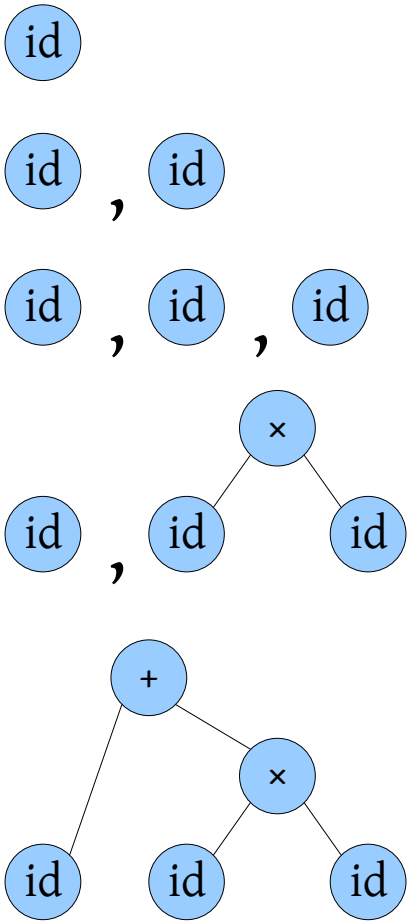
# Compilation

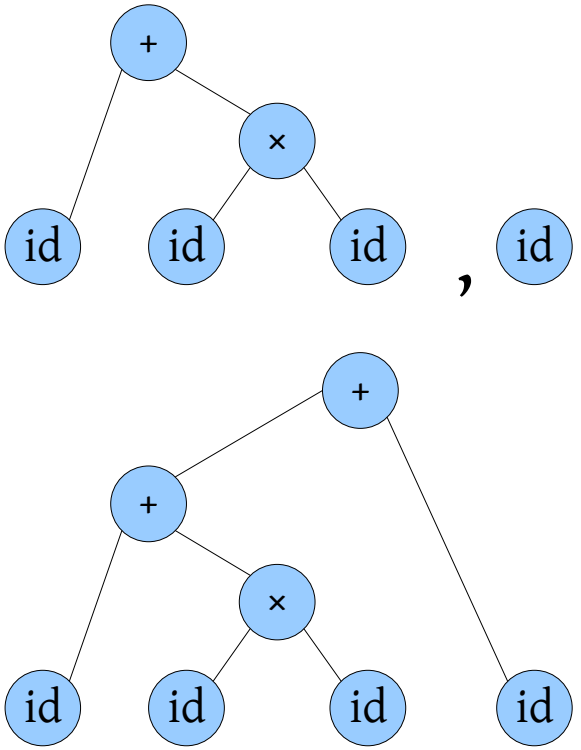
Analyse descendante

# Principe de l'algorithme

- On « construit » (en fait parcours) l'arbre syntaxique en partant des feuilles.
- On construit d'abord les sous-arbres d'un nœud pour ensuite créer le nœud lui-même
- L'algorithme utilise une pile pour contenir les sous-arbres en train d'être construits
- Quand on construit un nouveau nœud, on applique une production « à l'envers » ; on appelle cela effectuer une *réduction*
- C'est le lookahead qui dicte que réduction effectuer

# Exemple : id + id × id + id



# Item LR0

- Le principe de l'algorithme est d'effectuer plusieurs analyses LL en parallèle : on se souvient de toutes les productions applicables (au lieu d'une seule déterminée au début)
- Pour symboliser qu'une production est en train d'être appliquée, on la représente avec un point à l'endroit où l'on se trouve :

  E ::= E • '+' T

  signifie que l'on a déjà lu un E, et que l'on peut éventuellement lire '+' T

# État LR0

- Une état LR0 est un ensemble d'items qui indique les productions en cours
- Par exemple, un état pourrait être
  - E ::= E '+' E •
  - E ::= E • '+' E
  - T ::= • id
  - T ::= • n
  - T ::= • ( E )
- **Noter que E ::= E '+' E • et E ::= E • '+' E sont des items différents**

# Automate des items LR0

- Les analyses ascendantes sont basées sur un automate appelé *automate des items LR0*
- On construit cet automate en découvrant les états à partir de ceux déjà créés (comme lors de la déterminisation d'un automate fini)
- La construction s'arrête quand toutes les possibilités sont explorées

