Programming Language Syntax

- LL parsing -

Chapter 2, Section 2.3

Parsing

- Parser
  - in charge of the entire compilation process
    - *Syntax-directed translation*
  - calls the scanner to obtain tokens
  - assembles the tokens into a syntax tree
  - passes the tree to the later phases of the compiler
    - semantic analysis
    - code generation
    - code improvement
  - a parser is a language *recognizer*
  - context-free grammar is a language *generator*

Parsing

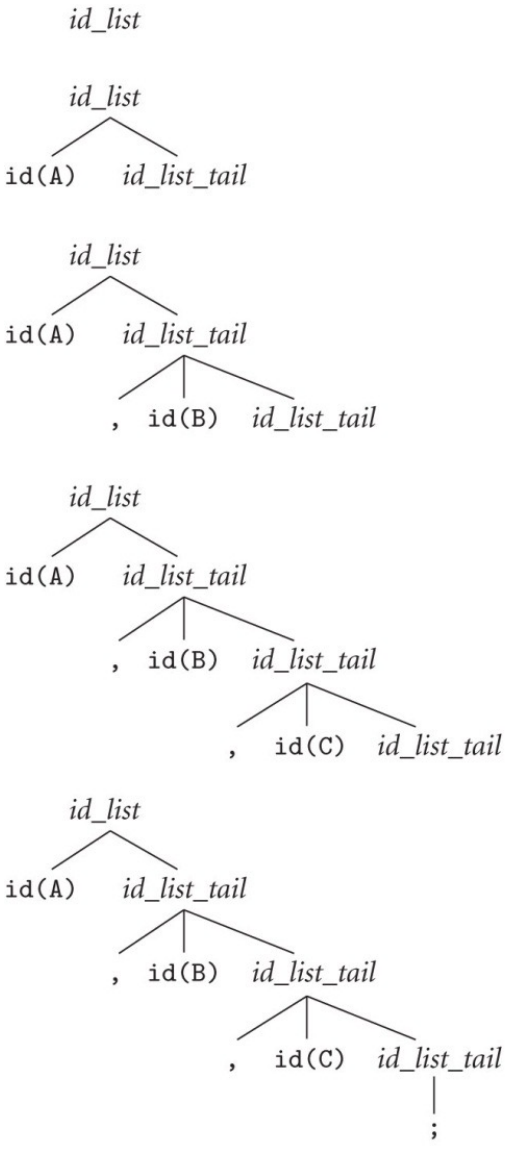
- Context-free language recognition
  - Earley, Cocke-Younger-Kasami alg's
  - $O(n^3)$ time
    - too slow
  - There are classes of grammars with $O(n)$ parsers:
    - **LL**: '**L**eft-to-right, **L**eftmost derivation'.
    - **LR**: '**L**eft-to-right, **R**ightmost derivation′

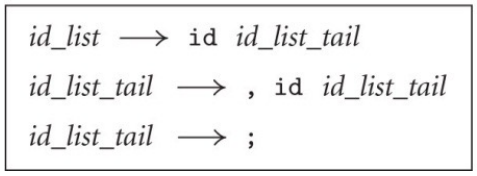
Class	Direction of scanning	Derivation discovered	Parse tree construction	Algorithm used
LL	left-to-right	left-most	top-down	predictive
LR	left-to-right	right-most	bottom-up	shift-reduce

