(k) ou LALR(k) est non-ambiguë
- Analyse LR(k),  $k > 1$  ? -> (trop) coûteux !

21

## Modes d'un compilateur idéal : parser / erreur

- Au départ : mode parser
- Détection d'une erreur -> mode erreur
  - Localisation et diagnostic de l'erreur (trouver la faute à son origine)
  - Correction de l'erreur si possible
  - -> retour au mode parser
- Continuer l'analyse pour découvrir le plus possible d'erreurs

22

## Traitement des erreurs : quatre méthodes

- Mode panique : *sans correction*, avec ou sans récupération par restart ou continuation sur symbole de synchronisation (mot-clé *error* en JCup)
- Avec *correction au niveau du syntagme* en utilisant des routines de traitement d'erreurs dans la table d'analyse
  - approprié pour les erreurs de programmation classiques
- Avec *correction locale* en définissant des règles spécifiques « cas d'erreur » dans la grammaire
- Avec *correction automatique globale* au niveau du programme :
  - l'analyseur « devine » les intentions du programmeur...
  - technique trop coûteuse pour être utilisée

23

## Mode panique

- Mode panique : localisation vague, sans correction, stop ou récupération
- Les différents modes de récupération :
  - Par continuation : on clôt le sous-arbre en cours
  - Par restart : on analyse quand même des sous-arbres de l'arbre en cours
- La localisation/correction dépend du mode de récupération

24

## Récupération par continuation

Soit X le non-terminal en cours d'analyse lorsque l'erreur est détectée.

- Récupération par continuation :
  - Sur un symbole terminal de X (last/dernier)
    - End pour bloc\_inst, ) pour liste\_param, ...
    - Pb : il n'existe pas toujours de marqueur de fin
  - Sur un premier terminal après l'occurrence de X (follow/suivant), lié au contexte
    - ; pour instruction, then ou do pour une condition, ...
    - Pb : risque de propagation d'erreurs sur construction récursive

25

## Récupération par restart

Soit X le non-terminal en cours d'analyse lorsque l'erreur est détectée.

- Récupération par restart :
  - Sur premier symbole d'un non-terminal accessible à partir de X (utilise first/premier)
    - While ou for pour bloc\_inst, ...
    - Pb : il n'existe pas toujours de symbole initial
  - Sur un symbole prédécesseur d'un non-terminal accessible à partir de X (utilise precede, last)
    - ; pour une instruction, = pour une expression dans une affectation, ...

26

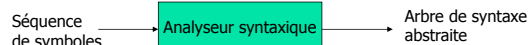
## 3.4. Génération d'analyseurs syntaxiques



- Génération de parsers descendants (LL) :
  - Ecrire son propre parser : facile bien que long, pas très robuste, difficile à modifier
  - Automatiquement avec le générateur de parsers LL JavaCC pour Java
- Génération de parsers ascendants (LR) :
  - Ecrire son propre parser : facile bien que long, pas très robuste, difficile à modifier
  - Automatiquement avec la famille de générateurs de parsers LR Yacc : Yacc et Bison pour C, *JCup* pour Java

27

## Conclusion sur l'analyse syntaxique



Analyseur syntaxique = automate à pile reconnaissant un langage algébrique en implantant :

- soit une analyse descendante LL (la grammaire doit être LL),
- soit une analyse ascendante LR (plus général au niveau grammaires).

28

# Plan de l'UE

## Introduction

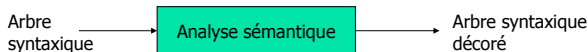
1. Analyse lexicale
  2. Grammaires algébriques
  3. Analyse syntaxique (descendante, ascendante)
  4. **Analyse sémantique**
  5. Production de code
- Conclusion - bilan

1

## 4. Analyse sémantique

2

## Analyse sémantique : objectif



- Vérification de propriétés sémantiques statiques des programmes (contextuelles)
  - Vérification de types, analyse de pointeurs, évaluation statique d'expressions, déclaration de variables...
- > l'analyse sémantique occupe une place très importante et complexe dans un compilateur

3

## Analyse sémantique : plan

- 4.1. Définition dirigée par la syntaxe (grammaire attribuée)
- 4.2. Arbre syntaxique décoré
- 4.3. Attributs hérités ou synthétisés
- 4.4. Schéma de traduction dirigé par la syntaxe
- 4.5. Graphe de dépendances
- 4.6. Évaluation des attributs
- 4.7. Exemples

4

## 4.1. Définition dirigée par la syntaxe (DDS)

**Définition :** Une *définition dirigée par la syntaxe* (ou *grammaire attribuée*) est la donnée d'une grammaire et de son ensemble de règles sémantiques :

- chaque symbole (terminal ou non) de la grammaire possède un *ensemble d'attributs*,
- chaque règle de production de la grammaire possède un *ensemble de règles sémantiques* (suites d'instructions algorithmiques) permettant de calculer les valeurs des attributs des symboles de la production.

**Notation :** On note  $X.a$  l'attribut  $a$  du symbole  $X$ . S'il y a plusieurs symboles  $X$  dans une production, on note  $X_{(0)}$  s'il est en partie gauche, et sinon  $X_{(1)} \dots X_{(n)}$  de gauche à droite.

5

## Exemple de DDS

```
E' -> E           // Résultats dans E.val et E.nbpar
E -> E + T        {E(0).val = E(1).val + T.val;
                  E(0).nbpar = max(E(1).nbpar, T.nbpar)}
E -> T            {E.val = T.val; E.nbpar = T.nbpar}
T -> (E)          {T.val = E.val; T.nbpar = E.nbpar + 1}
T -> i            {T.val = lexical.val(i); T.nbpar = 0}
```

Attributs : *val* (valeur de l'expression arithmétique) et *nbpar* (niveau maximal de parenthésage)

6

## 4.2. Arbre syntaxique décoré

**Définition :** Un *arbre syntaxique décoré* (ou *arbre attribué*) est un arbre syntaxique sur les nœuds duquel sont rajoutées les valeurs de ses attributs.

### Remarques :

- Une DDS n'implique aucun ordre pour l'exécution des actions sémantiques, et donc l'évaluation des attributs.
- Mais, utiliser des attributs impose cependant un ordre !

7

## 4.3. Attributs hérités, attributs synthétisés

**Définition :** Un attribut est dit *hérité* lorsqu'il est calculé à partir des attributs du non-terminal de la partie gauche, et éventuellement des attributs d'autres non-terminaux de la partie droite.

**Définition :** Une grammaire attribuée n'ayant que des attributs hérités et telle que ces attributs ne dépendent pas des frères droits est dite *L-attribuée*.

**Définition :** Un attribut est dit *synthétisé* lorsqu'il est calculé pour le non-terminal de la partie gauche en fonction des attributs des non-terminaux de la partie droite (et éventuellement d'autres attributs de lui-même).

