CS307: Principles of Programming Languages

LECTURE 9: PARSING

LECTURE OUTLINE

- THEORETICAL BACKGROUND
  - REGULAR EXPRESSIONS
  - CHOMSKY HIERARCHY
  - CONTEXT FREE GRAMMARS
- SCANNING
- **PARSING**
  - **TOP-DOWN**
  - **BOTTOM-UP**
  - **LL AND LR PARSING**
- CLASSIC PARSING TOOLS
  - BISON
  - LEX
  - PLY
  - TPG

PARSING

- PARSER – TYPICAL FLOW
  - CALL SCANNER (LEXICAL ANALYZER) TO GET TOKENS
  - ASSEMBLE TOKENS INTO A SYNTAX TREE
  - PASS THE TREE TO LATER PHASES OF THE COMPILER (*SYNTAX-DIRECTED TRANSLATION*)
- MOST PARSERS HANDLE A CONTEXT-FREE GRAMMAR (CFG)

PARSING

- TERMINOLOGY:
  - SYMBOLS
    - **TERMINALS** – SYMBOLS THAT HAVE NO ADDITIONAL BREAKDOWN
    - **NON-TERMINALS** – SYMBOLS THAT REPRESENT A SEQUENCE OF OTHER SYMBOLS
  - **PRODUCTIONS** – A POSSIBLE EXPANSION OF A NON-TERMINAL INTO A SEQUENCE OF TERMINALS AND NON-TERMINALS
  - **DERIVATION** – A SEQUENCE OF PRODUCTION APPLICATIONS DERIVING A PARSE OR SYNTAX TREE
    - (LEFT-MOST AND RIGHT-MOST – CANONICAL)
  - **PARSE TREES** – A TREE DATA STRUCTURE REPRESENTING THE SYNTAX OF A PROGRAM
  - **SENTENTIAL FORMS** – INTERMEDIATE EXPANSIONS (APPLICATIONS OF PRODUCTIONS) OF A PROGRAM'S CODE

PARSING

- FOR ANY CFG, CAN CREATE A PARSER THAT RUNS IN $O(N^3)$ TIME
  - EARLEY'S ALGORITHM
  - COCKE-YOUNGER-KASAMI(CYK) ALGORITHM
- $O(N^3)$ TIME NOT ACCEPTABLE FOR A PARSER
  - BECOMES PROHIBITIVE WITH LONGER SOURCE CODE

PARSING

- THERE ARE LARGE CLASSES OF GRAMMARS FOR WHICH PARSERS CAN RUN IN LINEAR TIME
- THE TWO MOST IMPORTANT: LL AND LR
  - LL: 'LEFT-TO-RIGHT, LEFTMOST DERIVATION'
  - LR: 'LEFT-TO-RIGHT, RIGHTMOST DERIVATION'
- LEFTMOST DERIVATION
  - WORK ON THE LEFT SIDE OF THE PARSE TREE
  - EXPAND LEFT-MOST NON-TERMINAL IN A SENTENTIAL FORM
- RIGHTMOST DERIVATION
  - WORK ON THE RIGHT SIDE OF THE TREE
  - EXPAND RIGHT-MOST NON-TERMINAL IN A SENTENTIAL FORM

PARSING

- LL PARSERS
  - TOP-DOWN – APPLY PRODUCTIONS STARTING AT START SYMBOL
  - PREDICTIVE – MUST PREDICT WHICH PRODUCTION TO USE
- LR PARSERS
  - BOTTOM-UP – GATHER TOKENS AND 'COMBINE' THEM BY APPLYING PRODUCTION IN REVERSE
  - SHIFT-REDUCE
    - SHIFT TOKENS ONTO A STACK
    - REDUCE GROUPS OF TERMINALS AND NON-TERMINALS TO A SINGLE NON-TERMINAL ACCORDING TO PRODUCTIONS
- SEVERAL IMPORTANT SUB-CLASSES OF LR PARSERS
  - SLR
  - LALR
  - "FULL LR"