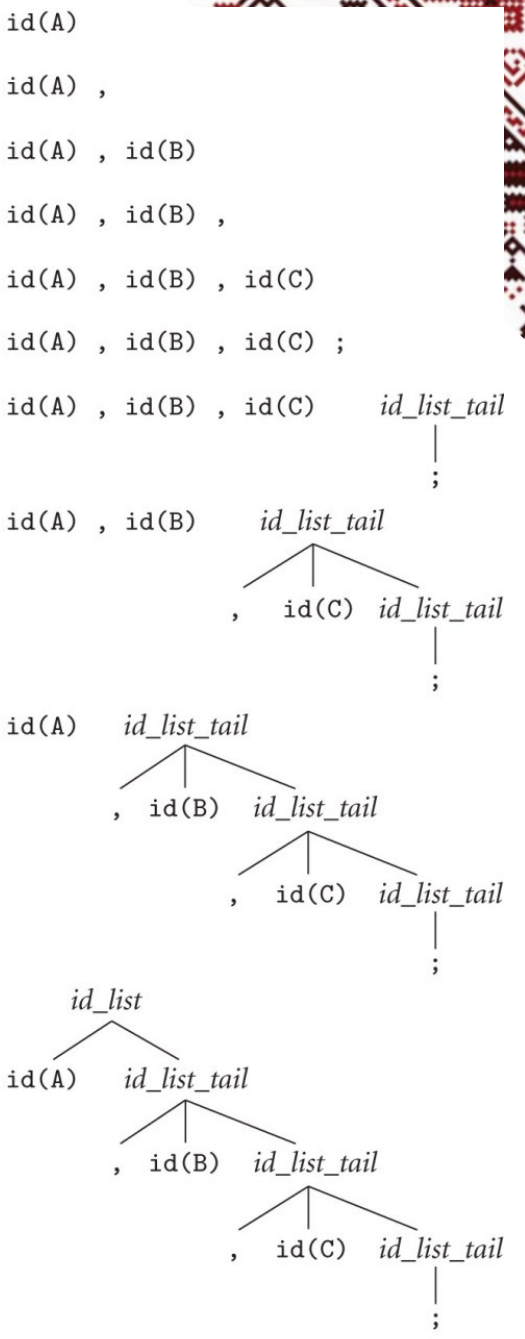
Parsing

- Top-down vs. Bottom-up
- Top-down
  - *predict* based on next token
- Bottom-up
  - *reduce* right-hand side
  - Example:

```
A, B, C;
```



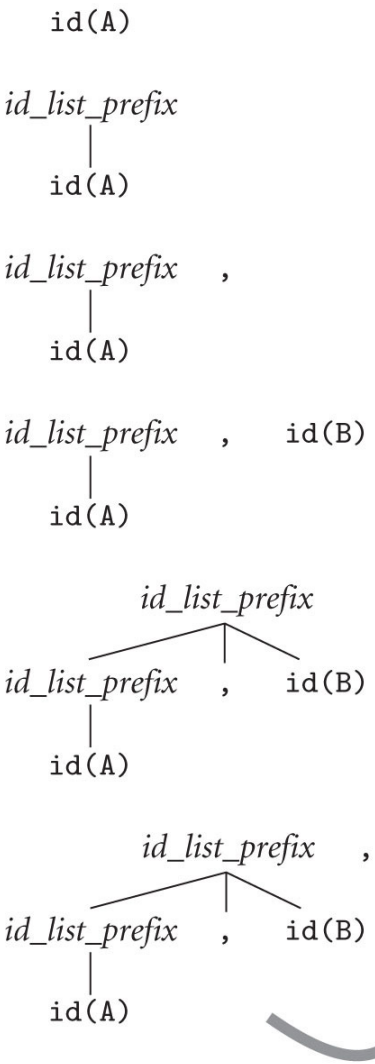


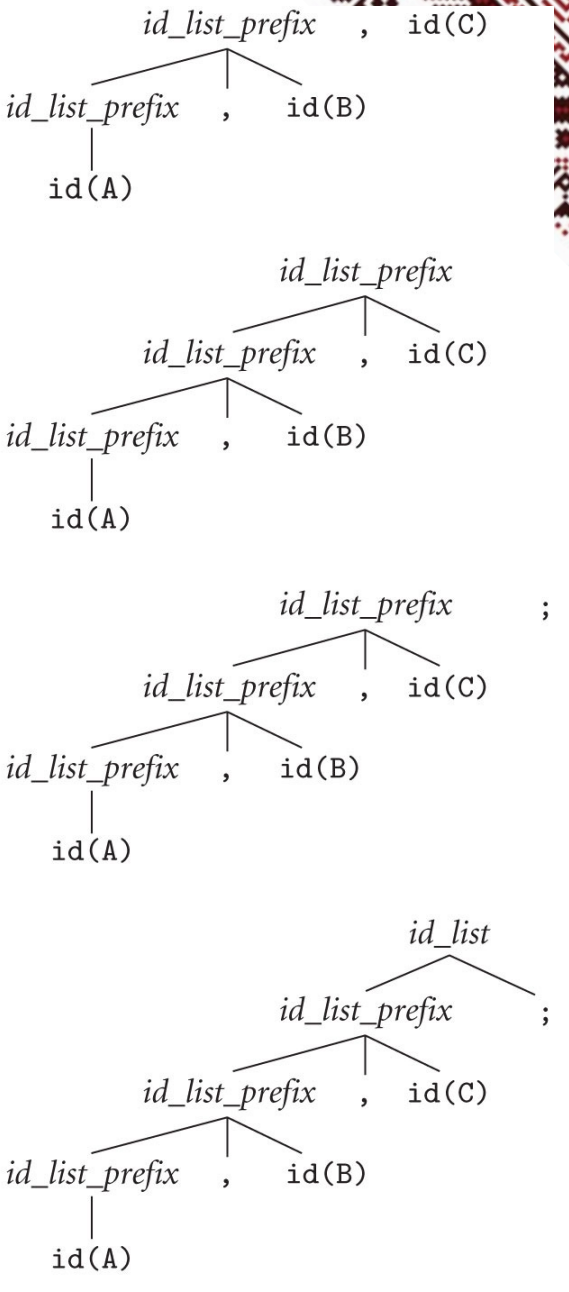


Parsing

- Bottom-up
  - better grammar
  - cannot be parsed top-down
  - Example:

