Programming Language Syntax: Top-down Parsing

Read: Scott, Chapter 2.3.2 and 2.3.3

1

Lecture Outline

- *Top-down parsing*
  - *Predictive parsing*
  - *LL(1) parsing table*
  - *FIRST, FOLLOW, and PREDICT sets*
  - *LL(1) grammars*
- *Bottom-up parsing*
  - *A brief overview, no detail*

2

Predictive Parsing

- "Predicts" production to apply based on one or more lookahead token(s)
- Predictive parsers work with LL(k) grammars
  - First L stands for "left-to-right" scan of input
  - Second L stands for leftmost derivation
    - Parse corresponds to leftmost derivation
  - k stands for "need k tokens of lookahead to predict"
- We are interested in LL(1)

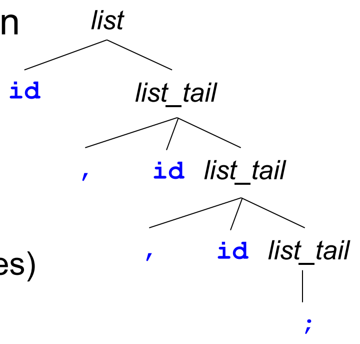
Question

list $\rightarrow$ `id` *list_tail*
list_tail $\rightarrow$ `,` `id` *list_tail* | `;`

- Can we always predict (i.e., for <u>any</u> input) what production to apply, based on just one token of lookahead?

`id , id , id ;`

  - Yes, there is at most one choice (i.e., at most one production applies)
  - This grammar is an LL(1) grammar



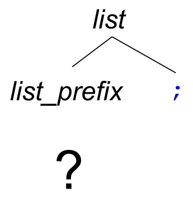
Question

list → *list_prefix* **;**

$list\_prefix \to \underbrace{list\_prefix\ ,\ \mathtt{id}}_{\substack{\text{id, id} \\ \text{id, id, id} \\ \text{etc...}}} \mid \underbrace{\mathtt{id}}_{\underline{\text{id}}}$

- A new grammar
- What language does it generate?
  - Same, comma-separated lists of ids
- Can we predict based on one token of lookahead?

```
id , id , id ;
↑
```



Predictive Parsing

- "Predicts" production to apply based on one or more lookahead token(s)
  - Parser always gets it right!
  - There is no need to backtrack, undo expansion and try a different production
- Predictive parsers work with LL(k) grammars

Predictive Parsing

- Expression grammar:
  - Not LL(1)
- Unambiguous version:
  - Still not LL(1). Why?

expr → *expr* **+** *expr* } id*id+id
  | *expr* ***** *expr* } id*id
  | **id**

id*id +id   id*id

expr → *expr* **+** *term* | *term*
term → *term* ***** **id** | **id**

- LL(1) version:

SPECIAL SYMBOL

expr → *term term_tail*
term_tail → **+** *term term_tail* | **ε**
term → **id** *factor_tail*
factor_tail → ***** **id** *factor_tail* | **ε**

Exercise

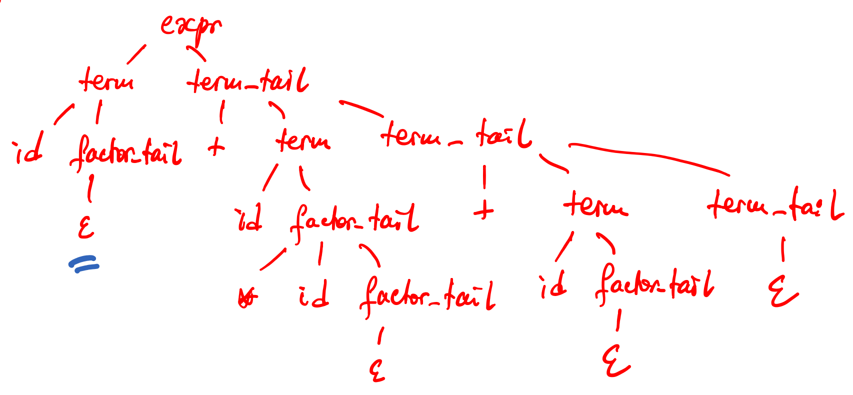
expr → *term term_tail*
term_tail → **+** *term term_tail* | **ε**
term → **id** *factor_tail*
factor_tail → ***** **id** *factor_tail* | **ε**

- Draw parse tree for expression

term-tail

id + id * id + id

term
=



Predictive Recursive Descent

start → *expr* $$
expr → *term term_tail*
term → id *factor_tail*
term_tail → + *term term_tail* | ε
factor_tail → * id *factor_tail* | ε

```
start()
    case lookahead() of
        id:  expr(); match($$)          ($$ - end-of-input marker)
        otherwise PARSE_ERROR

expr()
    case lookahead() of
        id:  term(); term_tail()
        otherwise PARSE_ERROR

term_tail()
    case lookahead() of          Predicting production term_tail → + term  term_tail
        +:  match('+'); term(); term_tail()
        $$:  skip                Predicting epsilon production term_tail → ε
        otherwise: PARSE_ERROR
```