**Définition :** Une grammaire attribuée n'ayant que des attributs synthétisés est dite *S-attribuée*.

8

## Initialisation des attributs synthétisés

Si un non-terminal  $X$  possède un attribut synthétisé  $s$  et une règle de production de la forme :

- $X \rightarrow \alpha$  avec  $\alpha \in V_T^+$ , alors la valeur de  $s$  est *en général* fournie par l'analyseur lexical

$E \rightarrow i \quad \{E.val = lexical.val(i)\}$

- $X \rightarrow \varepsilon$ , alors la valeur de  $s$  est *souvent* l'élément neutre de l'opération servant à calculer  $s$  (s'il existe)

$S \rightarrow aSb \quad \{S_{(0)}.nba = S_{(1)}.nba + 1\}$   
 $S \rightarrow SS \quad \{S_{(0)}.nba = S_{(1)}.nba + S_{(2)}.nba\}$   
 $S \rightarrow \varepsilon \quad \{S_{(0)}.nba = 0\}$

avec  $nba$  : entier attribut synthétisé de  $S$

9

## Initialisation des attributs hérités

L'axiome de la grammaire ne peut pas avoir d'attribut hérité (puisqu'il ne peut pas être initialisé)

$S \rightarrow aaSb \quad \{S_{(1)}.nba = S_{(0)}.nba + 2\}$   
 $S \rightarrow \varepsilon \quad \{Afficher(S.nba)\}$

avec  $nba$  : entier attribut hérité de  $S$

Pour initialiser cet attribut, on peut créer un nouvel axiome  $S'$  et une règle  $S' \rightarrow S$  dont l'action sémantique permet l'initialisation

$S' \rightarrow S \quad \{S.nba = 0\}$

10

## 4.4. Schéma de traduction dirigé par la syntaxe

**Définition :** Un *schéma de traduction dirigé par la syntaxe* (STDS ou TDS) est une DDS dans laquelle l'ordre d'exécution des actions sémantiques est imposé.

### Exemple (avec attribut hérité) :

$S' \rightarrow \{S.nba = 0;\} S \{afficher(\text{« Fin »});\}$   
 $S \rightarrow a \{S_{(1)}.nba = S_{(0)}.nba + 1;\} S$   
 $\quad \mid \varepsilon \{afficher(S.nba);\}$

11

## Schéma de traduction dirigé par la syntaxe

### Exemple (avec attribut synthétisé) :

$S' \rightarrow S \{afficher(S.nba);\}$   
 $S \rightarrow aS \{S_{(0)}.nba = S_{(1)}.nba + 1;\}$   
 $\quad \mid \varepsilon \{S.nba = 0;\}$

### !! Contre-exemple (avec variable globale) :

~~$S' \rightarrow \{nba = 0;\} S \{afficher(nba);\}$~~   
 ~~$S \rightarrow a \{nba++;\} S \mid \varepsilon$~~

12

## Exemple : déclarations d'identificateurs (1)

Soit la grammaire G :

```
L -> T D
T -> int | float
D -> id , D | id
```

Avec les attributs :

- type : *Type* (attribut synthétisé de T)
- typeh : *Type* (attribut hérité de D)
- table : *Table* (attribut synthétisé de D et L), le type *Table* étant défini comme un ensemble de couples (*identificateur*, *type*).

13

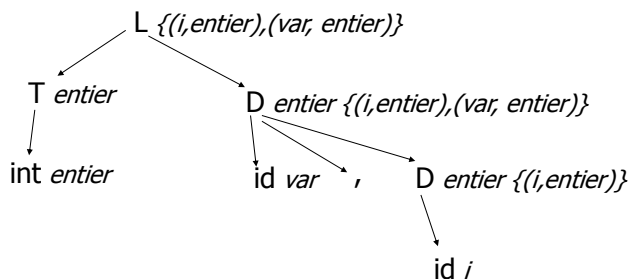
## Exemple : calcul de la table des identificateurs (1)

La grammaire attribuée G' associée à G pour calculer la table des identificateurs est la suivante :

```
L -> T D    {D.typeh = T.type; L.table = D.table;}
T -> int    {T.type = « entier »}
T -> float  {T.type = « réel »}
D -> id , D {D1.typeh = D0.typeh;
              D0.table = add((lexical.va(id), D1.typeh), D1.table);}
D -> id     {D.table = add((lexical.va(id), D.typeh),
                           table_vide);}
```

14

## Exemple : arbre de dérivation décoré pour « int var, i » (1)



L.table, D.table, D.typeh et T.type

15

## Exemple : déclarations d'identificateurs (2)

Reprenons la grammaire G :

```
L -> T D
T -> int | float
D -> id , D | id
```

Autre solution avec seulement des attributs synthétisés :

- type : *Type* (attribut synthétisé de T)
- table : *Table* (attribut synthétisé de L), le type *Table* étant défini comme un ensemble de couples (*identificateur*, *type*)
- collect : *Ensemble d'identificateurs* (attribut synthétisé de D)

16

## Exemple : calcul de la table des identificateurs (2)

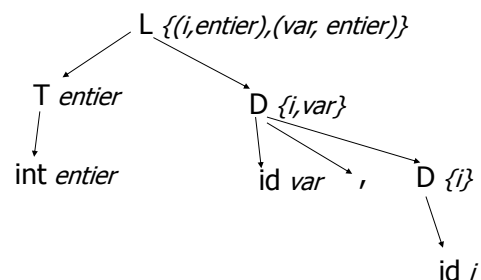
La grammaire *S-attribuée* G'' associée à G pour calculer la table des identificateurs est la suivante :

```
L -> T D    {L.table = MakeTable(T.type, D.collect);}
T -> int    {T.type = « entier »}
T -> float  {T.type = « réel »}
D -> id , D {D0.collect = add(lexical.va(id), D1.collect);}
D -> id     {D.collect = add(lexical.va(id), coll_vide);}
```

Avec MakeTable(type, C) : crée une table de (*identificateur*, *type*) pour chaque identificateur de C

17

## Exemple : arbre de dérivation décoré pour « int var, i » (2)



L.table, D.collect et T.type

18



## 4.5. Graphe de dépendances

**Définition :** On appelle *graphe de dépendances* le graphe orienté représentant les interdépendances entre les divers attributs. Les attributs sont les sommets du graphe. On a un arc de x vers y si et seulement si le calcul de y dépend de celui de x.

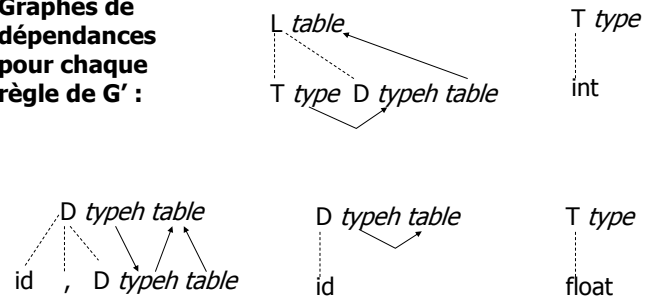
### Remarques :

- On peut construire un graphe de dépendances pour chaque règle de production ou pour un arbre syntaxique.
- Si un graphe de dépendances contient un *cycle* alors l'évaluation des attributs est impossible ;
- sinon, il existe une énumération des couples (*nœud,attribut*) telle que si  $([i],x) \rightarrow ([j],y)$  alors  $([i],x)$  se trouve avant  $([j],y)$  dans l'énumération ; l'ordre d'évaluation est alors le même que l'ordre d'énumération.

19

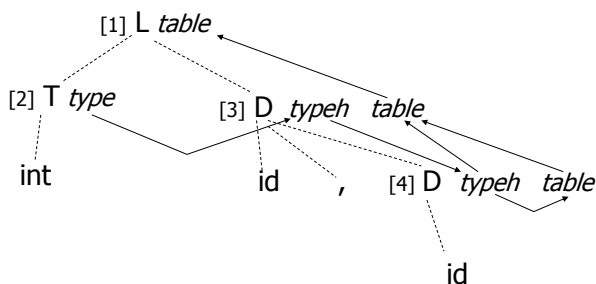
## Graphe de dépendances de $G'$

**Graphes de dépendances pour chaque règle de  $G'$  :**



20

## Graphe de dépendances pour l'arbre de dérivation de « int var, i » par $G'$



### Ordre d'évaluation :

$([2],type), ([3],typeh), ([4],typeh), ([4],table), ([3],table)$  et  $([1],table)$ .