    `A, B, C;`





$$\begin{aligned} \textit{id\_list} &\longrightarrow \textit{id\_list\_prefix}\ ; \\ \textit{id\_list\_prefix} &\longrightarrow \textit{id\_list\_prefix}\ ,\ \texttt{id} \\ &\longrightarrow \texttt{id} \end{aligned}$$

Parsing

- LL(k), LR(k)
  - k = no. tokens of look-ahead required to parse
  - almost all real compilers use LL(1), LR(1)
  - LR(0) - *prefix property*:
    - no valid string is a prefix of another valid string

LL Parsing

- LL(1) grammar for calculator language
  - less intuitive: operands not on the same right-hand side
  - parsing is easier ($$ added to mark the end of the program)

program → *stmt_list* `$$`
stmt_list → *stmt* *stmt_list* | ε
stmt → `id :=` *expr* | `read id` | `write` *expr*
expr → *term* *term_tail*
term_tail → *add_op* *term* *term_tail* | ε
term → *factor* *fact_tail*
fact_tail → *mult_op* *fact* *fact_tail* | ε
factor → `(` *expr* `)` | `id` | `number`
add_op → `+` | `-`
mult_op → `*` | `/`

- compare with LR grammar:

expr → *term* | *expr add_op term*
term → *factor* | *term mult_op factor*
factor → `id` | `number` | `-` *factor* | (*expr*)
add_op → `+` | `-`
mult_op → `*` | `/`

- Top-down parsers
  - by hand – *recursive descent*
  - table-driven

LL Parsing

- Recursive descent parser
  - one subroutine for each nonterminal
- Example:

```
read A
read B
sum := A + B
write sum
write sum / 2
```

  - Continued on the next slide

```
procedure match(expected)
    if input_token = expected then consume_input_token()
    else parse_error

-- this is the start routine:
procedure program()
    case input_token of
        id, read, write, $$ :
            stmt_list()
            match($$)
        otherwise parse_error

procedure stmt_list()
    case input_token of
        id, read, write : stmt(); stmt_list()
        $$ : skip        -- epsilon production
        otherwise parse_error
```