Predictive Recursive Descent

start → *expr* $$
expr → *term term_tail*
term → id *factor_tail*
term_tail → + *term term_tail* | ε
factor_tail → * id *factor_tail* | ε

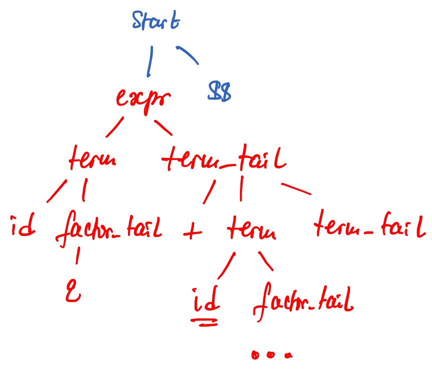
```
term()
    case lookahead() of
        id: match('id'); factor_tail()
        otherwise: PARSE_ERROR

factor_tail()
    case lookahead() of          Predicting production factor_tail → *id factor_tail
        *: match('*'); match('id'); factor_tail();
        +,$$: skip
        otherwise PARSE_ERROR    Predicting production factor_tail → ε
```

Predictive Recursive Descent

- E.g., parse `id + id * id $$`

When matching this id, the following recursive descend procedures are on the call stack: start(), expr(), term_tail(), term().

```
start()
  expr()
    term() ✓
      match(id) ✓
      factor_tail() ✓   ε
    term_tail()
      match(+)
      term()
        match(id)
        factor_tail()
          ...
```



Programming Languages CSCI 4430, A. Milanova 11

11

LL(1) Parsing Table

- But how does the parser "predict"?
  - E.g., how does the parser know to expand a *factor_tail* by *factor_tail* → **ε** on **+** and **$$**?
- It uses the LL(1) parsing table
  - One dimension is nonterminal to expand
  - Other dimension is lookahead token
    - We are interested in one token of lookahead
  - Entry "nonterminal on token" contains the production to apply or contains nothing

Programming Languages CSCI 4430, A. Milanova 12

12

LL(1) Parsing Table

- One dimension: nonterminal to expand
- Other dimension: lookahead token

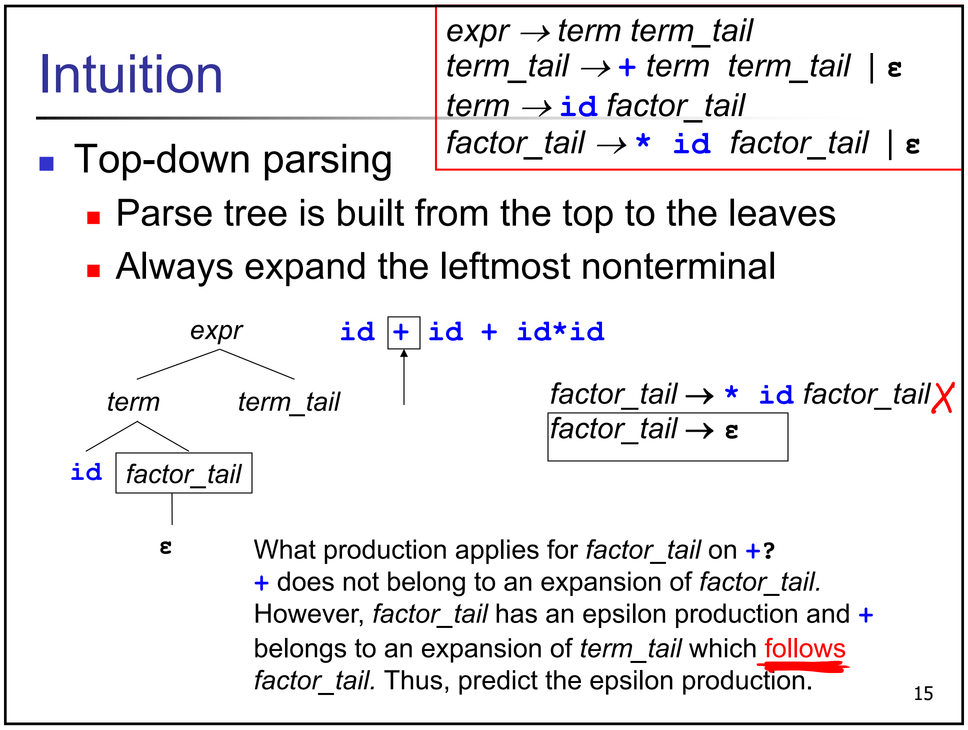
	a
A	α

- E.g., entry "nonterminal *A* on terminal **a**" contains production *A* → α
- Meaning: when parser is at nonterminal *A* and lookahead token is **a**, then parser expands *A* by production *A* → α

LL(1) Parsing Table

start → *expr* **$$**
expr → *term term_tail*
term → **id** *factor_tail*
term_tail → **+** *term term_tail* | **ε**
factor_tail → ***** **id** *factor_tail* | **ε**