21

## 4.6. Évaluation des attributs

- En toute généralité, cela nécessite un arbre de dérivation et un graphe de dépendances
- Jusqu'ici, nous avons vu des analyses LL et LR qui *parcourent l'arbre de dérivation sans le construire...*
- Doit-on vraiment construire l'arbre (et le graphe) ?
  - Si analyse sémantique après l'analyse syntaxique : oui  
-> coûteux en mémoire/ordre d'évaluation « quelconque »
  - Si analyse sémantique pendant l'analyse syntaxique : non !
- À quelles conditions analyses syntaxique//sémantique ?
  - Pour l'analyse LR -> grammaires S-attribuées
  - Pour l'analyse LL -> grammaires L-attribuées

22

## Calcul d'attributs dans le cadre d'une analyse (ascendante) LR(k)

- Pré-requis :** grammaire S-attribuée
- Principe :** on stocke les valeurs des attributs avec les non-terminaux associés dans la pile d'analyse
- Lors d'une réduction, les attributs des non-terminaux de la partie droite sont dans la pile, donc utilisables pour calculer les attributs du non-terminal à gauche.

23

## Exemple (sur grammaire $G_1$ )

```

E' -> E      {E'.val = E.val;}
E -> E + T   {E(0).val = E(1).val + T.val;}
E -> T       {E.val = T.val;}
T -> T * F   {T(0).val = T(1).val * F.val;}
T -> F       {T.val = F.val;}
F -> i       {F.val = lexical.val(i);}
    
```

-> Calcul de la valeur d'une expression arithmétique

24

## Rappels tables Action et Successeur SLR(1) pour G1

	+	*	i	\$
0	erreur	erreur	d 3	erreur
1	d 5	erreur	erreur	accepter
2	r T->F	r T->F	erreur	r T->F
3	r F->i	r F->i	erreur	r F->i
4	r E->T	d 6	erreur	r E->T
5	erreur	erreur	d 3	erreur
6	erreur	erreur	d 3	erreur
7	r T->T*F	r T->T*F	erreur	r T->T*F
8	r E->E+T	d 6	erreur	r E->E+T

	E'	E	T	F
0		1	4	2
1				
2				
3				
4				
5			8	2
6				7
7				
8				

25

## Analyse du mot « 10+2\*3 »

Pile	Entrée (lexicale)	Règle appliquée
0	i+i*i\$	d3
0 i 3 10	+i*i\$	r F->i, 2 {F.val = lexical.va(i);}
0 F 2 10	+i*i\$	r T->F, 4 {T.val = F.val;}
0 T 4 10	+i*i\$	r E->T, 1 {E.val = T.val;}
0 E 1 10	+i*i\$	d5
0 E 1 + 5 10	i*i\$	d3
0 E 1 + 5 i 3 10 2	*i\$	r F->i, 2 {F.val = lexical.va(i);}

26

## Suite de l'analyse de 10+2\*3

Pile	Entrée (lexicale)	Règle appliquée
0 E 1 + 5 F 2 10 2	*i\$	r T->F, 8 {T.val = F.val;}
0 E 1 + 5 T 8 10 2	*i\$	d6
0 E 1 + 5 T 8 * 6 10 2	i\$	d3
0 E 1 + 5 T 8 * 6 i 3 10 2 3	\$	r F->i, 7 {F.val = lexical.va(i);}
0 E 1 + 5 T 8 * 6 F 7 10 2 3	\$	r T->T*F, 8 {T <sub>(0)</sub> .val = T <sub>(1)</sub> .val * F.val;}
0 E 1 + 5 T 8 10 6	\$	r E->E+T, 1 {E <sub>(0)</sub> .val = E <sub>(1)</sub> .val + T.val;}
0 E 1 16	\$	<b>mot accepté !</b> avec E.val = 16

27

## Schéma de traduction JavaCup

- En JavaCup, les actions sémantiques sont *toujours* évaluées lors de réductions
- D'où une possible réécriture automatique de la grammaire par JavaCup
- Mais attention, cela peut modifier le caractère LR de la grammaire !

28

## Exemple

Soit G :

S' -> S  
S -> {code1} ab  
S -> {code2} ac

- G est LR(0), et donc LR(1)

Soit la grammaire transformée G' de G :

S' -> S  
S -> X ab  
X -> ε {code1}  
S -> Y ac  
Y -> ε {code2}

- G' est-elle LR(0), ou LR(1) ?

29

## G' est-elle LR(0) ou LR(1) ?

0  
S' -> .S  
S -> .Xab  
S -> .Yac  
X -> .  
Y -> .

Conflit LR(0) !

0  
S' -> .S,\$  
S -> .Xab,\$  
S -> .Yac,\$  
X -> .,a  
Y -> .,a

Conflit LR(1) !

-> G' n'est ni LR(0) ni SLR(1) ni LR(1)... mais LR(2) !

30

## Calcul d'attributs dans le cadre d'une analyse (descendante) LL(k)

- **Pré-requis** : grammaire L-attribuée
- **Principe** : on stocke les valeurs des attributs avec les non-terminaux associés dans la pile d'analyse
- Lors d'une dérivation, les attributs du non-terminal de la partie gauche sont dans la pile, donc utilisables pour calculer les attributs des non-terminaux à droite.
- De plus, lorsqu'un non-terminal de la partie droite est en cours d'analyse, les attributs des non-terminaux à sa gauche sont également disponibles.

31

## Exemple : extrait d'analyseur LL récursif

$X \rightarrow \{code1\} Y \{code2\} bWT \{code3\} Z \{code4\}$

### Procédure X()

selon symbole d'entrée faire :

cas ... : {code1}; Y(); {code2}; consommer(b);  
W(); T(); {code3}; Z(); {code4};  
autres cas ...

Fin Procédure X;

32

## Extrait d'analyseur LL récursif avec évaluation d'attributs

$X \rightarrow X_1 X_2 \dots X_k$

Procédure X( $h_1, \dots, h_n, s_1, \dots, s_m$ )

- calcul des attributs hérités  $h_1^1, \dots, h_n^1$  de  $X_1$  en fonction de  $h_1, \dots, h_n$  ;
- $X_1(h_1^1, \dots, h_n^1, s_1^1, \dots, s_m^1)$ ; //calcul des valeurs  $s_1^1, \dots, s_m^1$
- calcul des attributs hérités  $h_1^2, \dots, h_n^2$  de  $X_2$  en fonction des  $h_i$ , des  $h_i^1$  et des  $s_1^1$  ;
- $X_2(h_1^2, \dots, h_n^2, s_1^2, \dots, s_m^2)$ ; //calcul des valeurs  $s_1^2, \dots, s_m^2$
- ...
- $X_k(h_1^k, \dots, h_n^k, s_1^k, \dots, s_m^k)$ ; //calcul des valeurs  $s_1^k, \dots, s_m^k$
- calcul des attributs synthétisés  $s_1, \dots, s_m$  de X

Fin Procédure X;

33

## Exemple

Soit la grammaire attribuée G :

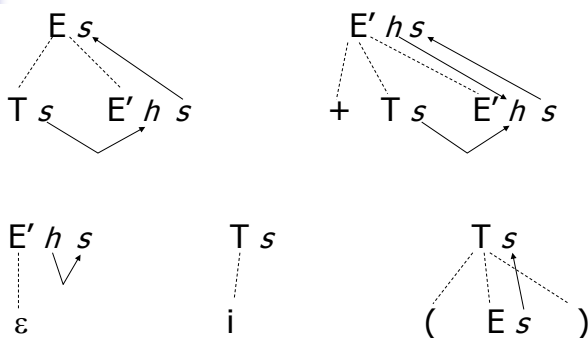
$E \rightarrow T E' \quad \{E'.h = T.s; E.s = E'.s;\}$   
 $E' \rightarrow + T E' \quad \{E'_1.h = E'_0.h + T.s; E'_0.s = E'_1.s;\}$   
 $E' \rightarrow \epsilon \quad \{E'.s = E'.h;\}$   
 $T \rightarrow i \quad \{T.s = lexical.val(i);\}$   
 $T \rightarrow (E) \quad \{T.s = E.s;\}$

s : entier (attribut synthétisé de E, E' et T)

h : entier (attribut hérité de E')