LL Parsing

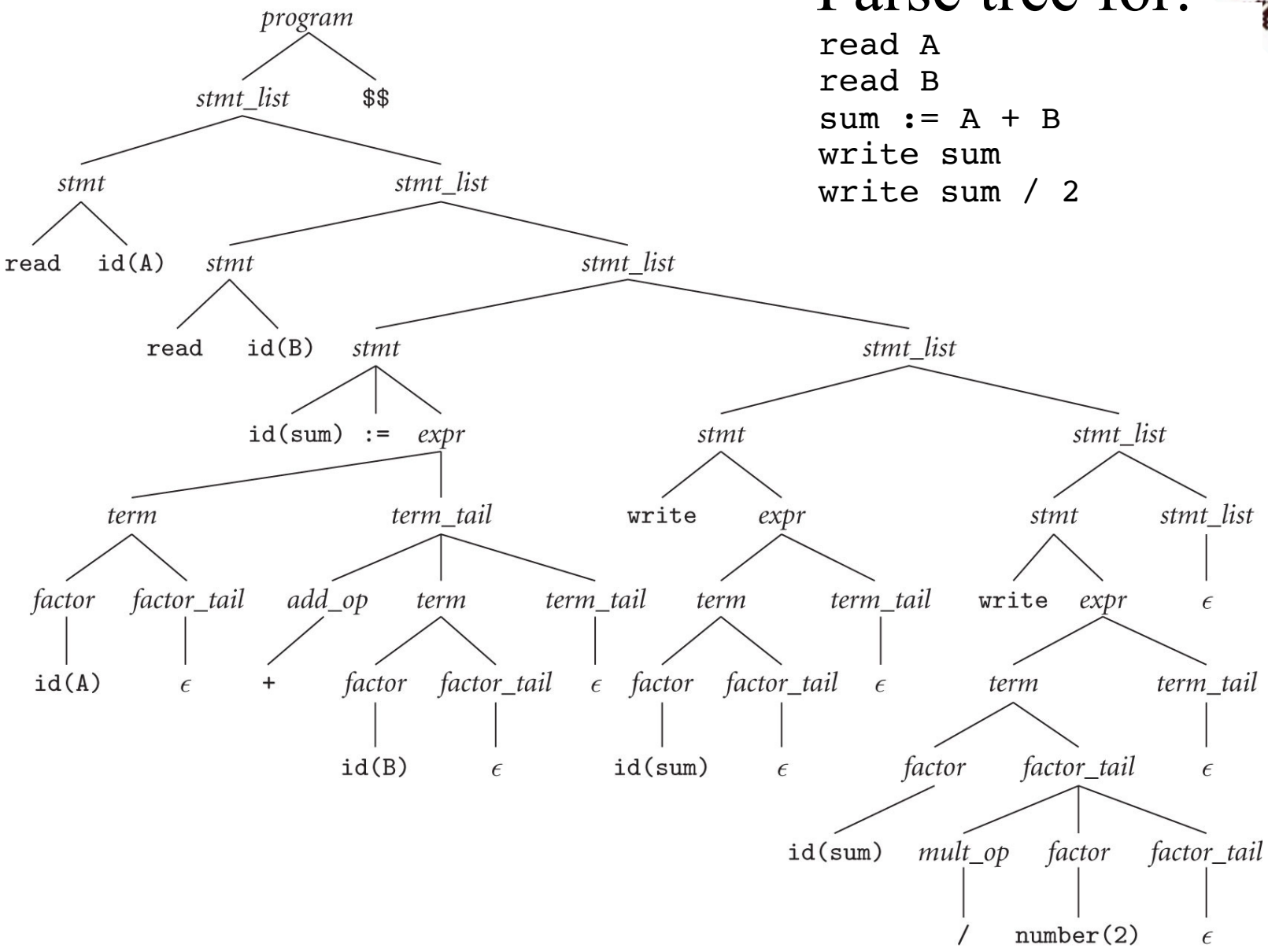
```
procedure stmt()
    case input_token of
        id : match(id); match(:=); expr()
        read : match(read); match(id)
        write : match(write); expr()
        otherwise parse_error

procedure expr()
    case input_token of
        id, number, ( : term(); term_tail()
        otherwise parse_error

procedure term_tail()
    case input_token of
        +, - : add_op(); term(); term_tail()
        ), id, read, write, $$ :
            skip        -- epsilon production
        otherwise parse_error

procedure term()
    case input_token of
        id, number, ( : factor(); factor_tail()
        otherwise parse_error
```

```
procedure factor_tail()
    case input_token of
        *, / : mult_op(); factor(); factor_tail()
        +, -, ), id, read, write, $$ :
            skip        -- epsilon production
        otherwise parse_error

procedure factor()
    case input_token of
        id : match(id)
        number : match(number)
        ( : match(); expr(); match())
        otherwise parse_error

procedure add_op()
    case input_token of
        + : match(+)
        - : match(-)
        otherwise parse_error

procedure mult_op()
    case input_token of
        * : match(*)
        / : match(/)
        otherwise parse_error
```

LL Parsing

- Parse tree for:

```
read A
read B
sum := A + B
write sum
write sum / 2
```

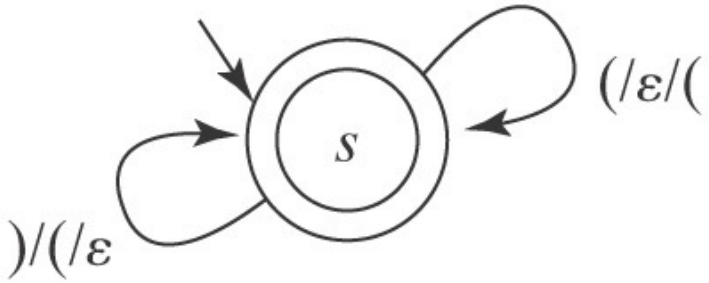


LL Parsing

- Table-driven LL parsing:
  - repeatedly look up action in 2D table based on:
    - current leftmost non-terminal and
    - current input token
  - actions:
    - (1) match a terminal
    - (2) predict a production
    - (3) announce a syntax error

LL Parsing

- Table-driven LL parsing:
  - Push-down automaton (PDA)
    - Finite automaton with a stack
    - Example: balanced parentheses: input / pop / push



  - Parsing stack: containing the expected symbols
    - initially contains the starting symbol
    - predicting a production: push the right-hand side in reverse order