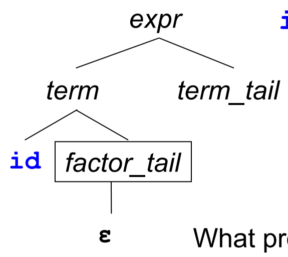
	id	**+**	*****	**$$**
start	expr $$	—	—	—
expr	term term_tail	—	—	—
term_tail	—	+ term term_tail	—	ε
term	id factor_tail	—	—	—
factor_tail	—	ε	* id factor_tail	ε



Intuition

expr → *term term_tail*
term_tail → **+** *term term_tail* | **ε**
term → **id** *factor_tail*
factor_tail → ***** **id** *factor_tail* | **ε**

- Top-down parsing
  - Parse tree is built from the top to the leaves
  - Always expand the leftmost nonterminal



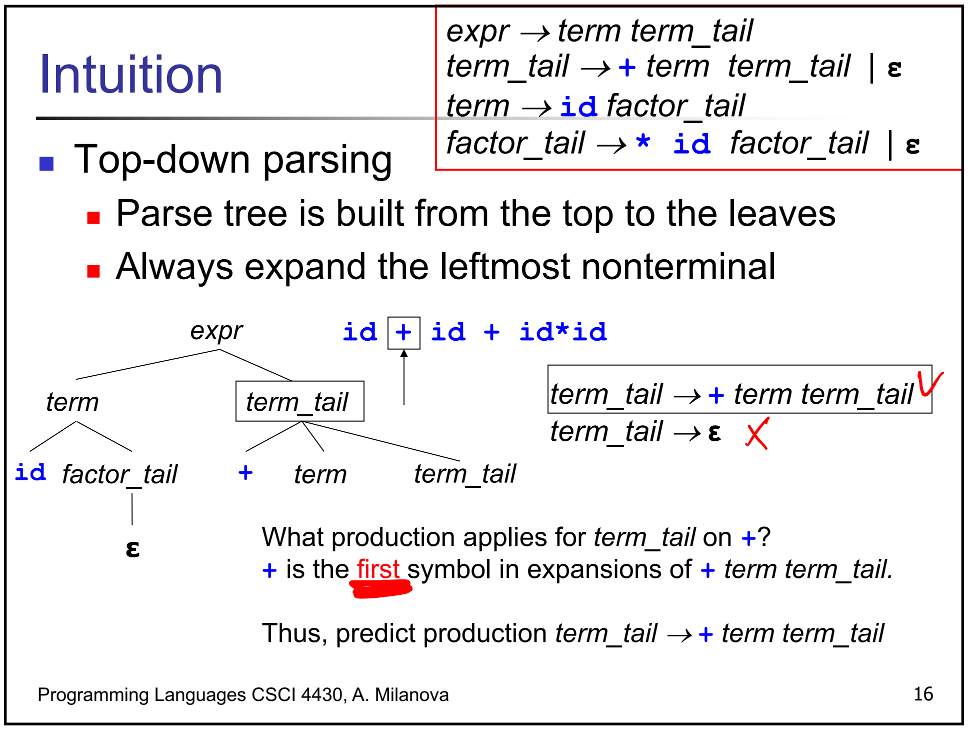
id $\boxed{+}$ **id + id*id**

factor_tail → ***** **id** *factor_tail* ✗
$\boxed{\textit{factor\_tail} \rightarrow \varepsilon}$

What production applies for *factor_tail* on **+**?
+ does not belong to an expansion of *factor_tail.*
However, *factor_tail* has an epsilon production and **+** belongs to an expansion of *term_tail* which follows *factor_tail.* Thus, predict the epsilon production.

15

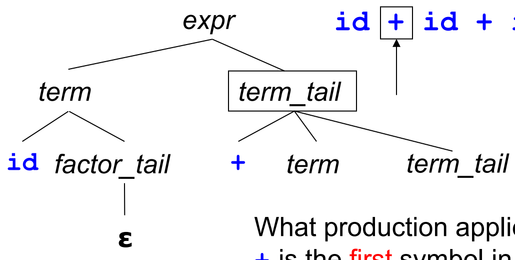
15



Intuition

expr → *term term_tail*
term_tail → **+** *term term_tail* | **ε**
term → **id** *factor_tail*
factor_tail → ***** **id** *factor_tail* | **ε**

- Top-down parsing
  - Parse tree is built from the top to the leaves
  - Always expand the leftmost nonterminal



id $\boxed{+}$ **id + id*id**

$\boxed{\textit{term\_tail} \rightarrow + \textit{ term term\_tail}}$ ✓
term_tail → **ε** ✗

What production applies for *term_tail* on **+**?
+ is the first symbol in expansions of **+** *term term_tail.*

Thus, predict production *term_tail* → **+** *term term_tail*

16

LL(1) Tables and LL(1) Grammars

- We can construct an LL(1) parsing table for any context-free grammar
  - In general, the table will contain multiply-defined entries. That is, for some nonterminal and lookahead token, more than one production applies
- A grammar whose LL(1) parsing table has no multiply-defined entries is said to be LL(1) grammar
  - LL(1) grammars are a very special subclass of context-free grammars. Why?