# Clôture

- Lors de la création d'un nouvel état, on ajouter certains items
- Il faut alors en effectuer la *clôture*
- Dès que un item contient un • devant un non-terminal E, on ajoute un item

$$E ::= \bullet \gamma$$

  pour toute production E ::= γ issue de E.
- Si des items dont le • est devant un non-terminal sont ajoutés, on recommence avec ce non-terminal

# Exemple

- Avec la grammaire
  - E ::= E '+' T | T
  - T ::= T '×' F | F
  - F ::= 'id' | '(' E ')'
- On complète
  - E ::= E '+' • T
- par
  - T ::= • T '×' F
  - T ::= • F
- et puis par
  - F ::= • id
  - F ::= • '(' E ')'

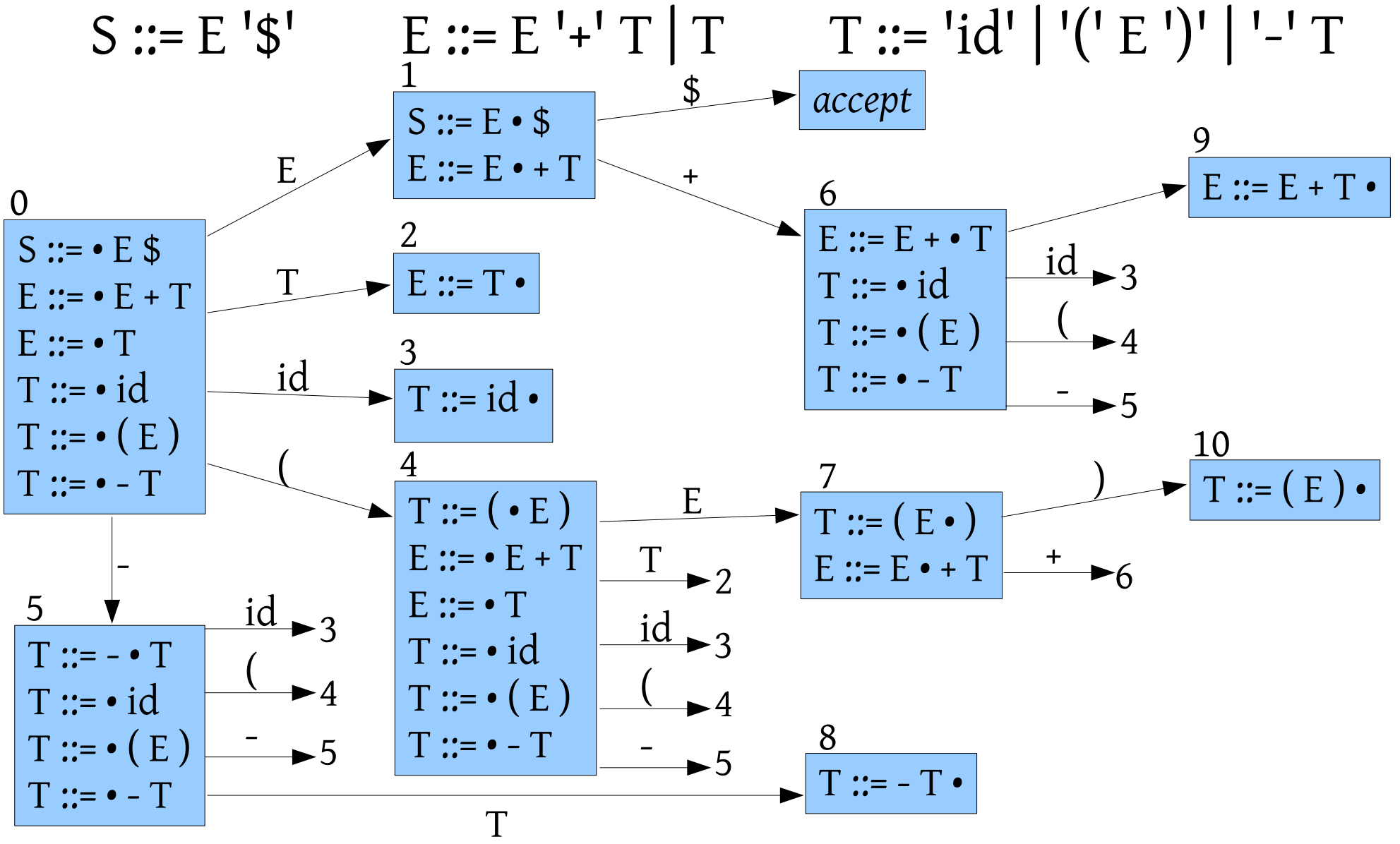
# Création de nouveaux états

- On crée un nouvel état à partir d'un autre en faisant bouger le • d'un cran vers la droite en « lisant » soit un terminal soit un non-terminal
- Si on décide de lire 'x', on ajoute au nouvel état tous les items A ::= γ 'x' • δ pour tout item A ::= γ • 'x' δ de l'ancien
  - **Attention à ne pas oublier d'items (il y en a souvent qu'un mais pas tout le temps)**
- Il faut bien vérifier que l'état créé n'existe pas déjà