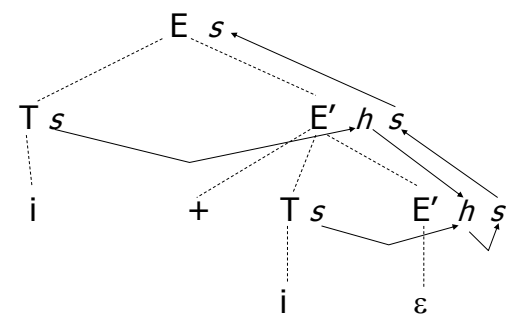
34

## Graphe de dépendances de G



35

## Idée de l'exécution sur « 3+4 »



36

## 4.7. Exemples d'analyses sémantiques typiques

- Table des symboles
- Vérification de types

37

## Calcul d'une table des symboles

Lorsqu'un identificateur est rencontré dans un programme (pour la première fois), il est ajouté comme entrée dans une structure appelée *table des symboles*.

Au cours de l'analyse, on complète l'entrée concernant cet identificateur avec des informations comme :

- sa nature : classe, méthode, attribut, variable, ... ;
- son type (dans le cas d'un attribut ou d'une variable) ou son profil (dans le cas d'une méthode) ;
- son contrôle d'accès : private, protected, public ;
- sa classe d'appartenance (la classe dans laquelle il est défini) ;
- son initialisation ;
- etc....

38

## Table des symboles : exemple

```
class A {  
    int var;  
    public boolean b=true;  
    int m1 (A x) {...}  
    private boolean n1 (A x, int n) {...}  
}
```

Ident	Nature	Type/Profil	Accès	Classe	Init ?	Val init
A	Classe		Package			
var	Variable	int	Package	A	Non	
b	Variable	boolean	Public	A	Oui	true
m1	Méthode	A -> int	Package	A		
n1	Méthode	A, int -> boolean	Private	A		

39

## Table des symboles : utilisation

- Lors d'une déclaration, l'identificateur est stocké dans la table :
    - s'il était déjà présent : erreur « identificateur déjà défini »
  - Lors d'une utilisation, on vérifie que l'identificateur est :
    - présent dans la table,
    - d'un type attendu en tenant compte des règles de sous-typage et de coercion (cast implicite),
    - initialisé si nécessaire (membre droit d'une affectation).
- Sinon : erreur.

40

## Table des symboles et portée des identificateurs

Dans les langages de programmation, il existe une notion de *portée des identificateurs* :

- variables locales et paramètres formels dans une méthode/procédure (ADA, Java, C++),
- variables locales à un bloc (C, ADA, JAVA),
- attributs d'une classe (C++, JAVA),
- classes locales à une classe (JAVA, C++),

```
void foo()  
{int x = 10;  
  {char x = 'c';}  
}
```

Ok en C

Faux en Java !

41

## Table des symboles et portée des identificateurs (suite)

Pour gérer la portée, on utilise une *pile de table des symboles* :

- Au début de l'analyse d'une structure locale, on empile une *nouvelle table vide* qui se remplit au cours de l'analyse de cette structure locale
- A la fin de cette dernière, on dépile la table des symboles au sommet
- L'ajout d'une entrée se fait toujours dans la table en sommet de pile
- En revanche, pour la recherche d'un identificateur, on doit "descendre" dans la pile jusqu'à rencontrer celui-ci

42

## Table des symboles : exemple

```
class essai{
  char y = 'a';
  boolean foo( char x )
  {char z = 'b';
   return (x == y );}
}
```

z	char
x	char
foo	char->boolean
y	char

43

## Typage (statique)

- *Typage statique* : vérification du typage d'un programme lors de la *compilation*.
- *Typage dynamique* : le typage n'est vérifié que lors de l'*exécution*.

**Typage statique** : On vérifie notamment que :

- une méthode ou un attribut appartient bien à la classe de l'objet sur lequel il s'applique
- une méthode est invoquée avec des arguments du bon type
- le membre droit et le membre gauche d'une affectation sont du même type
- etc...

44

## Vérification du typage : souvent complexe

- sous-typage et conversion implicite :
  - en C : on peut affecter un char à un int, effectuer des opérations arithmétiques sur les pointeurs, ...
  - en JAVA : le nombre 10 est vu selon le contexte comme un entier ou un flottant.
- héritage (JAVA, C++) :
  - un objet de la classe c' est également un objet de la classe c si c' hérite de c.
  - une méthode invoquée pour un objet de classe c' peut être définie dans une classe c dont c' hérite.
- surcharge des fonctions (Java, ADA) : un identificateur peut désigner des méthodes ou fonctions différentes.
- fonctions polymorphes (ML) et généricité (ADA) : le type des fonctions et variables dépend soit d'une instantiation particulière (ADA), soit de l'utilisation particulière qui en est faite (ML).

45

## Retour sur le traitement des erreurs

- Mode panique : *sans correction*, avec ou sans récupération par restart ou continuation sur symbole de synchronisation (mot-clé *error* en JCup)
- Avec correction au niveau du syntagme en utilisant des routines de traitement d'erreurs dans la table d'analyse
  - approprié pour les erreurs de programmation classiques
- Avec correction locale en définissant des règles spécifiques « cas d'erreur » dans la grammaire
- Avec correction automatique globale au niveau du programme :
  - l'analyseur « devine » les intentions du programmeur...
  - technique trop coûteuse pour être utilisée

46

## Rappel de l'exemple des expressions arithmétiques

- (0)  $E' \rightarrow E$
- (1)  $E \rightarrow E + E$
- (2)  $E \rightarrow E * E$
- (3)  $E \rightarrow ( E )$
- (4)  $E \rightarrow nb$

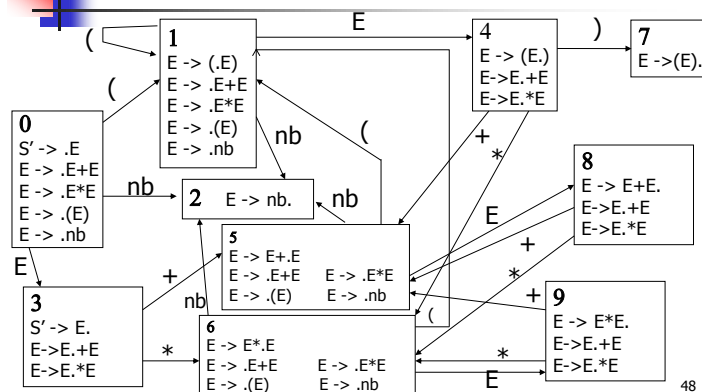
Avec règles de priorités :  
Prio(\*) > Prio(+)

Et associativité à gauche

Grammaire ambiguë... → ... mais conflits résolus

47

## Rappel de l'automate LR(0) pour l'exemple des expressions arithmétiques



48

### Rappel de la table LR(0) pour l'exemple des expressions arithmétiques - *modifiée avec priorités et associativité*

	nb	+	*	(	)	\$	E
0	d2			d1			3
1	d2			d1			4
2	r4	r4	r4	r4	r4	r4	
3		d5	d6			<b>Accepter</b>	
4		d5	d6		d7		
5	d2			d1			8
6	d2			d1			9
7	r3	r3	r3	r3	r3	r3	
8	r1	r1	d6	r1	r1	r1	
9	r2	r2	r2	r2	r2	r2	