FIRST and FOLLOW sets

- Let α be any sequence of nonterminals and terminals
  $\Rightarrow^*$ IS ZERO OR MORE APPLICATIONS OF GRAMMAR PRODUCTIONS
  - FIRST(α) is the set of terminals **a** that begin the strings derived from α. E.g., *expr* **$$** $\Rightarrow^*$ **id**..., thus **id** in FIRST(*expr* **$$**)
  - If there is a derivation α $\Rightarrow^*$ **ε**, then **ε** is in FIRST(α)
- Let *A* be a nonterminal
  - FOLLOW(*A*) is the set of terminals **b** (including special end-of-input marker **$$**) that can appear immediately to the right of *A* in some sentential form:

    start $\Rightarrow^*$ …*A***b**… $\Rightarrow^*$…

Applying expr => term term_tail

expr $$ => term term_tail $$ =>
id factor_tail term_tail $$

Computing FIRST

Notation:
α is an arbitrary sequence of terminals and nonterminals

- Apply these rules until no more terminals or **ε** can be added to any FIRST(α) set

  (1) If α starts with a terminal **a**, then FIRST(α) = { **a** }

  (2) If α is a nonterminal *X*, where $X \rightarrow \varepsilon$, then add **ε** to FIRST(α)

  (3) If α is a nonterminal $X \rightarrow \overset{\varepsilon}{\underline{Y_1}}Y_2 \ldots Y_k$ then add **a** to FIRST(*X*) if for some *i*, **a** is in FIRST(Y_i) and **ε** is in all of FIRST(Y_1), … FIRST(Y_{i-1}). If **ε** is in all of FIRST(Y_1), … FIRST(Y_k), add **ε** to FIRST(*X*).

  - Everything in FIRST(Y_1) - **{ε}** is surely in FIRST(*X*)
  - If Y_1 does not derive **ε**, then we add nothing more; Otherwise, we add FIRST(Y_2) - **{ε}**, and so on

  Similarly, if α is $Y_1Y_2 \ldots Y_k$, we'll repeat the above

Warm-up Exercise

start → *expr* **$$**
expr → *term term_tail*
term → **id** *factor_tail*
term_tail → **+** *term term_tail* | **ε**
factor_tail → ***** **id** *factor_tail* | **ε**

FIRST(*term*) = { **id** }

FIRST(*expr*) = {id}

FIRST(*start*) = {id}

FIRST(*term_tail*) = {+ , ε}

FIRST(**+** *term term_tail*) = { +}

FIRST(*factor_tail*) = { *, ε}

Exercise

$\boxed{\underset{A\to\varepsilon}{Ay \Rightarrow y}}$ → E.g. we can derive string y from $\underline{Ay}$

start → *S* $$	*B* → **z** *S* | **ε**
S → **x** *S* | *A* **y**	*C* → **v** *S* | **ε**
A → *BCD* | **ε**	*D* → **w** *S*

$Ay \Rightarrow BCDy \Rightarrow CDy \Rightarrow Dy \Rightarrow wSy \Rightarrow wAyy \Rightarrow \underline{w}yy$

Compute FIRST sets:

FIRST(**x** *S*) = {x}	FIRST(*S*) = {x, y, z, v, w}
FIRST(*A* **y**) = {y, z, v, w}	FIRST(*A*) = {ε, z, v, w}
FIRST(*BCD*) = {z, v, w}	FIRST(*B*) = {z, ε}
FIRST(**z** *S*) = {z}	FIRST(*C*) = {v, ε}
FIRST(**v** *S*) = {v}	FIRST(*D*) = {w}
FIRST(**w** *S*) = {w}	