因为是stack 而LL是从左边往右边 所以右边的要在stack底部 push的时候就要reverse order来push了

LL Parsing

- Table-driven LL parsing:

```
terminal = 1 . . number_of_terminals
non_terminal = number_of_terminals + 1 . . number_of_symbols
symbol = 1 . . number_of_symbols
production = 1 . . number_of_productions

parse_tab : array [non_terminal, terminal] of record
        action : (predict, error)
        prod : production
prod_tab : array [production] of list of symbol
-- these two tables are created by a parser generator tool

parse_stack : stack of symbol

parse_stack.push(start_symbol)
loop
        expected_sym : symbol := parse_stack.pop()
        if expected_sym ∈ terminal
                match(expected_sym)                          -- as in Figure 2.17
                if expected_sym = $$ then return             -- success!
        else
                if parse_tab[expected_sym, input_token].action = error
                        parse_error
                else
                        prediction : production := parse_tab[expected_sym, input_token].prod
                        foreach sym : symbol in reverse prod_tab[prediction]
                                parse_stack.push(sym)
```

LL Parsing

We push the whole to into stack, so the top of stack is the left most non-terminal

- LL(1): `parse_tab` for parsing for calculator language
- productions: 1..19
- '–' means error
- `prod_tab` (not shown) gives RHS

1	*program* → *stmt_list* $$
2,3	*stmt_list* → *stmt stmt_list* | ε
4,5,6	*stmt* → id := *expr* | read id | write *expr*
7	*expr* → *term term_tail*
8,9	*term_tail* → *add_op term term_tail* | ε
10	*term* → *factor fact_tail*
11,12	*fact_tail* → *mult_op fact fact_tail* | ε
13,14,15	*factor* → (*expr*) | id | number
16,17	*add_op* → + | –
18,19	*mult_op* → * | /

横是terminal 看token和nonterminal能有啥组合

这是left-hand side

Top-of-stack nonterminal	Current input token											
	id	number	read	write	:=	(	)	+	-	*	/	$$
program	1	–	1	1	–	–	–	–	–	–	–	1
stmt_list	2	–	2	2	–	–	–	–	–	–	–	3
stmt	4	–	5	6	–	–	–	–	–	–	–	–
expr	7	7	–	–	–	7	–	–	–	–	–	–
term_tail	9	–	9	9	–	–	9	8	8	–	–	9
term	10	10	–	–	–	10	–	–	–	–	–	–
factor_tail	12	–	12	12	–	–	12	12	12	11	11	12
factor	14	15	–	–	–	13	–	–	–	–	–	–
add_op	–	–	–	–	–	–	–	16	17	–	–	–
mult_op	–	–	–	–	–	–	–	–	–	18	19	–

LL Parsing

- Example:

```
read A
read B
sum := A + B
write sum
write sum / 2
```

两边读第一个token

这里是实际的input

这边是stack有的token

直到找到能互相消除的terminal然后就互相消除

Parse stack	Input stream	Comment
program	`read A read B ...`	initial stack contents
stmt_list `$$`	`read A read B ...`	predict *program* ⟶ *stmt_list* `$$`
stmt stmt_list `$$`	`read A read B ...`	predict *stmt_list* ⟶ *stmt stmt_list*
`read id` *stmt_list* `$$`	`read A read B ...`	predict *stmt* ⟶ `read id`
`id` *stmt_list* `$$`	`A read B ...`	match `read`
stmt_list `$$`	`read B sum := ...`	match `id`
stmt stmt_list `$$`	`read B sum := ...`	predict *stmt_list* ⟶ *stmt stmt_list*
`read id` *stmt_list* `$$`	`read B sum := ...`	predict *stmt* ⟶ `read id`
`id` *stmt_list* `$$`	`B sum := ...`	match `read`
stmt_list `$$`	`sum := A + B ...`	match `id`
stmt stmt_list `$$`	`sum := A + B ...`	predict *stmt_list* ⟶ *stmt stmt_list*
`id :=` *expr stmt_list* `$$`	`sum := A + B ...`	predict *stmt* ⟶ `id :=` *expr*
`:=` *expr stmt_list* `$$`	`:= A + B ...`	match `id`
expr stmt_list `$$`	`A + B ...`	match `:=`
term term_tail stmt_list `$$`	`A + B ...`	predict *expr* ⟶ *term term_tail*
factor factor_tail term_tail stmt_list `$$`	`A + B ...`	predict *term* ⟶ *factor factor_tail*
`id` *factor_tail term_tail stmt_list* `$$`	`A + B ...`	predict *factor* ⟶ `id`
factor_tail term_tail stmt_list `$$`	`+ B write sum ...`	match `id`
term_tail stmt_list `$$`	`+ B write sum ...`	predict *factor_tail* ⟶ ϵ
add_op term term_tail stmt_list `$$`	`+ B write sum ...`	predict *term_tail* ⟶ *add_op term term_tail*
`+` *term term_tail stmt_list* `$$`	`+ B write sum ...`	predict *add_op* ⟶ `+`
term term_tail stmt_list `$$`	`B write sum ...`	match `+`
factor factor_tail term_tail stmt_list `$$`	`B write sum ...`	predict *term* ⟶ *factor factor_tail*
`id` *factor_tail term_tail stmt_list* `$$`	`B write sum ...`	predict *factor* ⟶ `id`
factor_tail term_tail stmt_list `$$`	`write sum ...`	match `id`

