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

Recursive Descent

start → *expr* **$$**

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

term → `id` *factor_tail*   *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
        +:  match( '+' ); term(); term_tail()
        $$:  skip
        otherwise: PARSE_ERROR
```

Predicting production *term_tail* → **+** *term term_tail*

Predicting epsilon production *term_tail* → ε

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
        *: match( '*' ); match( 'id' ); factor_tail();
        +,$$: skip
        otherwise PARSE_ERROR
```

Predicting production *factor_tail* → ***id** *factor_tail*

Predicting production *factor_tail* → ε

Recursive Descent Parsing

- Parse `id + id * id $$`

```
start()
    expr()
        term()
            id  factor_tail() //return
        term_tail()
            + term()
                id  factor_tail()
                    * id  factor_tail()
        ...
```

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

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

LL(1) Parsing Table

Goal: Algorithm that constructs this LL(1) table for a given grammar.

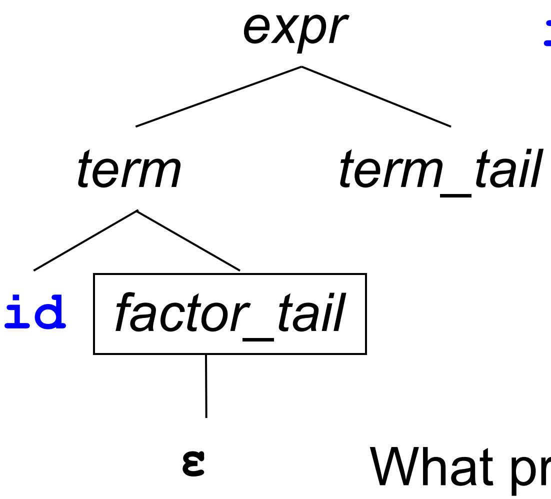
start → *expr* **$$**
expr → *term term_tail*
term → **id** *factor_tail*
term_tail → **+** *term term_tail* | **ε**
factor_tail → ***** **id** *factor_tail* | **ε**

	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 + id + id*id`

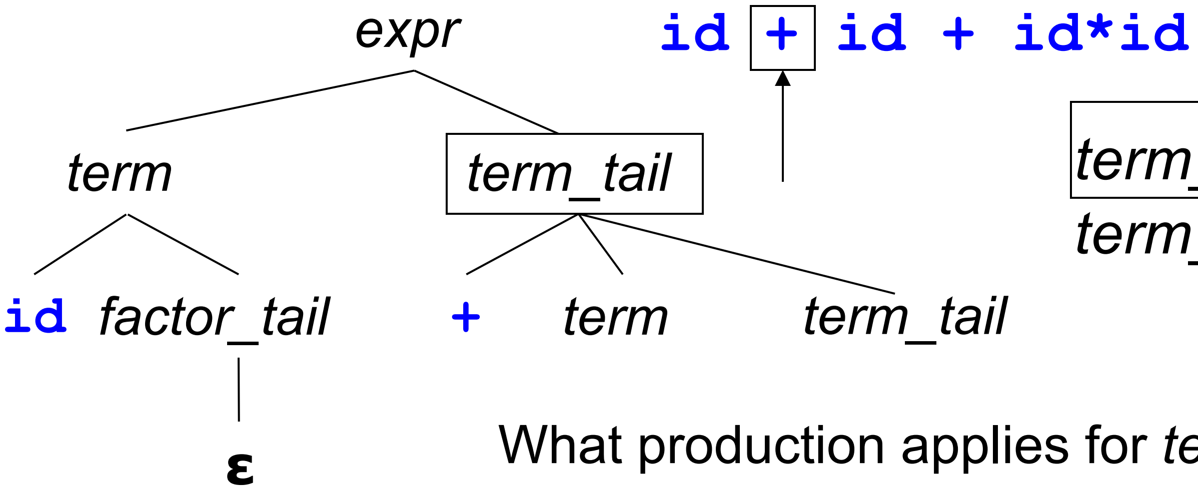
factor_tail → **`*`** **`id`** *factor_tail* ✗
factor_tail → **ε** ✓