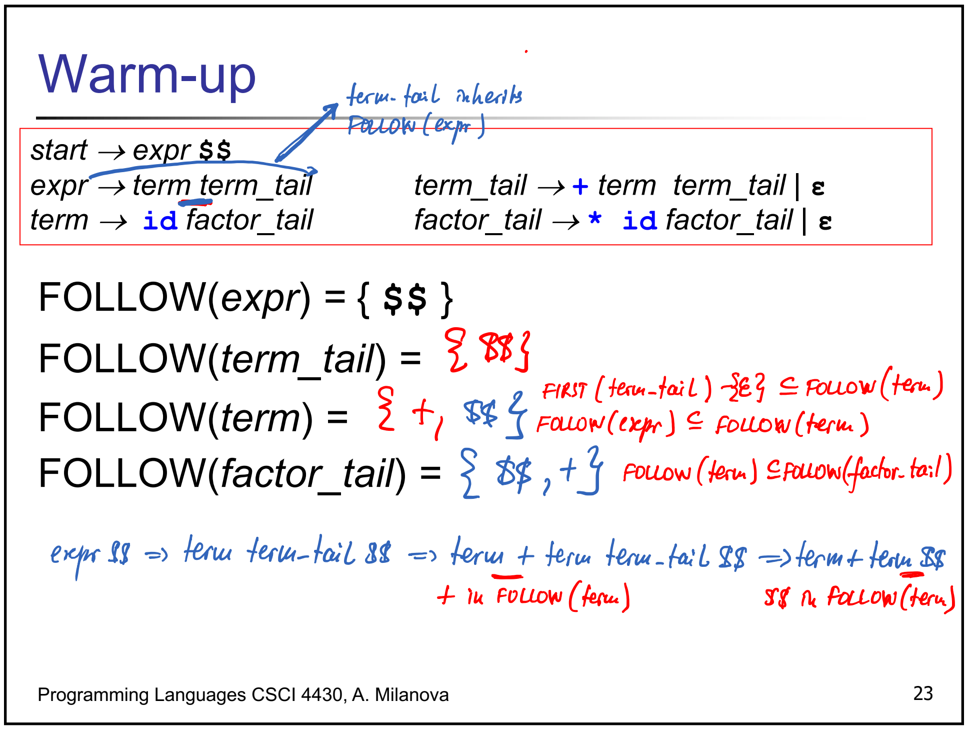
Computing FOLLOW

Notation:
A,*B*,*S* are nonterminals.
α,β are arbitrary sequences of terminals and nonterminals.

- Apply these rules until nothing can be added to any FOLLOW(*A*) set
  - (1) If there is a production *A* → α*B*β, then everything in FIRST(β) except for **ε** should be added to FOLLOW(*B*)
  - (2) If there is a production *A* → α*B*, or a production *A* → α*B*β, where FIRST(β) contains **ε**, then everything in FOLLOW(*A*) should be added to FOLLOW(*B*)

Because: $start \Rightarrow^* \ldots Ab\ldots \Rightarrow \ldots \alpha Bb\ldots$

Thus b ∈ FOLLOW(A) must be in FOLLOW(B) as well.



Warm-up

term_tail inherits FOLLOW(expr)

start → *expr* **$$**
expr → *term term_tail*
term → **id** *factor_tail*
term_tail → **+** *term term_tail* | **ε**
factor_tail → ***** **id** *factor_tail* | **ε**

FOLLOW(*expr*) = { **$$** }

FOLLOW(*term_tail*) = { $$ }

FOLLOW(*term*) = { +, $$ } FIRST(term_tail) − {ε} ⊆ FOLLOW(term) FOLLOW(expr) ⊆ FOLLOW(term)

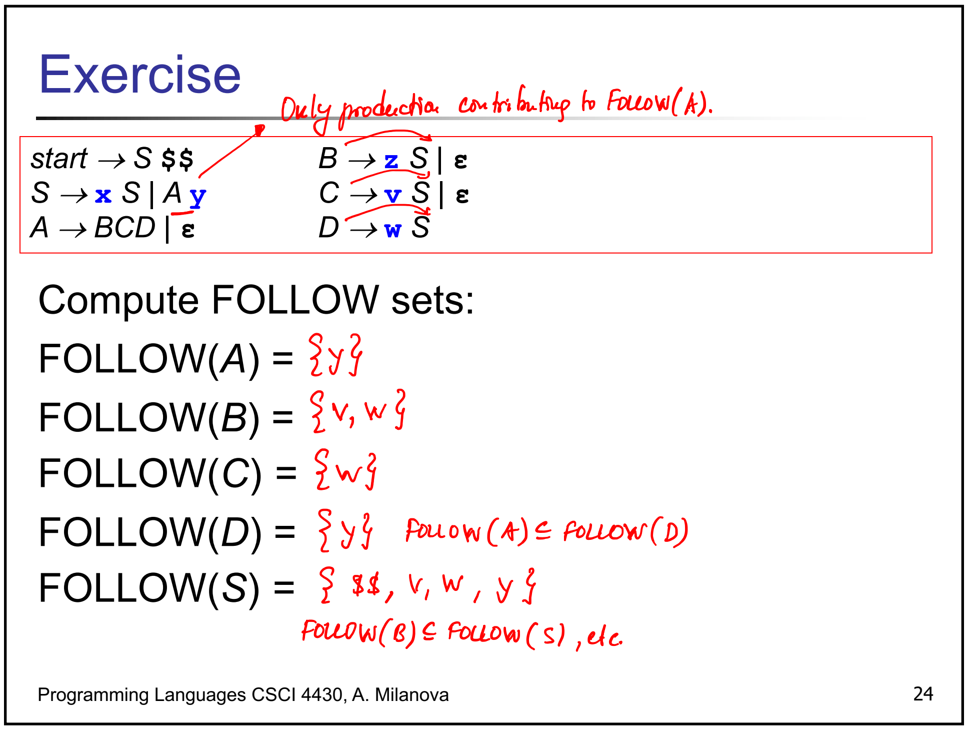
FOLLOW(*factor_tail*) = { $$, + } FOLLOW(term) ⊆ FOLLOW(factor_tail)

expr $$ ⇒ term term_tail $$ ⇒ term + term term_tail $$ ⇒ term + term $$

+ in FOLLOW(term) $$ in FOLLOW(term)

Programming Languages CSCI 4430, A. Milanova 23

23



Exercise

Only production contributing to FOLLOW(A).

start → *S* $$
S → **x** *S* | *A* **y**
A → *BCD* | **ε**
B → **z** *S* | **ε**
C → **v** *S* | **ε**
D → **w** *S*

Compute FOLLOW sets:

FOLLOW(*A*) = {y}

FOLLOW(*B*) = {v, w}

FOLLOW(*C*) = {w}

FOLLOW(*D*) = {y} FOLLOW(A) ⊆ FOLLOW(D)

FOLLOW(*S*) = { $$, v, w, y }

FOLLOW(B) ⊆ FOLLOW(S), etc.