# État initial, items et états finaux

- Le premier état, est celui qui contient l'item S ::= • E '$'
- Il faut donc au moins y ajouter la clôture de E
- Attention, il y a souvent toutes les productions dans cet état, mais pas tout le temps !
- Un item dont le • est à la fin est un *item final*
- Un état qui contient un item final est appelé *état final*

# Exemple


S ::= E '$'
E ::= E '+' T | T
T ::= 'id' | '(' E ')' | '-' T
0
S ::= • E $
E ::= • E + T
E ::= • T
T ::= • id
T ::= • ( E )
T ::= • - T
1
S ::= E • $
E ::= E • + T
accept
2
E ::= T •
3
T ::= id •
4
T ::= ( • E )
E ::= • E + T
E ::= • T
T ::= • id
T ::= • ( E )
T ::= • - T
5
T ::= - • T
T ::= • id
T ::= • ( E )
T ::= • - T
6
E ::= E + • T
T ::= • id
T ::= • ( E )
T ::= • - T
7
T ::= ( E • )
E ::= E • + T
8
T ::= - T •
9
E ::= E + T •
10
T ::= ( E ) •
E
T
id
(
-
$
+
)

# Analyseurs LR

- Un état tout seul ne suffit pas à décrire complètement l'état du parsing
- L'état complet est donné par une pile d'états, qui permet, par exemple, de compter le nombre de parenthèses ouvertes
- Les actions des analyseurs LR consistent à modifier l'état et la pile et/ou d'avancer sur l'entrée
  - soit on empile le terminal lu dans la pile
  - soit on effectue une réduction : on remplace le membre droit d'une production situé au sommet de la pile par le membre gauche (contraire de LL)

# Analyse LR0

- L'analyse LR0 n'utilise pas de *lookahead*.
- Il y a deux actions possibles :
  - **shift**, qui empile le terminal lu sur l'entrée dans la pile
  - **reduce**, qui applique une production en faisant un réduction
- On commence par empiler l'état initial
- En fonction de l'état au sommet de la pile
  - On applique **reduce** si l'état est constitué d'un seul item final
  - S'il n'y a pas d'item final, on applique **shift**
  - Sinon, il y a conflit LR0 (plusieurs items dont un final)

# Analyse LR0

- Chaque case de la pile contient un terminal ou un non-terminal et un état
- L'état courant est l'état présent au sommet de la pile
- Quand l'analyseur effectue un shift, on empile **le terminal lu et l'état où mène ce terminal** dans l'automate des items, depuis l'état courant
- Quand l'analyseur effectue un reduce par la production T ::= γ, on dépile autant que la taille de γ (donc rien si ε), et on empile **T et l'état où mène T** dans l'automate des items, depuis l'état courant.

# Table LR0

- La table LR0 contient deux parties :
  - La première contient l'action à effectuer en fonction de l'état
    - l'état à empiler en fonction du terminal lu dans le cas d'un **shift**
    - le production par laquelle on effectue un **reduce**
  - La seconde est la table des goto, c'est-à-dire à quel état on arrive après une réduction, en fonction du non-terminal de gauche et de l'état d'où l'on part après avoir dépilé le membre droit
- Ces deux tables sont données par l'automate des items LR0, et suffisent à effectuer le parsing

# Table LR0 : exemple

| | $ | ( | ) | id | - | + | E | T |
|---|---|---|---|---|---|---|---|---|
| 0 | | s4 | | s3 | s5 | | 1 | 2 |
| 1 | *acc* | | | | | s6 | | |
| 2 | reduce | | | | | | | |
| 3 | reduce | | | | | | | |
| 4 | | s4 | | s3 | s5 | | 7 | 2 |
| 5 | | s4 | | s3 | s5 | | | 8 |
| 6 | | s4 | | s3 | s5 | | | 9 |
| 7 | | | s10 | | | s6 | | |
| 8 | reduce | | | | | | | |
| 9 | reduce | | | | | | | |
| 10 | reduce | | | | | | | |

