Langages et Compilation

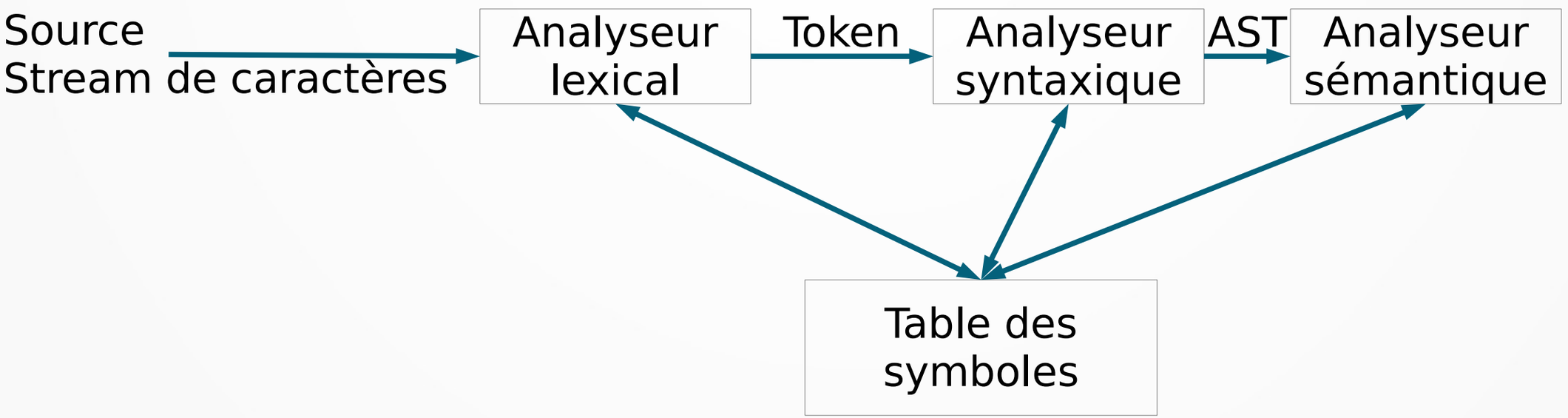
L4: Analyse Syntaxique (2)

T. Goubier

L3A

2019/2020

L4: Analyse Syntaxique (2)



L4: Analyse Syntaxique (2)

- L3: top-down parsing
- L4: bottom-up parsing
  - Une manière plus puissante de faire de l'analyse syntaxique
  - Celle utilisée dans SmaCC
    - Et Bison, et Yacc, etc.
  - Principe
    - Partir des tokens
    - Et reconstruire des phrases de plus en plus complexes (arbre)
    - Sans jamais avoir à revenir en arrière

L4: Analyse Syntaxique (2)

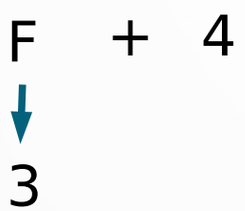
- Approche

3 + 4

E → E + T | T
T → T * F | F
F → (E) | <numb

L4: Analyse Syntaxique (2)

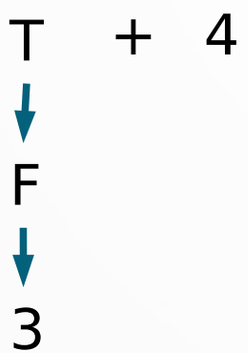
- Approche



E → E + T | T
T → T * F | F
F → (E) | <numb

L4: Analyse Syntaxique (2)

- Approche

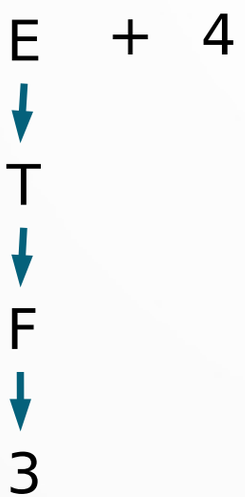


E → E + T | T
T → T * F | F
F → (E) | <numb

L4: Analyse Syntaxique (2)

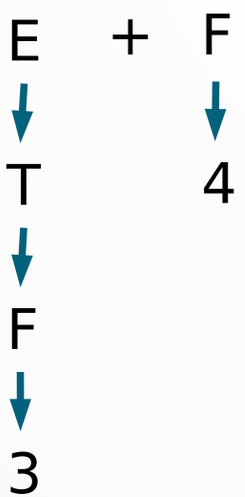
- Approche

E → E + T | T
T → T * F | F
F → (E) | <numb



L4: Analyse Syntaxique (2)

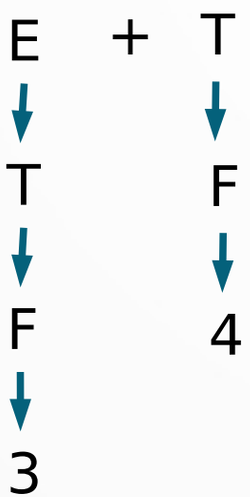
- Approche



E → E + T | T
T → T * F | F
F → (E) | <numb

L4: Analyse Syntaxique (2)