Programming Languages CSCI 4430, A. Milanova 24

24

PREDICT Sets

We predict $A \to \alpha$ on every terminal in FIRST(α), i.e. α can derive a string beginning with that terminal

$$\text{PREDICT}(A \to \alpha) = \begin{cases} \text{FIRST}(\alpha) & \text{if } \alpha \text{ does not derive } \varepsilon \\ (\text{FIRST}(\alpha) - \{\varepsilon\}) \cup \text{FOLLOW}(A) & \text{if } \alpha \text{ derives } \varepsilon \end{cases}$$

α can derive a string beginning with terminal $b \in \text{FIRST}(\alpha) - \{\varepsilon\}$.

α can derive ε and what follows can derive a string beginning with terminal $b \in \text{FOLLOW}(A)$.

Constructing LL(1) Parsing Table

- Algorithm uses PREDICT sets:

```
foreach production $A \to \alpha$ in grammar $G$
  foreach terminal $\mathbf{a}$ in PREDICT($A \to \alpha$)
    add $A \to \alpha$ into entry parse_table[$A$,$\mathbf{a}$]
```

- If each entry in parse_table contains at most one production, then *G* is said to be LL(1)

Exercise

start → *S* **$$**	*B* → **z** *S* | **ε**
S → **x** *S* | *A* **y**	*C* → **v** *S* | **ε**
A → *BCD* | **ε**	*D* → **w** *S*

Compute PREDICT sets:

PREDICT(*S* → **x** *S*) = {x}   ONLY ON x.

PREDICT(*S* → *A* **y**) = {y, z, v, w}   ONLY ON FIRST(Ay).

PREDICT(*A* → *BCD*) = {z, v, w}   Note NO CONFLICTS FOR A: ON z, v, w, PREDICT A→BCD. ON y, PREDICT A→ε.

PREDICT(*A* → **ε**) = {y}

… *etc*…

GRAMMAR IS LL(1).

Writing an LL(1) Grammar

- Most context-free grammars are not LL(1) grammars
- Obstacles to LL(1)-ness
  - Left recursion is an obstacle. Why?

    expr → *expr* **+** *term* | *term*
    term → *term* ***** **id** | **id**

  - Common prefixes are an obstacle. Why?

    stmt → **if b then** *stmt* **else** *stmt* |
    if b then *stmt* |
    a

Removal of Left Recursion

- Left recursion can be removed from a grammar mechanically
- Started from this left recursive expression grammar:

$expr \rightarrow expr$ **+** $term \mid term$
$term \rightarrow term$ ***** **id** $\mid$ **id**

- After removal of left recursion, we obtain this equivalent grammar, which is LL(1):

$expr \rightarrow term\ term\_tail$
$term\_tail \rightarrow$ **+** $term\ \ term\_tail \mid \varepsilon$
$term \rightarrow$ **id** $factor\_tail$
$factor\_tail \rightarrow$ ***** **id** $factor\_tail \mid \varepsilon$

Removal of Common Prefixes

- Common prefixes can be removed mechanically as well by using left-factoring
- Original if-then-else grammar:

$stmt \rightarrow$ **if b then** *stmt* **else** *stmt* |
if b then *stmt* |
a

- After left-factoring:

$stmt \rightarrow$ **if b then** *stmt* *else_part* | **a**
$else\_part \rightarrow$ **else** *stmt* | ε

Exercise

start → *stmt* **$$**
stmt → **if b then** *stmt else_part* | **a**
else_part → **else** *stmt* | **ε**

- Compute FIRSTs:

FIRST(*stmt* **$$**), FIRST(**if b then** *stmt else_part*),
FIRST(**a**), FIRST(**else** *stmt*)

- Compute FOLLOW:

FOLLOW(*else_part*)

- Compute PREDICT sets for all 5 productions and fill in the LL(1) parsing table. Is the grammar LL(1)?