# Fonctionnement

| entrée | pile | action | note |
|---|---|---|---|
| id+id+id$ | 0 | shift 3 | |
| +id+id$ | 0 (id,3) | reduce T ::= 'id' | 0,T goto 2 |
| +id+id$ | 0 (T,2) | reduce E ::= T | 0,E goto 1 |
| +id+id$ | 0 (E,1) | shift 6 | |
| id+id$ | 0 (E,1) (+,6) | shift 3 | |
| +id$ | 0 (E,1) (+,6) (id,3) | reduce T ::= 'id' | 6,T goto 9 |
| +id$ | 0 (E,1) (+,6) (T,9) | reduce E ::= E '+' T | 0,E goto 1 |
| id$ | 0 (E,1) | shift 6 | |
| $ | 0 (E,1) (+,6) | shift 3 | |
| $ | 0 (E,1) (+,6) (id,3) | reduce T ::= 'id' | 6,T goto 9 |
| $ | 0 (E,1) (+,6) (T,9) | reduce E ::= E '+' T | 0,E goto 1 |
| $ | 0 (E,1) | *accept* | |

# Raffinement SLR

- On peut résoudre facilement certains conflits LR0 grâce à la méthode SLR (simple LR)
- Quand un état contient un item final E ::= γ •, on effectue la réduction par E ::= γ que pour les **Suivant**(E)
- Il y a conflit SLR quand il y plusieurs actions possibles pour un état et un terminal d'entrée donné
- Pour SLR, on a besoin de lire un caractère à l'avance
- SLR ne coûte pas plus cher que LR0, mais accepte beaucoup plus de grammaires
- C'est l'algorithme par défaut « à la main »

# Exemple

- S ::= E '$'
- E ::= E '+' T | E '-' T | T
- T ::= T '×' F | T '÷' F | F
- F ::= '(' E ')' | '-' F | 'id'

# LR1

- Un raffinement supplémentaire est LR1
- Il est beaucoup plus gourmand et lourd
- Dans chaque état, on s'intéresse en plus à ce qu'il peut y avoir comme *lookahead* après chaque item de l'état
  - par exemple, après E ::= T, il ne peut y avoir que ) et $
- Les items LR1 sont constitués d'un item LR0 et d'un terminal : l'un des possibles *lookahead*
  - par exemple, E ::= • T , $ et E ::= • T , )
- La valeur du terminal est ce qu'il y peut y avoir **une fois la réduction par la production effectuée** et non après le •

# États LR1

- La construction de l'automate des items LR1 se fait de la même manière que LR0 sauf :
  - on augmente la grammaire sans mettre un $ en plus
  - l'item initial est S ::= • E , $
  - en faisant avancer les •, on garde la *lookahead* de l'item
  - la règle de clôture est :
    - pour tout item

      $$X ::= \gamma \bullet A\ \delta\ ,\ a$$

      on ajoute tous les items

      $$A ::= \bullet\ \eta\ ,\ u$$

      pour toute production A ::= η issue de A et tout u de **Premier**(δa) *(attention, extension)*

# Exemple

- S ::= I
- I ::= V '=' E
- I ::= E
- E ::= V
- V ::= 'id'
- V ::= '*' E

- rien n'est annulable
- Premier(_)={id,*}

S ::= • I , $
I ::= • V '=' E , $
I ::= • E , $
V ::= • 'id' , =
V ::= • '*' E , =
E ::= • V , $
V ::= • 'id' , $
V ::= • '*' E , $

S ::= • I , $
I ::= • V '=' E , $
I ::= • E , $
V ::= • 'id' , = $
V ::= • '*' E , = $
E ::= • V , $

# Table LR1

- La table LR1 est construite sur le même modèle que la table LR0 (actions et goto)
- Les réductions ne sont faites que pour les *lookahead* des items LR1 correspondant
- La marche de l'analyseur est la même que pour LR0
- Il y a conflit LR1 quand il y plusieurs actions possibles pour un état et un terminal d'entrée donné