49

### Mode panique :

sans correction, avec récupération par continuation sur symbole de synchronisation

-> Ajout d'une règle de récupération d'erreur dans la grammaire : mot clé *error*

(0)  $E' \rightarrow E$

(1)  $E \rightarrow E + E$

(2)  $E \rightarrow E * E$

(3)  $E \rightarrow ( E )$

(4)  $E \rightarrow \text{nb}$

(5)  $E \rightarrow \text{error}$  ) {afficher(« récup E sur ')' »)}

Avec règles de priorités :  
Prio(\*) > Prio(+)

Et associativité à gauche

50

### Mode panique avec récupération par continuation sur ')' – table LR(0) *modifiée*

	nb	+	*	(	)	\$	E
0	d2			d1	<b>r5</b>		3
1	d2			d1	<b>r5</b>		4
2	r4	r4	r4	r4	r4	r4	
3		d5	d6		<b>r5</b>	<b>Accepter</b>	
4		d5	d6		d7		
5	d2			d1	<b>r5</b>		8
6	d2			d1	<b>r5</b>		9
7	r3	r3	r3	r3	r3	r3	
8	r1	r1	d6	r1	r1	r1	
9	r2	r2	r2	r2	r2	r2	

51

### Correction au niveau du syntagme – reprise de l'exemple

Grammaire des expressions arithmétiques :

(0)  $E' \rightarrow E$

(1)  $E \rightarrow E + E$

(2)  $E \rightarrow E * E$

(3)  $E \rightarrow ( E )$

(4)  $E \rightarrow \text{nb}$

Avec règles de priorités :  
Prio(\*) > Prio(+)

Et associativité à gauche

-> Ajout de routines d'erreur dans la table LR(0)

52

### Correction au niveau du syntagme – table LR(0) *modifiée avec priorités et associativité* + routines d'erreur

	nb	+	*	(	)	\$	E
0	d2	<b>e1</b>	<b>e1</b>	d1	<b>e2</b>	<b>e1</b>	3
1	d2	<b>e1</b>	<b>e1</b>	d1	<b>e2</b>	<b>e1</b>	4
2	r4	r4	r4	r4	r4	r4	
3	<b>e3</b>	d5	d6	<b>e3</b>	<b>e2</b>	<b>Accepter</b>	
4	<b>e3</b>	d5	d6	<b>e3</b>	d7	<b>e4</b>	
5	d2	<b>e1</b>	<b>e1</b>	d1	<b>e2</b>	<b>e1</b>	8
6	d2	<b>e1</b>	<b>e1</b>	d1	<b>e2</b>	<b>e1</b>	9
7	r3	r3	r3	r3	r3	r3	
8	r1	r1	d6	r1	r1	r1	
9	r2	r2	r2	r2	r2	r2	

53

### Correction au niveau du syntagme – routines d'erreur

- e1** : dans états 0, 1, 5 ou 6

*Message* : **nb** ou parenthèse ouvrante **attendu** (au lieu d'opérateur ou de \$)

*Correction* : empiler un nb et aller en état 2

- e2** : dans états 0, 1, 3, 5 ou 6

*Message* : parenthèse fermante en trop

*Correction* : ignorer cette parenthèse fermante

54

## Correction au niveau du syntagme – routines d'erreur

- **e3** : dans états 3 ou 4

*Message* : **opérateur** ou *parenthèse fermante attendu* (au lieu de nb ou parenthèse ouvrante)

*Correction* : empiler un opérateur et calculer état

- **e4** : dans état 4

*Message* : opérateur ou **parenthèse fermante attendue** (au lieu de \$)

*Correction* : empiler une parenthèse fermante et aller en état 7

55

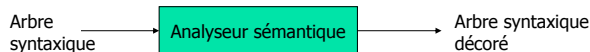
## Correction locale - règles spécifiques pour cas d'erreur

Si des erreurs courantes peuvent être décrites précisément : augmenter la grammaire avec des règles reconnaissant ces erreurs

$E \rightarrow E + )$	{action erreur nombre oublié}
$E \rightarrow E * )$	{action erreur nombre oublié}
$E \rightarrow E E$	{action erreur opérateur oublié}

56

## Conclusion sur l'analyse sémantique



- Vérification de propriétés sémantiques statiques des programmes (contextuelles)
- Définition dirigée par la syntaxe (grammaire algébrique attribuée par des attributs hérités ou synthétisés)
- Implantation par appel de procédures d'évaluation des attributs dans l'algorithme d'analyse syntaxique, suivant un schéma de traduction

57

## Plan de l'UE

### Introduction

1. Analyse lexicale
2. Grammaires algébriques
3. Analyse syntaxique (descendante, ascendante)
4. Analyse sémantique
5. **Production de code**

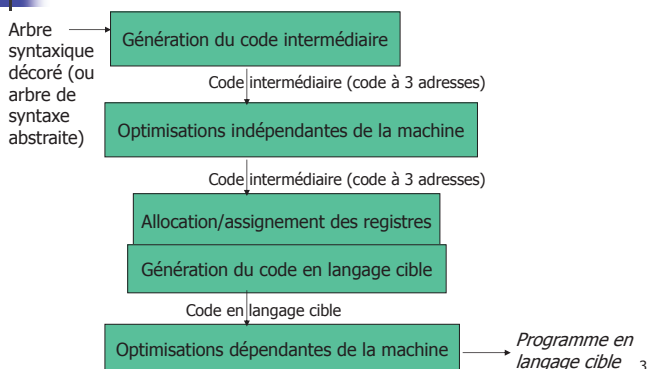
Conclusion - bilan

1

## 5. Production de code

2

## Production de code : objectif



3

## Production de code : plan

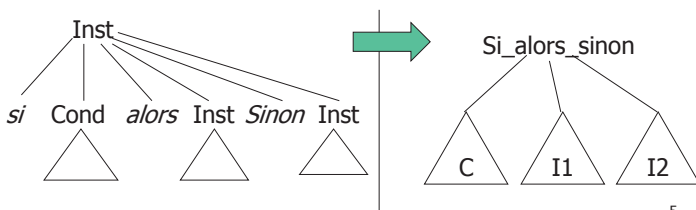
- 5.1. Arbre de syntaxe abstraite et sa simplification
  - par réécriture ou évaluation partielle
- 5.2. Environnement d'exécution
- 5.3. Génération de code intermédiaire (« code à 3 adresses »)
- 5.4. Optimisation de code intermédiaire indépendante de la machine
  - par analyse statique du flot de contrôle/données : évaluation/réduction des constantes, code mort, code redondant, extraction des invariants...
- 5.5. Production de code cible (Machine abstraite -> concrète)
  - Allocation/assignement des registres
  - Génération de code cible
  - Optimisation de code cible (dépendante de la machine)

4

## 5.1. Arbre de syntaxe abstraite

-> La production de code intermédiaire s'appuie sur l'arbre syntaxique abstrait construit lors de l'analyse syntaxique

**Inst -> si Cond alors Inst sinon Inst**



5

## Simplification de l'arbre de syntaxe abstraite par réécriture

- Si\_alors\_sinon(vrai,x1,x2) -> x1
- Si\_alors\_sinon(faux,x1,x2) -> x2
- Mult(x1,0) -> 0
- Mult(0,x1) -> 0
- Mult(x1,1) -> x1
- Mult(1,x1) -> x1
- Etc...