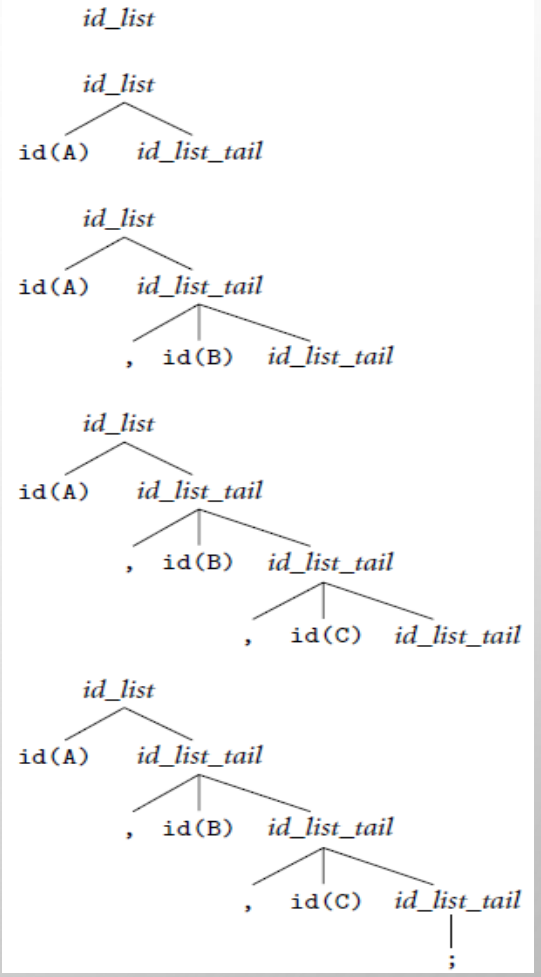
TOP-DOWN PARSING (LL)

- CONSIDER A GRAMMAR FOR
  - COMMA SEPARATED LIST OF IDENTIFIERS
  - TERMINATED BY A SEMICOLON

  id_list→ id *id_list_tail*
  id_list_tail→ , id *id_list_tail*

  id_list_tail→ ;

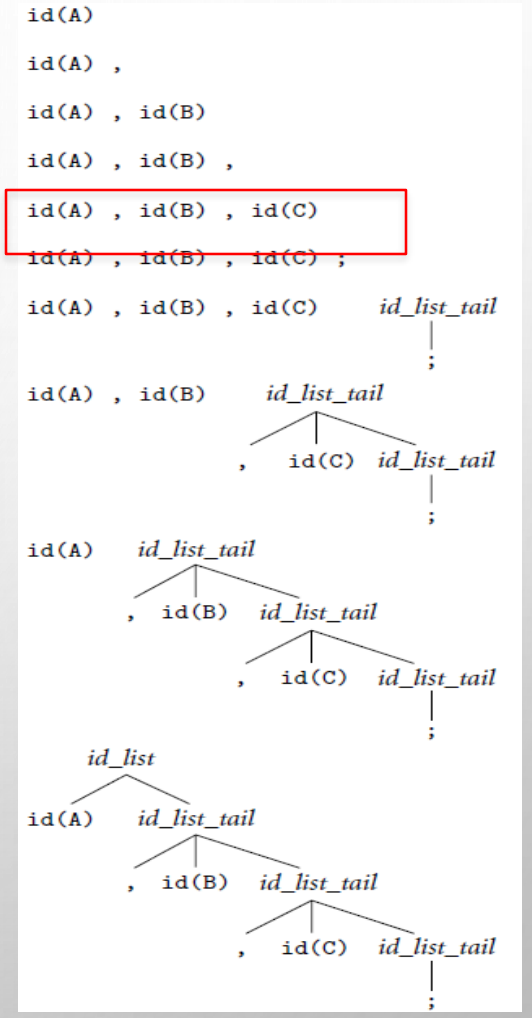
- TOP-DOWN CONSTRUCTION OF A PARSE TREE FOR THE STRING: "A, B, C;"STARTS FROM THE ROOT AND APPLIES RULES



BOTTOM-UP PARSING (LR)

id_list⟶ id *id_list_tail*
id_list_tail⟶ , id *id_list_tail*
id_list_tail⟶ ;

- BOTTOM-UP CONSTRUCTION OF A PARSE TREE FOR: “A, B, C;”
- THE PARSER FINDS:
  - LEFT-MOST LEAF OF THE TREE IS AN ID
  - NEXT LEAF IS A COMMA
  - PARSER CONTINUES SHIFTING NEW LEAVES INTO A FOREST OF PARTIALLY COMPLETED PARSE TREE FRAGMENTS



BOTTOM-UP PARSING (LR)