LL Parsing

- Example:

```
read A
read B
sum := A + B
write sum
write sum / 2
```

Parse stack	Input stream	Comment
term_tail stmt_list $$	write sum write ...	predict *factor_tail* $\longrightarrow \epsilon$
stmt_list $$	write sum write ...	predict *term_tail* $\longrightarrow \epsilon$
stmt stmt_list $$	write sum write ...	predict *stmt_list* $\longrightarrow$ *stmt stmt_list*
write *expr stmt_list* $$	write sum write ...	predict *stmt* $\longrightarrow$ write *expr*
expr stmt_list $$	sum write sum / 2	match write
term term_tail stmt_list $$	sum write sum / 2	predict *expr* $\longrightarrow$ *term term_tail*
factor factor_tail term_tail stmt_list $$	sum write sum / 2	predict *term* $\longrightarrow$ *factor factor_tail*
id *factor_tail term_tail stmt_list* $$	sum write sum / 2	predict *factor* $\longrightarrow$ id
factor_tail term_tail stmt_list $$	write sum / 2	match id
term_tail stmt_list $$	write sum / 2	predict *factor_tail* $\longrightarrow \epsilon$
stmt_list $$	write sum / 2	predict *term_tail* $\longrightarrow \epsilon$
stmt stmt_list $$	write sum / 2	predict *stmt_list* $\longrightarrow$ *stmt stmt_list*
write *expr stmt_list* $$	write sum / 2	predict *stmt* $\longrightarrow$ write *expr*
expr stmt_list $$	sum / 2	match write
term term_tail stmt_list $$	sum / 2	predict *expr* $\longrightarrow$ *term term_tail*
factor factor_tail term_tail stmt_list $$	sum / 2	predict *term* $\longrightarrow$ *factor factor_tail*
id *factor_tail term_tail stmt_list* $$	sum / 2	predict *factor* $\longrightarrow$ id
factor_tail term_tail stmt_list $$	/ 2	match id
mult_op factor factor_tail term_tail stmt_list $$	/ 2	predict *factor_tail* $\longrightarrow$ *mult_op factor factor_tail*
/ *factor factor_tail term_tail stmt_list* $$	/ 2	predict *mult_op* $\longrightarrow$ /
factor factor_tail term_tail stmt_list $$	2	match /
number *factor_tail term_tail stmt_list* $$	2	predict *factor* $\longrightarrow$ number
factor_tail term_tail stmt_list $$		match number
term_tail stmt_list $$		predict *factor_tail* $\longrightarrow \epsilon$
stmt_list $$		predict *term_tail* $\longrightarrow \epsilon$
$$		predict *stmt_list* $\longrightarrow \epsilon$

LL Parsing

- How to build the table:
  - FIRST(α) – tokens that can start an α ← This is string
  - FOLLOW(A) – tokens that can come after an A ← Non-terminal

We define follow for non-terminal ONLY

can α produce null or not ←

$$\texttt{EPS}(\alpha) \equiv \text{if } \alpha \Longrightarrow^* \varepsilon \text{ then } \texttt{true} \text{ else } \texttt{false}$$

$$\texttt{FIRST}(\alpha) \equiv \{c \mid \alpha \Longrightarrow^* c\beta\}$$ ← 一个non-terminal里面的右边第一个token

$$\texttt{FOLLOW}(A) \equiv \{c \mid S \Longrightarrow^+ \alpha A c \beta\}$$ ← 一个non-terminal后面的第一个terminal

$$\texttt{PREDICT}(A \rightarrow \alpha) \equiv \texttt{FIRST}(\alpha) \cup \text{if } \texttt{EPS}(\alpha) \text{ then } \texttt{FOLLOW}(A) \text{ else } \emptyset$$

  - If a token belongs to the predict set of more than one production with the same left-hand side, then the grammar is not LL(1)
  - Compute: pass over the grammar until nothing changes
  - Algorithm and examples on the next slides