# LALR

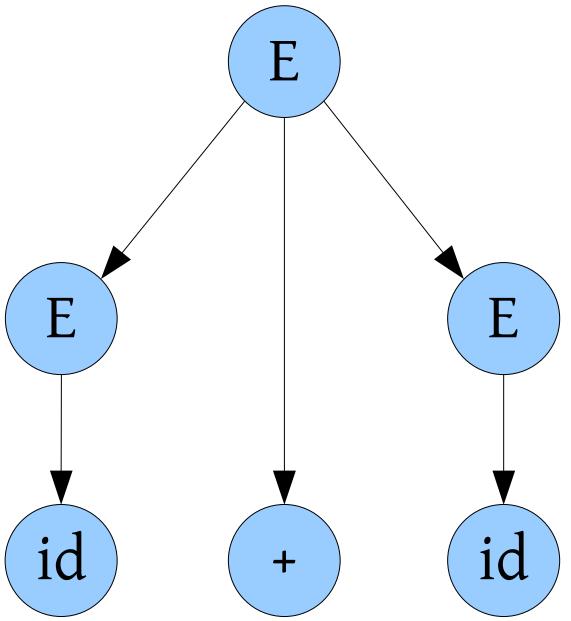
- LALR, pour *lookahead* LR, est une simplification de la table LR1 : un fois l'automate des items LR1 construits, on simplifie l'automate en ne considérant que la partie LR0 des items.
- On regroupe les états ayant les même items LR0 (mais des valeurs de *lookahead* différents).
- Il y a conflit LALR si deux états regroupés stipulent des actions différentes pour un terminal d'entrée donné.
- Il existe un algorithme compliqué (implémenté dans Tatoo) pour calculer l'automate LALR sans passer par LR1

# Construction de l'AST

- Lors de l'analyse LR, la suite des actions effectuées correspondent à un parcours de l'arbre syntaxique **en profondeur postfixe**
- Un shift correspond à parcourir une feuille
- Un reduce correspond à parcourir un nœud après avoir parcouru ses fils
- On parcours l'arbre syntaxique sans qu'il soit présent en mémoire (analogue à l'approche SAX)

# Construction de l'AST

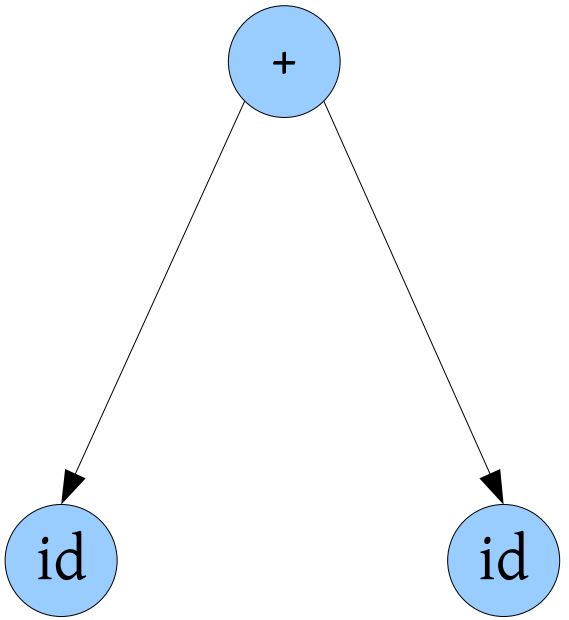
- shift 'id'
- reduce E ::= 'id'
- shift '+'
- shift 'id'
- reduce E ::= 'id'
- reduce E ::= E '+' E




- Plutôt que de construire l'arbre syntaxique en mémoire, il vaut mieux construire l'AST à partir de ce parcourt.

# Construction de l'AST

- shift 'id'
- reduce E ::= 'id'
- shift '+'
- shift 'id'
- reduce E ::= 'id'
- reduce E ::= E '+' E



# Grammaire ambigüe

- Une grammaire ambigüe provoque obligatoirement des conflits, LL, SLR, LALR ou LR
- On peut désambigüiser la grammaire (lourd pour de gros projets)
- On peut aussi résoudre les conflits en choisissant intelligemment l'actions à effectuer parmi les différentes possibles
  - À l'aide des informations contenues dans les items de l'état
  - En utilisant un mécanisme de priorité

# Exemple

- L'état suivant provoque un conflit avec +, on hésite entre shift et reduce (+ est dans **Suivant**(E))

E ::= E '+' E •
E ::= E • '+' E

- Choisir shift revient à effectuer le second + avant le premier : id+(id+id)
- Choisir reduce revient à effectuer le premier + avant le second : (id+id)+id
- On choisit donc reduce car + est associatif à gauche