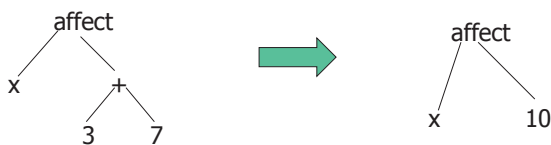
6



## Simplification de l'arbre de syntaxe abstraite par évaluation partielle

### évaluation partielle :

-> remplacer par sa valeur une expression calculable au moment de la compilation (entiers, réels, constantes, combinés par des opérateurs prédéfinis)



7

## 5.2. Environnement d'exécution : organisation de la mémoire

### Dans le langage source :

- Fonctions récursives ?
- Passage de paramètres (appel) ?
- Allocation mémoire dynamique ?
- Si oui, libération explicite ?

### Sur la machine cible :

- Longueur des mots ou adresses ?
- Entités directement adressables ?
- Instructions spécifiques accès ?
- Etc...

### Modèle général :

Code cible
Données statiques
Pile de contrôle
...
Tas
...

8

## 5.3. Le code intermédiaire

Le code intermédiaire contient :

- des variables de type de base : entier, booléen, flottant et éventuellement tableau
- des opérations "mémoire" explicites : allocation, libération (désallocation), arithmétique de pointeurs
- des expressions arithmétiques ou booléennes simples (utilisant des variables temporaires avec un opérateur)
- des sauts conditionnels ou non
- des CALL/RET utilisant une déclaration explicite des paramètres pour leur passage.

-> Le tout exprimé en termes de variables, registres (en nombre illimité !) et jeux d'instructions "simples" pour les calculs, les transferts, les branchements et les entrées/sorties.

9

## Code à 3 adresses simplifié (un extrait...)

Code à 3 adresses = séquence d'instructions numérotées

- $x := y \text{ op } z$  (affectation binaire)
  - $x := \text{opu } y$  (affectation unaire)
  - $x := y$  (copie)
  - $x[i] := y$  (affectation indicée)
  - aller en (a) (branchement inconditionnel)
  - si  $y \text{ op\_rel\_log } z$  alors aller en (a) (branchement conditionnel)
  - Lire  $x$
  - Écrire  $y$
- où
- op, opu et op\_rel\_log sont respectivement des opérateurs binaires, unaires, ou, relationnels ou logiques ("NON" inclus)
  - $x$  est une adresse de variable ou registre
  - $y, z$  sont des adresses de variables, registres ou constantes
  - (a) est une adresse du code, un numéro

10

## Génération du code intermédiaire

-> **Traduction dirigée par la syntaxe** à l'aide d'une grammaire attribuée (attributs *code*, *reg* et *inst\_suiv*)

### Génération du code pour instructions :

- Gestion des branchements et étiquettes

### Génération du code pour expressions (booléennes ou arithmétiques) :

- Par évaluation systématique : tous les termes composant une expression sont évalués
- Par évaluation *court-circuit* : seuls les termes nécessaires d'une expression sont évalués

11

## Exemples (1)

$x := 3*y + z;$   
 $y := y*x;$



(1)  $t := 3*y$   
(2)  $x := t + z$   
(3)  $y := y*x$

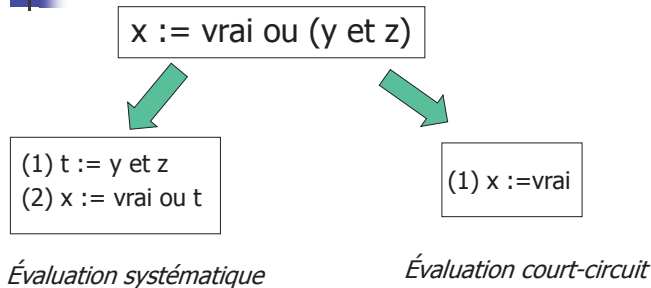
while  $x \neq 0$  do  
     $y[x] := z;$   
     $x := x - 1;$   
endwhile



(1) si  $x = 0$  aller en (5)  
(2)  $y[x] := z$   
(3)  $x := x - 1$   
(4) aller en (1)  
(5) ...

12

## Exemple (2)



13

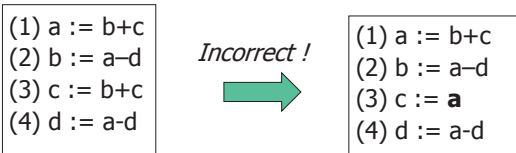
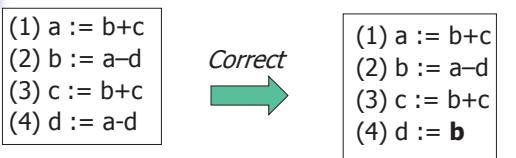
## 5.4. Optimisations du code intermédiaire

-> indépendantes de la machine cible

- Analyse statique de flot de données/contrôle :
  - Variables « constantes » dans une partie de programme (évaluation/réduction et propagation des constantes) ?
  - Code mort ?
  - Code redondant ?
  - Expression calculée plusieurs fois sans redéfinition de ses composantes (*élimination des expressions communes*) ?
  - Propagation des copies ?
  - Extraction des invariants : *invariants de boucles* ?
  - Etc...

14

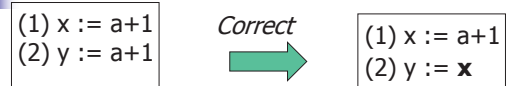
## Élimination des sous-expressions communes



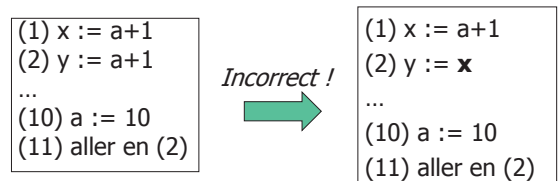
-> informations sur le flot de données nécessaires

15

## Élimination des sous-expressions communes (suite)



Mais :



-> informations sur le flot de contrôle nécessaires

16

## Graphe de flot de contrôle

-> On appelle *graphe de flot de contrôle* un graphe dont les sommets sont des blocs séquentiels de code (intermédiaire) et tel qu'un sommet *a* est relié à un sommet *b* si l'exécution du bloc *a* peut être suivie d'une exécution du bloc *b*, i.e :

- si le bloc *a* se termine par un branchement (conditionnel ou non) vers le bloc *b*
- si le bloc *b* suit le bloc *a* dans le code et *a* ne se termine pas par un branchement inconditionnel

-> L'optimisation « élimination des sous-expressions communes » est alors réalisée bloc par bloc en tenant compte du graphe de flot de contrôle.

17

## Construction du graphe de flot de contrôle

Comment déterminer les blocs du code intermédiaire ?

La *tête d'un bloc* est soit :

- la première instruction du programme
- une instruction suivant un branchement dans le code
- une instruction "visée" par un branchement

-> Un bloc est la suite d'instructions allant d'une tête à l'instruction précédant la tête suivante dans le code.

