Programming Language Syntax: Top-down Parsing

Read: Scott, Chapter 2.3.2 and 2.3.3

Lecture Outline

- Top-down parsing (also called LL parsing)
  - LL(1) parsing table
  - FIRST, FOLLOW, and PREDICT sets
  - LL(1) grammars

- Bottom-up parsing (also called LR parsing)
  - A brief overview, no detail

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 $A \rightarrow \alpha$
- Meaning: when parser is at nonterminal A and lookahead token is **`a`**, then parser expands A by production $A \rightarrow \alpha$

LL(1) Parsing Table

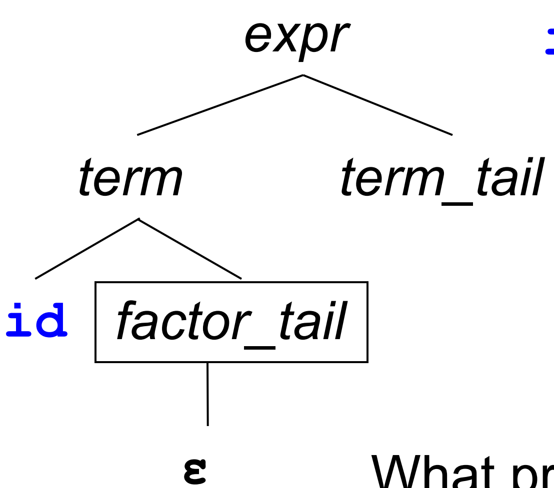
start → *expr* $$
expr → *term term_tail*
term → id *factor_tail*
term_tail → + *term term_tail* | ε
factor_tail → * id *factor_tail* | ε

	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

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

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



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

$factor\_tail \rightarrow$ ***** **id** $factor\_tail$
$\boxed{factor\_tail \rightarrow \varepsilon}$

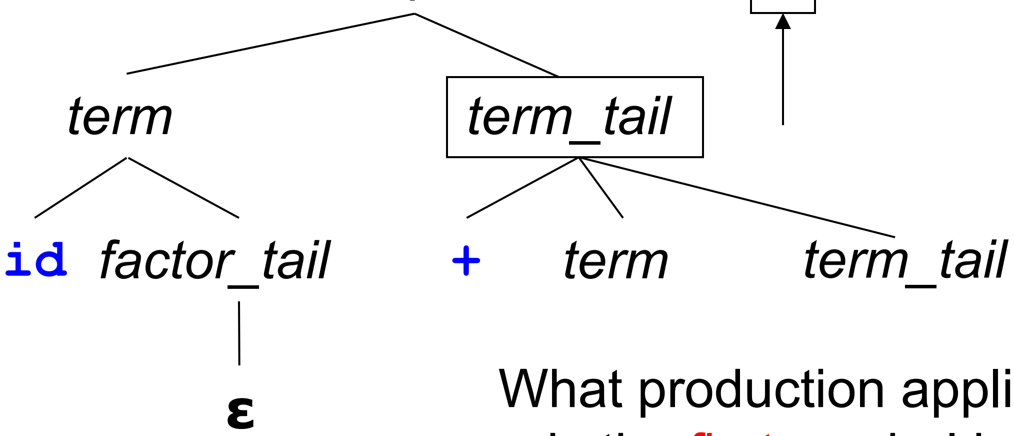
What production applies for *factor_tail* on **+?**
+ does not belong to an expansion of *factor_tail.*
However, *factor_tail* has an epsilon production and **+** belongs to an expansion of *term_tail* which follows *factor_tail.* Thus, predict the epsilon production.

Intuition

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

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

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



$\boxed{term\_tail \rightarrow \textbf{+}\ term\ term\_tail}$
$term\_tail \rightarrow$ **ε**

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

Thus, predict production $term\_tail \rightarrow$ **+** *term term_tail*

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
  - 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`… $\Rightarrow^*$…

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 $X \rightarrow$ **ε**, then add **ε** to FIRST(α)
  - (3) If α is a nonterminal $X \rightarrow Y_1 Y_2 \ldots Y_k$ 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) is surely in FIRST(X)
    - If Y_1 does not derive **ε**, then we add nothing more; Otherwise, we add FIRST(Y_2), and so on
  - Similarly, if α is $Y_1 Y_2 \ldots Y_k$, we'll repeat the above

Warm-up Exercise

start $\rightarrow$ *expr* **$$**

expr $\rightarrow$ *term term_tail*

term $\rightarrow$ **id** *factor_tail*

term_tail $\rightarrow$ **+** *term term_tail* | **ε**

factor_tail $\rightarrow$ ***** **id** *factor_tail* | **ε**

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

FIRST(*expr*) =

FIRST(*start*) =

FIRST(*term_tail*) =

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

FIRST(*factor_tail*) =

Exercise

$start \rightarrow S$ **\$\$**	$B \rightarrow$ **z** $S \mid \varepsilon$
$S \rightarrow$ **x** $S \mid A$ **y**	$C \rightarrow$ **v** $S \mid \varepsilon$
$A \rightarrow BCD \mid \varepsilon$	$D \rightarrow$ **w** S

Compute FIRST sets:

FIRST(**x** S) =

FIRST(A **y**) =

FIRST(BCD) =

FIRST(**z** S) =

FIRST(**v** S) =

FIRST(**w** S) =

FIRST(S) =

FIRST(A) =

FIRST(B) =

FIRST(C) =

FIRST(D) =

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 $A \rightarrow \alpha B\beta$, then everything in FIRST(β) except for $\boldsymbol{\varepsilon}$ should be added to FOLLOW(B)