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 → *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



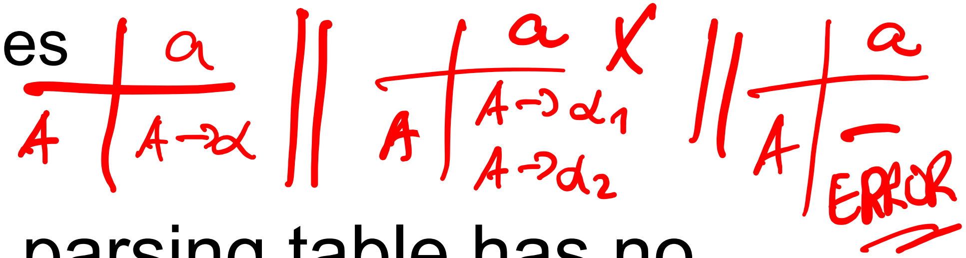
term_tail → **+** *term term_tail*
term_tail → **ε**

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*

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* **$$** ⇒* **id**..., thus **id** in FIRST(*expr* **$$**)
  - If there is a derivation α ⇒* **ε**, then **ε** is in FIRST(α)

$\alpha \Rightarrow \alpha' \Rightarrow \quad \Rightarrow \varepsilon$

- 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 ⇒* …*A***b**… ⇒*…

    $\downarrow$
    ε

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 $X \to \varepsilon$, then add ε to FIRST(α)

  (3) If α is a nonterminal $X \to \overset{\varepsilon}{\overbrace{Y_1}}\,\overset{b\ldots}{Y_2}\ldots Y_k$ 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 $Y_1 Y_2 \ldots Y_k$, 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) =

FIRST(AB)

term_tail =>

+ term term_tail =>

+ id factor_tail term_tail

=> + id term_tail

=> + id

+ term term_tail =>

+ id factor_tail term_tail =>

+ id term_tail => + id

Exercise

BCD ⟹ CD ⟹ D ⟹ wS

start → *S* **$$**
S → **x** *S* | *A* **y**
A → *BCD* | ε

CD
D

B → **z** *S* | ε
C → **v** *S* | ε
D → **w** *S*

Compute FIRST sets:

FIRST(**x** *S*) = {x}

✓ ✓ ✓ ✓

FIRST(*A* **y**) = {z, v, w, y}

FIRST(*BCD*) = {z, v, w}

FIRST(**z** *S*) = {z}

FIRST(**v** *S*) = {v}

FIRST(**w** *S*) = {w}

FIRST(*S*) = {x, z, v, w, y}

FIRST(*A*) = {z, v, w, ε}

FIRST(*B*) = {z, ε}

FIRST(*C*) = {v, ε}

FIRST(*D*) = {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 $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except for **ε** should be added to FOLLOW(*B*)

  $start \Rightarrow^{*} \ldots A \ldots \Rightarrow \ldots \alpha B \beta \ldots \Rightarrow^{*} \ldots \alpha \underline{\underline{Bb}} \ldots$

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

  $start \Rightarrow^{*} \ldots Ab \ldots \Rightarrow \ldots \alpha \underline{Bb} \ldots$

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*) = { $$, + }

expr $$ => term term_tail $$ => ... + term term_tail $$

=> ... + term + term term_tail $$

Exercise

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

PREDICT Sets

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

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

$start \rightarrow S\ \mathtt{\$\$}$

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

$A \rightarrow BCD \mid \varepsilon$

$B \rightarrow \mathtt{z}\ S \mid \varepsilon$

$C \rightarrow \mathtt{v}\ S \mid \varepsilon$

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

Compute PREDICT sets:

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

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

PREDICT($A \rightarrow BCD$) = $\{z, v, w\}$

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

… *etc*…

PREDICT($A \rightarrow BCD$) $\cap$ PREDICT($A \rightarrow \varepsilon$) $= \emptyset$

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?

$$\underline{expr} \to \overbrace{\underline{expr + term}}^{term\ +\ \ldots} \mid \overbrace{\underline{term}}^{term}$$

$$term \to term\ \mathbf{*}\ \mathbf{id} \mid \mathbf{id}$$

  - Common prefixes are an obstacle. Why?