18

## Exemple : construction du graphe de flot de contrôle

```

(1) i := 3
(2) t1 := 4*i
(3) t2 := a[i]
(4) j := 2
(5) j := j+1
(6) si j>100 aller en (12)
(7) si t2<b aller en (10)
(8) t2 := t2+3
(9) aller en (7)
(10) b := b-j
(11) aller en (5)
(12) a[t1] := t2
    
```

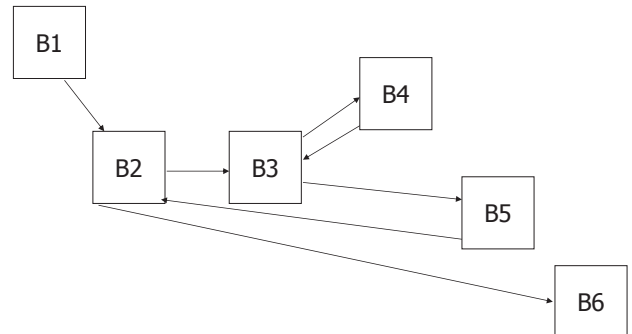
### Blocs :

```

B1 = {(1),(2),(3),(4)}
B2 = {(5),(6)}
B3 = {(7)}
B4 = {(8),(9)}
B5 = {(10),(11)}
B6 = {(12)}
    
```

19

## Exemple : graphe de flot de contrôle (suite)



20

## Optimisations locales et globales

- On appelle *optimisation locale* (ou *intra-bloc*) une optimisation interne à un bloc du flot de contrôle
- On appelle *optimisation globale* (ou *inter-blocs*) une optimisation qui traite en même temps plusieurs blocs du flot de contrôle
  - > le niveau global nécessite des informations sur le flot de données complémentaires / local
  - > par exemple, pour l'élimination des sous-expressions communes, on calcule pour chaque bloc B les ensembles d'expressions :
    - Produites :  $Prod(B)$
    - Supprimées :  $Supp(B)$
    - Disponibles en entrée ou en sortie de B ->  $In(B)$  et  $Ex(B)$

21

## Élimination des sous-expressions communes (opti locale au bloc B)

### Début

Disp = {}

**Pour chaque** instruction (i) de B de type  $x := \dots$  **Faire**

**Si** (i) de type  $x := op\ y$  ou  $x := y\ op\ z$  **alors**

**Si** Disp contient un couple  $(op\ y, var)$  ou  $(y\ op\ z, var)$

**alors** remplacer l'instruction (i) par  $x := var$

**Si**  $x \neq var$  **alors** ôter de Disp tout couple  $(exp, x)$  **Fsi**

**sinon** ôter de Disp tout couple  $(exp, x)$   
ajouter à Disp le couple  $(op\ y, x)$  ou  $(y\ op\ z, x)$

**Fsi**

**Fsi**

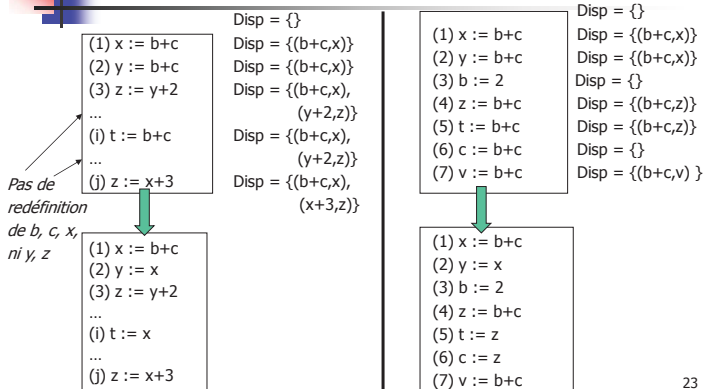
ôter de Disp tout couple  $(expr, v)$  tel que  $x \in expr$  (sauf cas  $\exists var$  et  $var=x$ )

**Refaire** avec instruction suivante de B

**Fin**

22

## Élimination des sous-expressions communes (ex. opti locale)



23

## Optimisation globale

Pour un bloc B,  $op\ x$  ou  $x\ op\ y$  est une expression :

- produite** si elle apparaît en membre droit d'une affectation, et si ni  $x$ , ni  $y$  ne sont redéfinis ensuite dans le bloc
- supprimée** si les valeurs de  $x$  ou  $y$  sont modifiées dans le bloc, et  $op\ x$  ou  $x\ op\ y$  n'est pas recalculée ensuite dans le bloc
- disponible en un point p** (souvent à l'entrée et à la sortie de B) si tous les chemins depuis le début du programme jusqu'à p évaluent  $op\ x$  ou  $x\ op\ y$ , et ni  $x$ , ni  $y$  ne sont redéfinis depuis la dernière évaluation de  $op\ x$  ou  $x\ op\ y$

24

## Algorithme de calcul de Prod(B) : ensemble des expressions produites par le bloc B

### Début

Prod(B) = {}

**Pour chaque** instruction de B

du type  $x := op\ y$  ou  $x := y\ op\ z$  **faire**

Ajouter à Prod(B) l'expression  $op\ y$  ou  $y\ op\ z$

Ôter de Prod(B) toute expression contenant  $x$

**Refaire** avec instruction suivante de B

Retourner Prod(B)

### Fin

25

## Algorithme de calcul de Supp(B) : ensemble des expressions supprimées par le bloc B

### Début

Supp(B) = {}

E = ensemble de toutes les expressions du programme

**Pour chaque** instruction de B

du type  $x := op\ y$  ou  $x := y\ op\ z$  **faire**

Ajouter dans Supp(B) toutes les expressions de E  
contenant  $x$  et non recalculées plus loin dans le  
bloc B

**Refaire** avec instruction suivante de B

Retourner Supp(B)

### Fin

26

## Algorithme de calcul de In(B) et Ex(B) : ensembles des expressions disponibles en entrée et en sortie du bloc B

### Début

$B_1$  = bloc point d'entrée du programme (non cible de rebouclage)

E = ensemble de toutes les expressions du programme

$In(B_1) = \{\}$   $Ex(B_1) = Prod(B_1)$

**Pour chaque** bloc  $B \neq B_1$  **faire**

$Ex(B) = E - Supp(B)$

**Refaire**

**Répéter**

**Pour chaque** bloc  $B \neq B_1$  **faire**

$In(B) = \cap Ex(P)$  pour tous les blocs P prédécesseurs de B

$Ex(B) = (In(B) - Supp(B)) \cup Prod(B)$

**Refaire**

**Jusqu'à** plus de changements

Retourner  $Ex(B)$  et  $In(B)$  pour tous les blocs du programme

### Fin

27

## Optimisation globale : élimination des sous-expressions communes

Pour chaque instruction (i) du bloc B du type

$x := op\ y$  ou  $x := y\ op\ z$

telle que  $op\ y$  ou  $y\ op\ z$  est disponible à l'entrée de B,  
et ni  $y$ , ni  $z$  ne sont modifiées avant (i) dans B :

- on considère une nouvelle variable  $u$
- dans tous les chemins possibles arrivant à l'instruction (i), on recherche la dernière évaluation de l'expression considérée
- on remplace cette dernière évaluation  $w := op\ y$  ou  $w := y\ op\ z$  par  

$$u := op\ y \quad \text{ou} \quad u := y\ op\ z$$

$$w := u \quad \quad \quad w := u$$
- on remplace l'instruction (i) par  $x := u$

-> Les algorithmes d'optimisation sont itératifs :  
différentes passes successives sont nécessaires

28

## Propagation des copies

(1)  $x := y$   
(2)  $z := x+1$   
(3)  $w := 2*z$   
(4)  $x := x-1$   
(5)  $t := x+w$



(1)  $z := y+1$   
(2)  $w := 2*z$   
(3)  $x := y-1$   
(4)  $t := x+w$

-> comme « l'élimination des sous-expressions communes »,  
la « propagation des copies » peut être faite en intra-bloc,  
i.e. localement, mais aussi globalement.