LL Parsing

- Constructing `EPS, FIRST, FOLLOW, PREDICT`

program $\longrightarrow$ *stmt_list* `$$`	`$$` $\in$ FOLLOW(*stmt_list*)
stmt_list $\longrightarrow$ *stmt* *stmt_list*	
stmt_list $\longrightarrow \epsilon$	EPS(*stmt_list*) = true
stmt $\longrightarrow$ `id :=` *expr*	`id` $\in$ FIRST(*stmt*)
stmt $\longrightarrow$ `read id`	`read` $\in$ FIRST(*stmt*)
stmt $\longrightarrow$ `write` *expr*	`write` $\in$ FIRST(*stmt*)
expr $\longrightarrow$ *term* *term_tail*	
term_tail $\longrightarrow$ *add_op* *term* *term_tail*	
term_tail $\longrightarrow \epsilon$	EPS(*term_tail*) = true
term $\longrightarrow$ *factor* *factor_tail*	
factor_tail $\longrightarrow$ *mult_op* *factor* *factor_tail*	
factor_tail $\longrightarrow \epsilon$	EPS(*factor_tail*) = true
factor $\longrightarrow$ `(` *expr* `)`	`(` $\in$ FIRST(*factor*) and `)` $\in$ FOLLOW(*expr*)
factor $\longrightarrow$ `id`	`id` $\in$ FIRST(*factor*)
factor $\longrightarrow$ `number`	`number` $\in$ FIRST(*factor*)
add_op $\longrightarrow$ `+`	`+` $\in$ FIRST(*add_op*)
add_op $\longrightarrow$ `-`	`-` $\in$ FIRST(*add_op*)
mult_op $\longrightarrow$ `*`	`*` $\in$ FIRST(*mult_op*)
mult_op $\longrightarrow$ `/`	`/` $\in$ FIRST(*mult_op*)

LL Parsing

- Algorithm for constructing `EPS, FIRST, FOLLOW, PREDICT`

(Continued on the next slide)

所有的terminal c EPS都是false — 因为它不可能被evaluate成null

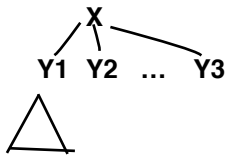
而terminal的first也就是他自己

这是initialization

```
-- EPS values and FIRST sets for all symbols:
    for all terminals c, EPS(c) := false; FIRST(c) := {c}
    for all nonterminals X, EPS(X) := if X ⟶ ϵ then true else false; FIRST(X) := ∅
    repeat
        ⟨outer⟩ for all productions X ⟶ Y1 Y2 ... Yk,
            ⟨inner⟩ for i in 1..k
                add FIRST(Yi) to FIRST(X)
                if not EPS(Yi) (yet) then continue outer loop
            EPS(X) := true
    until no further progress
-- Subroutines for strings, similar to inner loop above:

    function string_EPS(X1 X2 ... Xn)
        for i in 1..n
            if not EPS(Xi) then return false
        return true
```

Y是指左边的一个production可能性

如果Y没有EPS呢?



只要Y有EPS,
下一个Y的first
就可以加入X的first

LL Parsing

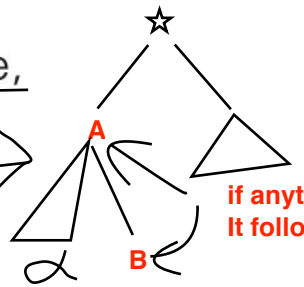
- Algorithm for constructing EPS, FIRST, FOLLOW, PREDICT

```
function string_FIRST(X1 X2 ... Xn)
    return_value := ∅
    for i in 1..n
        add FIRST(Xi) to return_value
        if not EPS(Xi) then return
```

如果不能产生NULL 就直接结束

```
-- FOLLOW sets for all symbols:
    for all symbols X, FOLLOW(X) := ∅
    repeat
        for all productions A ⟶ α B β,
            add string_FIRST(β) to FOLLOW(B)
        for all productions A ⟶ α B
                or A ⟶ α B β, where string_EPS(β) = true,
            add FOLLOW(A) to FOLLOW(B)
    until no further progress
```

如果beta可以消失




```
-- PREDICT sets for all productions:
    for all productions A ⟶ α
        PREDICT(A ⟶ α) := string_FIRST(α) ∪ (if string_EPS(α) then FOLLOW(A) else ∅)
```

LL Parsing

EPS(A) is true iff
A ∈ {*stmt_list*, *term_tail*, *factor_tail*}

- Example: the sets `EPS, FIRST, FOLLOW, PREDICT`

FIRST

program {`id, read, write, $$`}
stmt_list {`id, read, write`}
stmt {`id, read, write`}
expr {`(, id, number`}
term_tail {`+, -`}
term {`(, id, number`}
factor_tail {`*, /`}
factor {`(, id, number`}
add_op {`+, -`}
mult_op {`*, /`}

FOLLOW

program ∅
stmt_list {`$$`}
stmt {`id, read, write, $$`}
expr {`), id, read, write, $$`}
term_tail {`), id, read, write, $$`}
term {`+, -, ), id, read, write, $$`}
factor_tail {`+, -, ), id, read, write, $$`}
factor {`+, -, *, /, ), id, read, write, $$`}
add_op {`(, id, number`}
mult_op {`(, id, number`}