$\underline{stmt} \to$ <u>**if b then** *stmt*</u> **else** *stmt* |
<u>**if b then** *stmt*</u> |
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:

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

- After left-factoring:

stmt → **if b then** *stmt* *else_part* | **a**
else_part → **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
- Construct the LL(1) parsing table. Is this grammar an LL(1) grammar?

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

Exercise

- Compute FIRSTs:

FIRST(*stmt* **$$**) = { if, a }

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

FIRST(**a**) = {a}

FIRST(**else** *stmt*) = {else}

Exercise

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

- Compute FOLLOW:

FOLLOW(stmt) = FOLLOW(else_part)

FOLLOW(*else_part*) = { $$, else }

	else
else_part	else_part → ε else_part → else stmt

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?

No!

Exercise

if b then if b then a else a

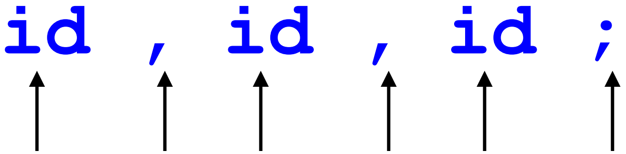
Lecture Outline

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

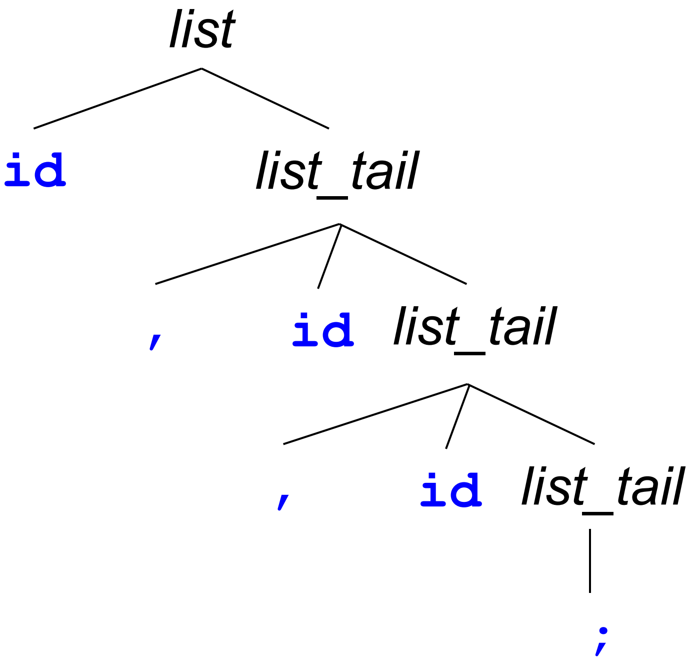
- Bottom-up parsing (also called LR parsing)
  - A brief overview, no detail

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 rightmost 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$\underline{;}$		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

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

  - 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`
`id`	`+id*id`	reduce by *term* → `id`
term	`+id*id`	reduce by *expr* → *term*
expr	`+id*id`	shift `+`
expr`+`	`id*id`	shift `id`
expr`+id`	`*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`
$expr+\underline{term*\mathtt{id}}$		reduce by $term \to term * \mathtt{id}$
$\underline{expr+term}$		reduce by $expr \to expr + term$
$expr$		ACCEPT, SUCCESS

$expr \to expr + term \mid term$

$term \to term * \mathtt{id} \mid \mathtt{id}$

`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