29

## Propagation des copies (optimisation locale au bloc B)

### Début

Copies = {}

**Pour chaque** instruction  $i$  de B **Faire**

**Si** ( l'instruction  $i$  utilise  $y$  et/ou  $z$   
et Copies contient  $(y, y', inst_y)$  et/ou  $(z, z', inst_z)$  )

**alors** remplacer  $y$  par  $y'$  et/ou  $z$  par  $z'$  dans l'instruction  $i$  **Fsi**

**Si** l'instruction  $i$  définit  $x$

**alors** Copies = Copies -  $\{(v1, x, i1)\} - \{(x, v2, i2)\} \forall v1, v2, i1, i2$  **Fsi**

**Si** (la nouvelle)  $i$  est de type  $x := y'$  **alors** Copies = Copies +  $\{(x, y', i)\}$  **Fsi**

**Refaire** avec instruction suivante de B

**Pour chaque** instruction de copie  $i$  de B (du type  $x := y$ ) **Faire**

**Si**  $x$  locale à B et  $x$  non utilisée entre  $i$  et sa prochaine redéfinition

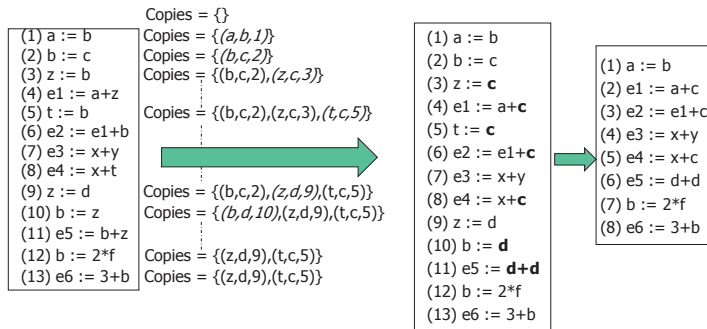
**alors** supprimer l'instruction  $i$  **Fsi**

**Refaire** avec instruction suivante de B

### Fin

30

## Propagation des copies (exemple d'optimisation locale)



31

## Optimisation globale : propagation des copies

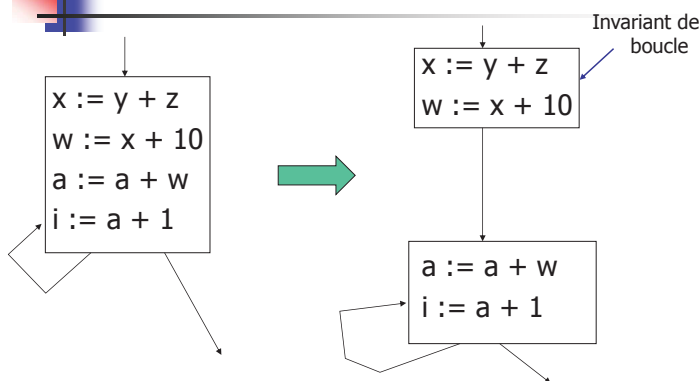
L'instruction de copie (i)  $x := y$  peut être éliminée, en remplaçant  $x$  par  $y$  dans toute expression utilisant  $x$  en un point  $p$ ,

si pour tout  $p$  :

- l'instruction (i) est l'unique définition de  $x$  atteignant  $p$ ,
- aucun chemin menant de (i) à  $p$  ne contient de redéfinition de la variable  $y$  (mais attention aux boucles !!)

32

## Optimisation globale : invariants de boucle



33

## 5.5. Production de code cible

Machine abstraite -> machine concrète

- Allocation/assignation des registres
- Génération de code cible :  
instructions générales -> instructions spécifiques à la machine cible
- Optimisation de code cible dépendante de la machine

34

## Production du code cible : but

-> **Génération de langage machine ou de langage d'assemblage pour une machine :**

- disposant d'un nombre fini de registres et d'une mémoire de taille de mots fixée,
- disposant d'une unité centrale :
  - jeu d'instructions
  - différents modes d'adressage
  - unité arithmétique et logique
- disposant d'un jeu d'appels système (interruptions)

35

## Production du code cible : problématique

- Comment assigner les variables du code intermédiaire à de la mémoire (RAM ou registres) ?
- Comment allouer les registres ? (Quelle variable dans quel registre ?)
- Comment transformer une instruction du code intermédiaire en une (série d')instruction(s) d'assembleur ?

-> Ces étapes dépendent de la machine mais également les unes des autres.

36

## Assignement/allocation des registres : problématique

- Nombre de variables dans le code intermédiaire élevé vs nombre de registres de la machine cible réduit
  - Utilisation spécifique de certains registres
- > Utilisation des registres nécessaire :
- pour les instructions arithmétiques ou logiques (au moins pour une des deux opérandes et le résultat)
  - pour un souci d'efficacité : un calcul entre deux registres est plus rapide qu'entre un registre et le contenu d'une adresse
  - pour certains modes d'adressage (adressage indexé)
  - ....

37

## Assignement/allocation des registres : réalisation

Le code produit doit :

- choisir aux différents points du programme quelle variable mettre dans un registre et dans quel registre
- charger les données dans les registres au moment opportun (calcul...)
- libérer les registres quand nécessaire en remplaçant les données en mémoire (calcul sur registre avec une autre valeur)

38

## Assignement de registres (méthode par coloriage de graphes)

- On construit un graphe d'interférence GI dont
- les sommets sont les variables du code intermédiaire
  - il y a une arête entre le sommet a et le sommet b si
    - la variable a est définie précédemment à une instruction utilisant la variable b
    - la variable a est utilisée postérieurement à cette instruction
- > deux variables qui interfèrent ne peuvent pas être assignées au même registre.

39

## Assignation de registres par coloriage de graphe : détails

- On considère k couleurs (les k registres disponibles) et on construit un coloriage du graphe GI :
- tous les sommets ont une couleur
  - deux sommets reliés ne peuvent avoir la même couleur
- On essaye en réalité de colorier le graphe avec le moins de couleurs possibles :
- si on utilise moins de k couleurs, on a un assignement des registres
  - si on utilise plus de k couleurs, alors on insère là où nécessaire le code pour le chargement/déchargement des registres

40

## Génération du code cible : choix des instructions assembleur les plus efficaces (1)

### Code intermédiaire : Code assembleur basique :

```
(1) i := 0
(2) Si i=100 aller en (6)
(3) t(i) := 0
(4) i := i+1
(5) Aller en (2)
(6) ...
```

```
MOV R0, t
MOV R1, 0
Debut : COMP R1, 100
JEQ Suite
MOV [R0], 0
ADD R0, 1
ADD R1, 1
JMP Debut
Suite : ...
```

41

## Génération du code cible : choix des instructions assembleur les plus efficaces (2)

### Code assembleur (avec INC) : Code assembleur (indexé) :

```
MOV R0, t
MOV R1, 0
Debut : COMP R1, 100
JEQ Suite
MOV [R0], 0
INC R0
INC R1
JMP Debut
Suite : ...
```

```
MOV R0, t
MOV RI, 0
Debut : COMP RI, 100
JEQ Suite
MOV [R0+RI], 0
INC RI
JMP Debut
Suite : ...
```

42



## Optimisations du code cible

-> dépendantes de la machine cible

- Inlining : « dépliage » des petites procédures
- Prise en compte des modes d'adressage
- Exploitation des possibilités de parallélisme du processeur cible
- Etc...

43



## Optimisation du code cible par inlining

Inlining (ou Développement en ligne)

-> on remplace l'appel à un sous-programme (CALL/RET) par le code du sous-programme

- on perd de la place (ne développer que les "petites" fonctions)
- on obtient un code plus efficace en temps

44



## Conclusion sur la production de code

- Production de code intermédiaire naïf facile avec grammaire attribuée
- Optimisations : compromis coût/efficacité recherchée à étudier !
- Pour un compilateur donné, il n'est pas forcément nécessaire de toutes les implanter
- Production et optimisation du code cible : connaissance approfondie de la machine cible
- Ce chapitre ne se veut pas exhaustif...

45



## Conclusion générale du cours

- Analyse lexicale, syntaxique et sémantique : environ 10 % du temps total de compilation
  - Une grammaire bien écrite, suffisamment détaillée, simplifie le traitement des erreurs et l'analyse sémantique
- Génération et optimisation du code : environ 90 % du temps de compilation !
  - Importance du code intermédiaire pour "porter" le compilateur à moindre coût

Lien avec les outils manipulés en TP :

- Création d'analyseurs LR (LALR) avec JFlex/JCUP

46