PREDICT

1. *program* ⟶ *stmt_list* `$$` {`id, read, write, $$`}
2. *stmt_list* ⟶ *stmt stmt_list* {`id, read, write`}
3. *stmt_list* ⟶ ϵ {`$$`}
4. *stmt* ⟶ `id :=` *expr* {`id`}
5. *stmt* ⟶ `read id` {`read`}
6. *stmt* ⟶ `write` *expr* {`write`}
7. *expr* ⟶ *term term_tail* {`(, id, number`}
8. *term_tail* ⟶ *add_op term term_tail* {`+, -`}
9. *term_tail* ⟶ ϵ {`), id, read, write, $$`}
10. *term* ⟶ *factor factor_tail* {`(, id, number`}
11. *factor_tail* ⟶ *mult_op factor factor_tail* {`*, /`}
12. *factor_tail* ⟶ ϵ {`+, -, ), id, read, write, $$`}
13. *factor* ⟶ `(` *expr* `)` {`(`}
14. *factor* ⟶ `id` {`id`}
15. *factor* ⟶ `number` {`number`}
16. *add_op* ⟶ `+` {`+`}
17. *add_op* ⟶ `-` {`-`}
18. *mult_op* ⟶ `*` {`*`}
19. *mult_op* ⟶ `/` {`/`}

LL Parsing

- Problems trying to make a grammar LL(1)
  - *left recursion*: $A \Longrightarrow^{+} A\alpha$ (Bad)
    - example – cannot be parsed top-down

$$id\_list \rightarrow id\_list\_prefix\ \texttt{;}$$
$$id\_list\_prefix \rightarrow id\_list\_prefix\ \texttt{,}\ \texttt{id}$$
$$id\_list\_prefix \rightarrow \texttt{id}$$

    - solved by *left-recursion elimination*

$$id\_list \rightarrow \texttt{id}\ id\_list\_tail$$
$$id\_list\_tail \rightarrow \texttt{,}\ \texttt{id}\ id\_list\_tail$$
$$id\_list\_tail \rightarrow \texttt{;}$$

  - General left-recursion elimination:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$

replaced by:

$$A \rightarrow \beta_1 B \mid \beta_2 B \mid \ldots \mid \beta_m B$$
$$B \rightarrow \alpha_1 B \mid \alpha_2 B \mid \ldots \mid \alpha_n B \mid \varepsilon$$

id_list_prefix –> id B
B –> ,id B | ε

尽可能把recursion往右边搞?

LL Parsing

- Problems trying to make a grammar LL(1)
  - *common prefixes*
    - example

same first token
so we don't know
which one get used

stmt → `id` `:=` *expr*
stmt → `id` `(` *argument_list* `)`

    - solved by *left-factoring*

stmt → `id` *stmt_list_tail*
stmt_list_tail → `:=` *expr*
stmt_list_tail → `(` *argument_list* `)`

- Note: Eliminating left recursion and common prefixes does NOT make a grammar LL; there are infinitely many non-LL languages, and the automatic transformations work on them just fine

LL Parsing

- Problems trying to make a grammar LL(1)
  - the *dangling else* problem
  - prevents grammars from being LL(*k*) for any *k*
  - Example: ambiguous (Pascal)

stmt → `if` *cond then_clause else_clause* | *other_stmt*
then_clause → `then` *stmt*
else_clause → `else` *stmt* | ε

`if` C_1 `then if` C_2 `then` S_1 `else` S_2

LL Parsing

- Dangling else problem
  - Solution: unambiguous grammar
  - can be parsed bottom-up but not top-down
    - there is no top-down grammar

You don't know how many token to look ahead, there could be tons of then and else

stmt → *balanced_stmt* | *unbalanced_stmt*

balanced_stmt → `if` *cond* `then` *balanced_stmt* `else` *balanced_stmt*
| *other_stmt*

unbalanced_stmt → `if` *cond* `then` *stmt*
| `if` *cond* `then` *balanced_stmt* `else` *unbalanced_stmt*

LL Parsing

- Dangling else problem
  - Another solution - *end-markers*

stmt → `IF` *cond then_clause else_clause* `END` | *other_stmt*
then_clause → `THEN` *stmt_list*
else_clause → `ELSE` *stmt_list* | ε

- Modula-2, for example, one says:

```
if A = B then
      if C = D then E := F end
else
      G := H
end
```

- Ada: `end if`
- other languages: `fi`

LL Parsing

- Problem with end markers: they tend to bunch up

```
if A = B then …
else if A = C then …
else if A = D then …
else if A = E then …
else ...
end end end end
```

  - To avoid this: `elsif`

```
if A = B then …
elsif A = C then …
elsif A = D then …
elsif A = E then …
else ...
end
```