Exercise

start → *stmt* **$$**
stmt → **if b then** *stmt else_part* | **a**
else_part → **else** *stmt* | **ε**

- Compute FIRSTs:

FIRST(*stmt* **$$**) =

FIRST(**if b then** *stmt else_part*) =

FIRST(**a**) =

FIRST(**else** *stmt*) =

Exercise

start → *stmt* **$$**
stmt → **if b then** *stmt else_part* | **a**
else_part → **else** *stmt* | **ε**

- Compute FOLLOW:

FOLLOW(*else_part*) =

Exercise

start → *stmt* **$$**
stmt → **if b then** *stmt else_part* | **a**
else_part → **else** *stmt* | **ε**

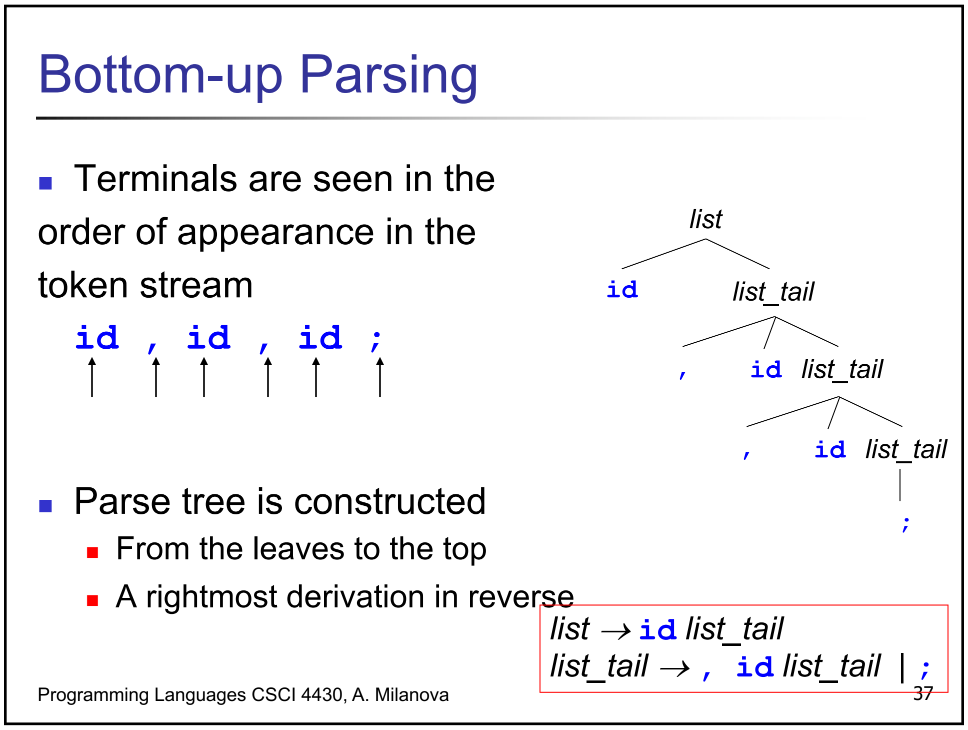
- Construct the LL(1) parsing table

- Is the grammar LL(1)?

Exercise

Lecture Outline

- *Top-down parsing*
  - *Predictive parsing*
  - *LL(1) parsing table*
  - *FIRST, FOLLOW, and PREDICT sets*
  - *LL(1) grammars*
- *Bottom-up parsing*
  - *A brief overview, no detail*

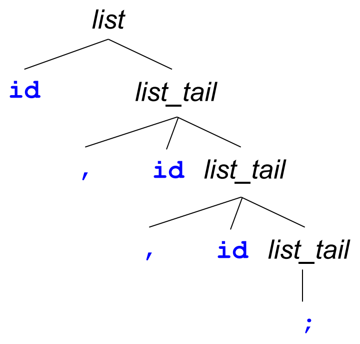


Bottom-up Parsing

- Terminals are seen in the order of appearance in the token stream

id , id , id ;