# Exemple

- L'état suivant provoque un conflit avec ×, on hésite entre shift et reduce (× est dans **Suivant**(E))

E ::= E '+' E •
E ::= E • '×' E
E ::= E • '+' E

- Choisir shift revient à effectuer × avant + : id+(id×id)
- Choisir reduce revient à effectuee + avant × : (id+id)×id
- On choisit donc shift car × est prioritaire par rapport à +

# Résolution par priorité

- On donne des priorités aux terminaux et aux productions
- Pour cela, on donne une valeur de priorité et d'associativité (gauche (+), droite (=) et non associatif (<)) aux opérateurs
- Deux opérateurs peuvent avoir la même priorité, mais doivent aussi avoir la même associativité
- Les productions mettant en scène ces opérateurs en reçoivent la priorité et l'associativité
- Attention aux opérateurs à sémantique multiple (moins unaire et moins binaire)

# Résolution par priorité

- Lors d'un conflit entre shift et reduce, on choisit l'opération prioritaire (priorité du terminal contre priorité de la production)
- Si la priorité est identique, on regarde l'associativité :
  - à gauche, on choisit la réduction
  - à droite, on choisit le shift
  - non-associatif, on choisit « error »
- Les ambigüités liées aux priorités ne génèrent pas de conflits reduce-reduce qui témoigne en général d'un problème de grammaire.

# Exemple

- E ::= E '+' E | E '-' E | '-' E | '(' E ')' | E '=' E | E '<' E | 'id
- - unaire prioritaire sur + et – binaire, prioritaires sur <, prioritaire sur =
- + et – associatifs à gauche, < non associatifs et = à droite (pas de sens pour – unaire)

# Cas du *if ... then ... else*

- La grammaire
  - I ::= 'if' E 'then' I 'else' I | 'if' E 'then' I

  pose un problème pour l'état

  I ::= 'if' E 'then' I • 'else' I
  I ::= 'if' E 'then' I •

  car else est dans **Suivant**(I)
- Si on privilégie shift, le else se rapport au if non attribué le plus proche
- Si on privilégie reduce, jamais shift n'est effectué et aucun programme contenant un else ne compile

# Cas du *if ... then ... else*

- On peut soit désambigüiser la grammaire :
  - I ::= 'if' E 'then' I | 'if' E 'then' I2 'else' I | ...
  - I2 ::= if E then I2 else I2 | ...
- Soit dire que 'else' est prioritaire par rapport au 'then' pour faire un shift du 'else' à la place du reduce par la production I ::= 'if' E 'then' I
- Utiliser des priorités permet de simplifier l'insertion de l'instruction if parmi les autres instructions

# Reduce-reduce

- S ::= E '$'
- E ::= A V | A
- A ::= 'a' | 'b'
- V ::= 'v' | ε

- L'état

  E ::= A • V
  E ::= A •
  V ::= 'v' •
  V ::= •

  a un conflit reduce-reduce pour '$' de **Suivant**(E)

- Ce conflit est dû à l'ambigüité E—›A et E—›AV—›A

# Gestion d'erreurs

- Il y a principalement deux manières de recouvrer les erreurs avec les analyseurs LR
  - le prévoir dans la grammaire en utilisant un terminal *error*
  - utiliser un algorithme de tentative d'insertion/suppression de terminaux
- Comme pour LL, on peut agir directement en mettant des actions spécifiques fonctions du contenu de l'état

# Le terminal *error*

- Les productions tentent de prévoir les erreurs du programmeur :
  - expr ::= '(' 'error' ')'
    - signifie que si une erreur arrive après une ')', on recherche '('
  - paramList ::= error ',' paramList
    - signifie qu'en cas d'erreur, on recherche la prochain ',' pour continuer la liste des paramètres

# Le terminal *error*

- Quand une erreur arrive, l'analyseur effectue les actions suivantes :
  - il dépile des états jusqu'à tomber sur un état qui accepte une action quand le *lookahead* est 'error'
    - cela revient à oublier des terminaux déjà pris en compte sur l'entrée, par exemple tout jusqu'à '('
  - il effectue les actions jusqu'à ce qu'un shift d'error soit effectué
  - il jette les terminaux de l'entrée jusqu'à ce que l'action ne soit pas un erreur
    - cela revient à oublier les terminaux à venir, par exemple jusqu'au prochain ')'