  (2) If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B\beta$, where FIRST($\beta$) contains $\boldsymbol{\varepsilon}$, then everything in FOLLOW(A) should be added to FOLLOW(B)

Warm-up

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*) =

FOLLOW(*factor_tail*) =

Exercise

$start \rightarrow S$ **$$**	$B \rightarrow$ **z** $S \mid \varepsilon$
$S \rightarrow$ **x** $S \mid A$ **y**	$C \rightarrow$ **v** $S \mid \varepsilon$
$A \rightarrow BCD \mid \varepsilon$	$D \rightarrow$ **w** S

Compute FOLLOW sets:

FOLLOW(A) =

FOLLOW(B) =

FOLLOW(C) =

FOLLOW(D) =

FOLLOW(S) =

PREDICT Sets

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

Constructing LL(1) Parsing Table

- Algorithm uses PREDICT sets:

foreach production $A \rightarrow \alpha$ in grammar G
  foreach terminal **`a`** in PREDICT($A \rightarrow \alpha$)
    add $A \rightarrow \alpha$ into entry parse_table[A,**`a`**]

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

Exercise

$start \rightarrow S\ \mathbf{\$\$}$

$S \rightarrow \mathbf{x}\ S \mid A\ \mathbf{y}$

$A \rightarrow BCD \mid \boldsymbol{\varepsilon}$

$B \rightarrow \mathbf{z}\ S \mid \boldsymbol{\varepsilon}$

$C \rightarrow \mathbf{v}\ S \mid \boldsymbol{\varepsilon}$

$D \rightarrow \mathbf{w}\ S$

Compute PREDICT sets:

PREDICT($S \rightarrow \mathbf{x}\ S$) =

PREDICT($S \rightarrow A\ \mathbf{y}$) =

PREDICT($A \rightarrow BCD$) =

PREDICT($A \rightarrow \boldsymbol{\varepsilon}$) =

… etc…

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

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

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

Removal of Common Prefixes

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

$$\begin{array}{ll} stmt \rightarrow & \underline{\textbf{if b then } stmt} \textbf{ else } stmt \mid \\ & \underline{\textbf{if b then } stmt} \mid \\ & \textbf{a} \end{array}$$

- After left-factoring:

$$\begin{array}{l} stmt \rightarrow \underline{\textbf{if b then } stmt}\ else\_part \mid \textbf{a} \\ else\_part \rightarrow \textbf{else } stmt \mid \varepsilon \end{array}$$

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
- Construct the LL(1) parsing table. Is this grammar an LL(1) grammar?

Exercise

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

- Compute FIRSTs:

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

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

FIRST(**`a`**) =

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

Exercise

$start \rightarrow stmt$ **$$**
$stmt \rightarrow$ **if b then** $stmt\ else\_part$ **|** **a**
$else\_part \rightarrow$ **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 this grammar an LL(1) grammar?

Exercise

Lecture Outline

- Top-down parsing (also called LL parsing)
  - LL(1) parsing table
  - FIRST, FOLLOW, and PREDICT sets
  - LL(1) grammars

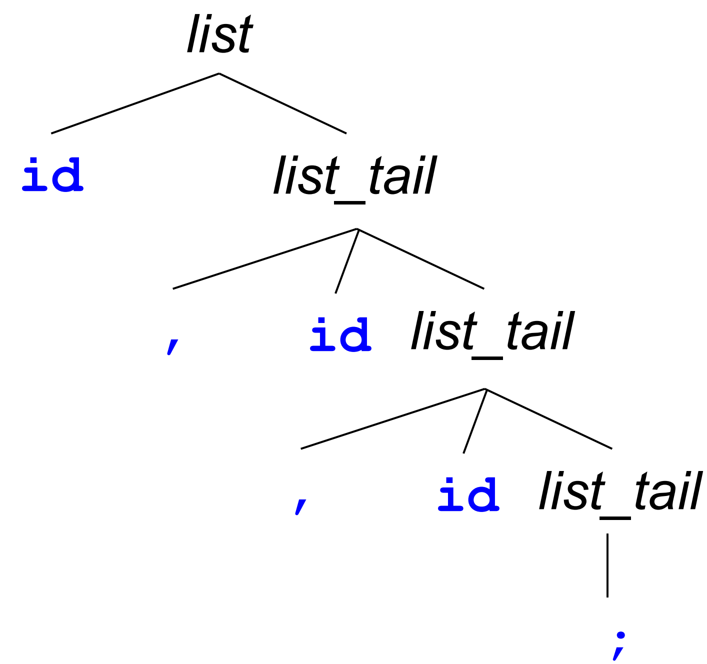
- Bottom-up parsing (also called LR parsing)
  - A brief overview, no detail

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 right-most derivation in reverse



list → **`id`** *list_tail*

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

Bottom-up Parsing

list → **id** *list_tail*
list_tail → **,** **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* → ;

Bottom-up Parsing

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

Stack	Input	Action
id,id,id *list_tail*		reduce by *list_tail* → **,id** *list_tail*
id,id *list_tail*		reduce by *list_tail* → **,id** *list_tail*
id *list_tail*		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 ← 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

$$\begin{aligned} &expr \rightarrow expr\ \mathbf{+}\ term \mid term \\ &term \rightarrow term\ \mathbf{*}\ \mathtt{id} \mid \mathtt{id} \end{aligned}$$

  - 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* → `id`
<u>*term*</u>	`+id*id`	reduce by *expr* → *term*
<u>*expr*</u>	`+id*id`	shift `+`
expr`+`	`id*id`	shift `id`
expr`+`<u>`id`</u>	`*id`	reduce by *term* → `id`

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

`id + id*id`

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