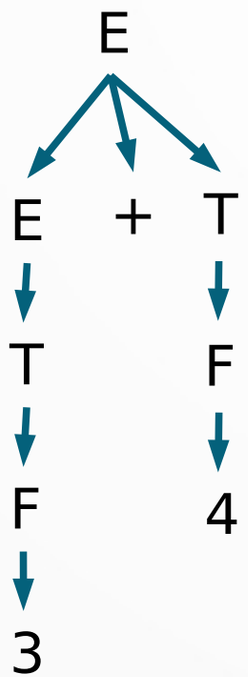
- Approche



E → E + T | T
T → T * F | F
F → (E) | <numb

L4: Analyse Syntaxique (2)

- Approche



E → E + T | T
T → T * F | F
F → (E) | <numb

L4: Analyse Syntaxique (2)

- Principes
  - Deux opérations: shift et reduce
  - Reduce:
    - Remplacer la phrase ω par A, si on a une production A → ω
    - C’est l’inverse d’une dérivation
  - Shift:
    - ajoute le token courant à la phrase ω
  - En pratique, c’est construire une dérivation à droite

L4: Analyse Syntaxique (2)

- Théorie
  - notion de handle
    - on a: $S \Rightarrow^* \alpha A \omega \Rightarrow \alpha \beta \omega$
    - alors: $A \rightarrow \beta$ et la position après α est une handle
    - note : ω ne contient que des terminaux (c'est l'entrée non encore traitée)
    - note2: α peut contenir des terminaux et des non-terminaux
  - Principe du parse
    - Commencer par ω
    - A chaque étape, chercher la handle β et appliquer la réduction $A \rightarrow \beta$
      - (remplacer β par A)
    - Jusqu'à atteindre S

L4: Analyse Syntaxique (2)

- Shift-Reduce parsing (l'implémentation)
  - Une pile, un buffer d'entrée (contenant les tokens)
  - État initial:

Stack	Input
$	ω $

  - État final

Stack	Input
$ S	$

L4: Analyse Syntaxique (2)

- Exemple
  - 3 + 4

Stack	Input	Action
$	3 + 4 $	Shift
$ 3	+ 4 $	Reduce F → <number>
$ F	+ 4 $	Reduce T → F
$ T	+ 4 $	Reduce E → T
$ E	+ 4 $	Shift
$ E +	4 $	Shift
$ E + 4	$	Reduce F → <number>
$ E + F	$	Reduce T → F
$ E + T	$	Reduce E → E + T
$ E	$	Accept

L4: Analyse Syntaxique (2)

- Actions d'un parseur shift-reduce:
  - Shift
    - met le token suivant sur la pile
  - Reduce
    - considère une chaîne β sur la pile; le haut de la pile est le dernier caractère de β. Trouver le début de β dans la pile et remplacer par un non-terminal
  - Accept
    - signifie que la chaîne est acceptée
  - Error
    - retourne une erreur

L4: Analyse Syntaxique (2)

- Principe:
  - Shift jusqu'à ce que on ait une handle sur la pile
  - Reduce pour remplacer cette handle par un non-terminal
  - La handle n'est jamais à l'intérieur de la pile: elle se trouve toujours sur le dessus
    - Le dernier symbole de la handle est le haut de la pile

L4: Analyse Syntaxique (2)

- Démonstration
  - Cas 1 : la partie droite de A contient des non-terminaux
    - $A \rightarrow \beta\ B\ y$; $B \rightarrow \gamma$
    - $S \Rightarrow^* \alpha\ A\ z \Rightarrow \alpha\ \beta\ B\ y\ z \Rightarrow \alpha\ \beta\ \gamma\ y\ z$
  - Parseur

Stack	Input
$ α β γ	y z $
$ α β B	y z $
$ α β B y	z $
$ α A	z $

L4: Analyse Syntaxique (2)

- Démonstration
  - Cas 2: la partie droite de A ne contient que des terminaux
    - $A \rightarrow y$; $B \rightarrow \gamma$
    - $S \Rightarrow^* \alpha\ B\ x\ A\ z \Rightarrow \alpha\ B\ x\ y\ z \Rightarrow \alpha\ \gamma\ x\ y\ z$
  - Parseur

Stack	Input
$ α γ	x y z $
$ α B	x y z $
$ α B x y	z $
$ α B x A	z $

L4: Analyse Syntaxique (2)

- Conflits:
  - Deux types: shift/reduce et reduce/reduce
  - Shift/reduce
    - Ambiguité type if/else du C

stmt → if (expr) stmt else stmt
→ if (expr) stmt

Stack	Input
... if (expr) stmt	else ... $

L4: Analyse Syntaxique (2)

- Conflits:
  - Deux types: shift/reduce et reduce/reduce
  - Shift/reduce
    - Solution
      - Privilégier le shift au reduce (greedy: créer la phrase la plus longue)
      - Conforme à la sémantique du langage C (par exemple).