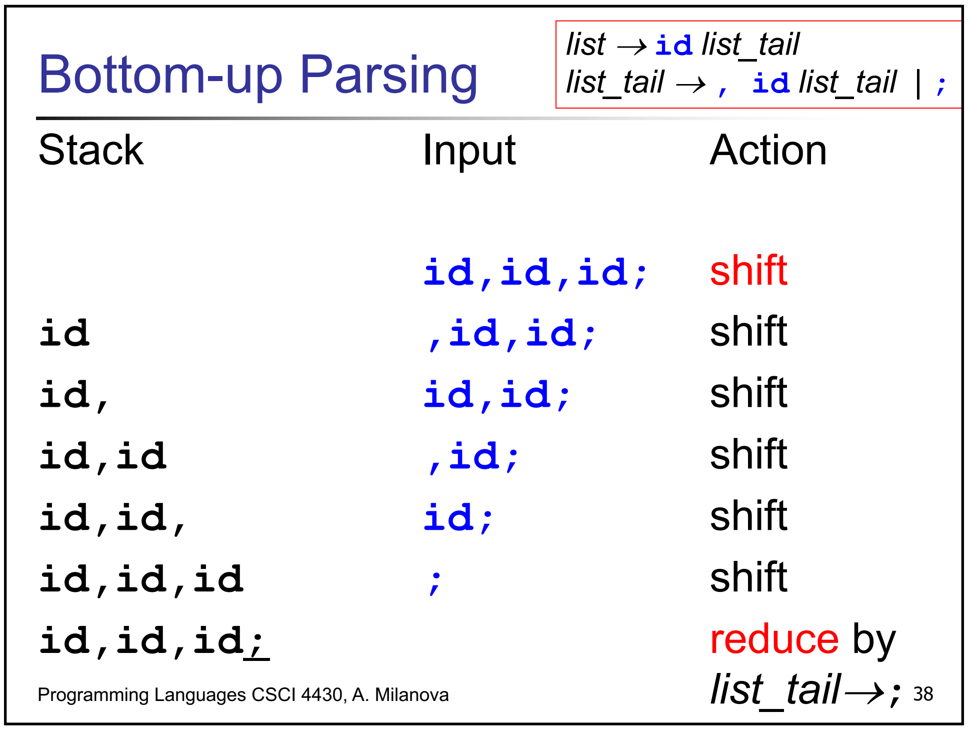
↑ ↑ ↑ ↑ ↑ ↑



- Parse tree is constructed
  - From the leaves to the top
  - A rightmost derivation in reverse

$list \rightarrow$ **id** $list\_tail$
$list\_tail \rightarrow$ **,** **id** $list\_tail$ | **;**

37



Bottom-up Parsing

$list \rightarrow$ **id** $list\_tail$
$list\_tail \rightarrow$ **,** **id** $list\_tail$ | **;**

Stack	Input	Action
	`id,id,id;`	shift
`id`	`,id,id;`	shift
`id,`	`id,id;`	shift
`id,id`	`,id;`	shift
`id,id,`	`id;`	shift
`id,id,id`	`;`	shift
`id,id,id;`		reduce by $list\_tail \rightarrow$;

38

Bottom-up Parsing

list → **`id`** *list_tail*
list_tail → **`,`** **`id`** *list_tail* | **`;`**

Stack	Input	Action
`id,id`<u>`,id` *list_tail*</u>		reduce by *list_tail* → **`,id`** *list_tail*
`id`<u>`,id` *list_tail*</u>		reduce by *list_tail* → **`,id`** *list_tail*
<u>`id` *list_tail*</u>		reduce by *list* → **`id`** *list_tail*
list		ACCEPT

Bottom-up Parsing

- Also called LR parsing
- LR parsers work with LR(k) grammars
  - L stands for "left-to-right" scan of input
  - R stands for "rightmost" derivation
  - k stands for "need k tokens of lookahead"
- We are interested in LR(0) and LR(1) and variants in between
- LR parsing is better than LL parsing!
  - Accepts larger class of languages
  - Just as efficient!

LR Parsing

- The parsing method used in practice
  - LR parsers recognize virtually all PL constructs
  - LR parsers recognize a much larger set of grammars than predictive parsers
  - LR parsing is efficient
- LR parsing variants
  - SLR (or Simple LR)
  - LALR (or Lookahead LR) – `yacc`/`bison` generate LALR parsers
  - LR (Canonical LR)
  - SLR < LALR < LR

Main Idea

- Stack $\leftarrow$ Input
- Stack: holds the part of the input seen so far
  - A string of both terminals and nonterminals
- Input: holds the remaining part of the input
  - A string of terminals
- Parser performs two actions
  - Reduce: parser pops a "suitable" production right-hand-side off top of stack, and pushes production's left-hand-side on the stack
  - Shift: parser pushes next terminal from the input on top of the stack

Example

- Recall the grammar

  $expr \rightarrow expr$ **`+`** $term \mid term$
  $term \rightarrow term$ **`*`** **`id`** $\mid$ **`id`**

  - This is not LL(1) because it is left recursive
  - LR parsers can handle left recursion!

- Consider string

  `id + id * id`

`id + id*id`

Stack	Input	Action
	`id+id*id`	shift `id`
<u>`id`</u>	`+id*id`	reduce by $term \rightarrow$ `id`
<u>*term*</u>	`+id*id`	reduce by $expr \rightarrow term$
<u>*expr*</u>	`+id*id`	shift `+`
expr`+`	`id*id`	shift `id`
expr`+`<u>`id`</u>	`*id`	reduce by $term \rightarrow$ `id`

$expr \rightarrow expr$ **`+`** $term \mid term$
$term \rightarrow term$ **`*`** **`id`** $\mid$ **`id`**

id + id*id

Stack	Input	Action
expr+*term*	*id	shift *
expr+*term**	id	shift id
expr+$\underline{term*\texttt{id}}$		reduce by *term*→*term* *id
$\underline{expr+term}$		reduce by *expr*→*expr*+*term*
expr		ACCEPT, SUCCESS

expr → *expr* + *term* | *term*
term → *term* * id | id

id + id*id

Sequence of reductions performed by parser

- id+id*id
- *term*+id*id
- *expr*+id*id
- *expr*+*term**id
- *expr*+*term*
- *expr*

• A rightmost derivation in reverse

• The stack (e.g., *expr*) concatenated with remaining input (e.g., +id*id) gives a sentential form (*expr*+id*id) in the rightmost derivation

expr → *expr* + *term* | *term*
term → *term* * id | id

The End