Stack	Input
... if (expr) stmt	else ... $

L4: Analyse Syntaxique (2)

- Conflits:
  - Deux types: shift/reduce et reduce/reduce
  - reduce/reduce

stmt → id (parameter) "function call"

parameter → id

expr → id

→ id (expr) "Array access"

Stack	Input
...id (id	) ... $

L4: Analyse Syntaxique (2)

- Conflits:
  - Deux types: shift/reduce et reduce/reduce
  - reduce/reduce
    - Solution: avoir un autre token fid pour les fonctions (utiliser une table des symboles)

      stmt → fid (parameter) "function call"
    - Un parseur shift/reduce sait utiliser la pile entière pour choisir le reduce

Stack	Input
...fid (id	) ... $

L4: Analyse Syntaxique (2)

- Implémentation: LR
  - LR(k)
    - "L" left to right: scan de gauche à droite de l'entrée
    - "R" rightmost : dérivation à droite
      - construite à l'envers
    - k: nombre de tokens de lookahead
      - en pratique, k = 0 ou 1
  - Parseurs basés sur des tables

L4: Analyse Syntaxique (2)

- Intérêt de LR ?
  - Les parseurs LR peuvent reconnaître presque toutes les constructions des langages de programmation
  - Le parsing LR est la méthode la plus générale de shift/reduce sans backtracking, et la plus efficace
  - Un parseur LR peut détecter une erreur de syntaxe au plus tôt
  - Les grammaires LR(k) sont un superset des grammaires LL(k)
- Défaut:
  - Construire la table est trop complexe à faire à la main
  - Éventuellement on peut écrire à la main: parseurs récursifs ascendants

L4: Analyse Syntaxique (2)

- Un parseur LR sait quand faire shift ou reduce via un automate
- Chaque état de cet automate correspond à un ensemble d’"items"
- Un "item" est une production avec un point dans le body
  - Par exemple, pour A → α β δ, les items possibles sont:
    - A → . α β δ
    - A → α . β δ
    - A → α β . δ
    - A → α β δ .

L4: Analyse Syntaxique (2)

- Pour A → ε , on a un seul item: A → .

- Signification:
  - Si on est à l'item A → α . β δ
    - alors nous avons dérivé α dans l'entrée et nous nous attendons à voir β et δ
  - Si on est à l'item A → α β δ .
    - Alors on a dérivé α, β et δ, et il est temps de réduire à A

- Construction:
  - L'automate LR(0)

L4: Analyse Syntaxique (2)

- Automate LR(0)
  - Grammaire augmentée G’
    - Si G est une grammaire avec comme symbole de départ S
    - Alors G’ est une grammaire augmentée avec
      - S’ symbole de départ
      - Et la production S’ → S
  - Deux fonctions
    - CLOSURE
    - GOTO

L4: Analyse Syntaxique (2)

- CLOSURE(I):
  - Démarrer avec I, un ensemble d'items
  - Si $A \rightarrow \alpha . B \beta$ est dans CLOSURE(I) et $B \rightarrow \gamma$ une production,
    - alors ajouter $B \rightarrow . \gamma$ à CLOSURE(I) s'il n'y est pas déjà.
  - Continuer jusqu'à ce qu'aucun nouvel item ne puisse être ajouté
- GOTO(I, X) (I ensemble d'items, X symbole de G)
  - C'est la closure de tous les items $A \rightarrow \alpha X . \beta$,
  - avec $A \rightarrow \alpha . X \beta \in$ CLOSURE(I)

L4: Analyse Syntaxique (2)

- Exemple:

  E → E + T | T

  T → T * F | F

  F → (E) | <number>

  - Augmentée

    S → E

  - Items

    S → . E

    S → E .

E → . E + T

E → E . + T

E → E + . T

E → E + T .

E → . T

E → T .

…

L4: Analyse Syntaxique (2)

- Question
  - Construire CLOSURE et GOTO ...
  - Commencer avec I_0 :

    S → . E
  - Compléter la closure (si A → α . B β, et B → γ, alors...)

    E → . E + T

    E → . T

    T → . T * F

    T → . F

    F → . (E)

    F → . <number>

L4: Analyse Syntaxique (2)

- Question
  - Construire CLOSURE et GOTO …
  - Continuer avec GOTO
    - GOTO(I_0, E):

      S → E .

      E → E . + T
    - GOTO(I_0, <number>)

      F → <number> .

L4: Analyse Syntaxique (2)

- Utilisation
  - Construction d'un automate LR(0) à partir de CLOSURE et GOTO
  - Chaque item set (I_0, I_1, I_2) est un état
  - Chaque GOTO(I_x, c) est un shift (c un terminal)
  - Chaque GOTO(I_x, A) est une réduction (A un non-terminal)
    - Et si I_n = GOTO(I_x, A) alors l'état après réduction est I_n