THE BOTTOM-UP CONSTRUCTION REALIZES

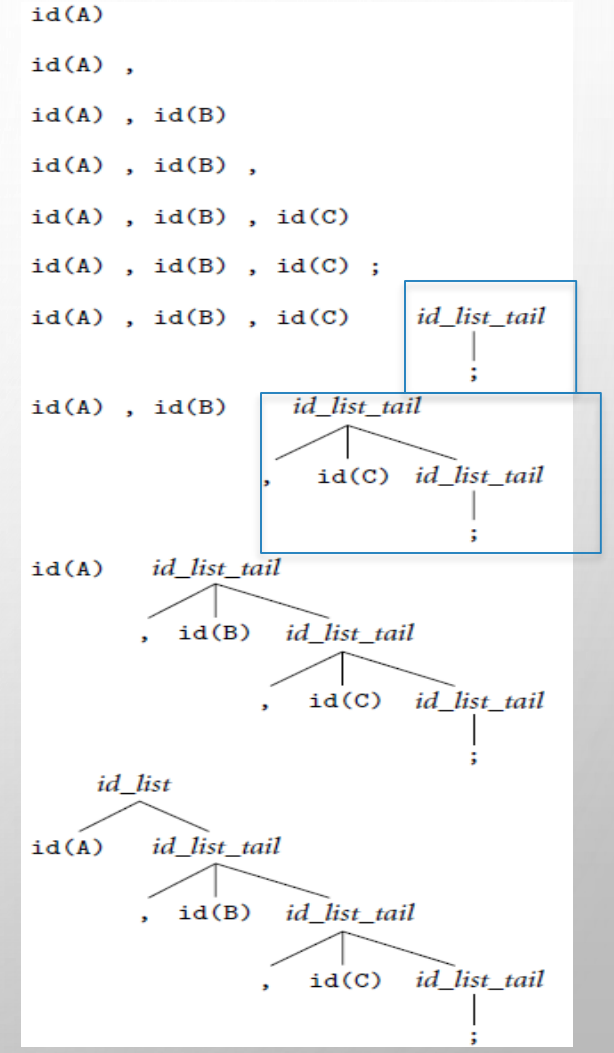
SOME FRAGMENTS CONSTITUTE A COMPLETE RIGHT-HAND SIDE.

- I.E. OCCURS WHEN THE PARSER HAS SEEN THE SEMICOLON—THE RIGHT-HAND SIDE OF *ID_LIST_TAIL*

- THE PARSER REDUCES THE SEMICOLON TO AN *ID_LIST_TAIL*

- THEN REDUCES ", ID *IDLIST TAIL*" INTO ANOTHER *ID LIST TAIL*.

- AFTER ONE MORE TIME IT CAN REDUCE "ID *IDLIST TAIL*" INTO PARSE TREE ROOT: *ID_LIST*.



PARSING

- NUMBER IN LL(1), LL(2),…
  - HOW MANY TOKENS OF LOOK-AHEAD ARE REQUIRED
  - ALMOST ALL REAL COMPILERS USE **ONE TOKEN** OF LOOK-AHEAD
- EARLIER EXPRESSION GRAMMAR (WITH PRECEDENCE AND ASSOCIATIVITY) IS LR(1), BUT NOT LL(1)
- EVERY LL(1) GRAMMAR IS ALSO LR(1)
  - CAVEAT: RIGHT RECURSION IN PRODUCTION REQUIRES VERY DEEP STACKS AND COMPLICATES SEMANTIC ANALYSIS

EXAMPLE: AN LL(1) GRAMMAR

program → stmt_list $$(end of file)

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 factor fact_tail

| ε

factor → (expr)

| id

| number

add_op → +

| -

mult_op → *

| /

LL PARSING

- ADVANTAGES OF THIS GRAMMAR
  - CAPTURES ASSOCIATIVITY AND PRECEDENCE
  - SIMPLICITY OF PARSING ALGORITHM
- DISADVANTAGES OF THE GRAMMAR
  - NOT AS 'PRETTY' AS OTHER EQUIVALENT GRAMMARS
  - OPERANDS OF AN OPERATOR NOT ON SAME RIGHT-HAND SIDE (RHS)

LL PARSING

- EXAMPLE (THE AVERAGE PROGRAM):

```
READ A
READ B
SUM := A + B
WRITE SUM
WRITE SUM / 2      $$
```

- PARSING PROCESS
  - START AT TOP
  - PREDICT NEXT NEEDED PRODUCTION BASED ON:
    - CURRENT LEFT-MOST NON-TERMINAL
    - CURRENT INPUT TOKEN

LL PARSING

- TABLE-DRIVEN PARSING:
  - LOOP: LOOK UP NEXT ACTION IN 2D TABLE BASED ON:
    - CURRENT LEFT-MOST NON-TERMINAL
    - CURRENT INPUT TOKEN
  - POSSIBLE ACTIONS:
    - MATCH A TERMINAL => PREDICT A PRODUCTION
    - SYNTAX ERROR

LL PARSING

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

- LL(1) PARSE TABLE FOR CALCULATOR LANGUAGE

READ A

READ B

SUM := A + B

WRITE SUM

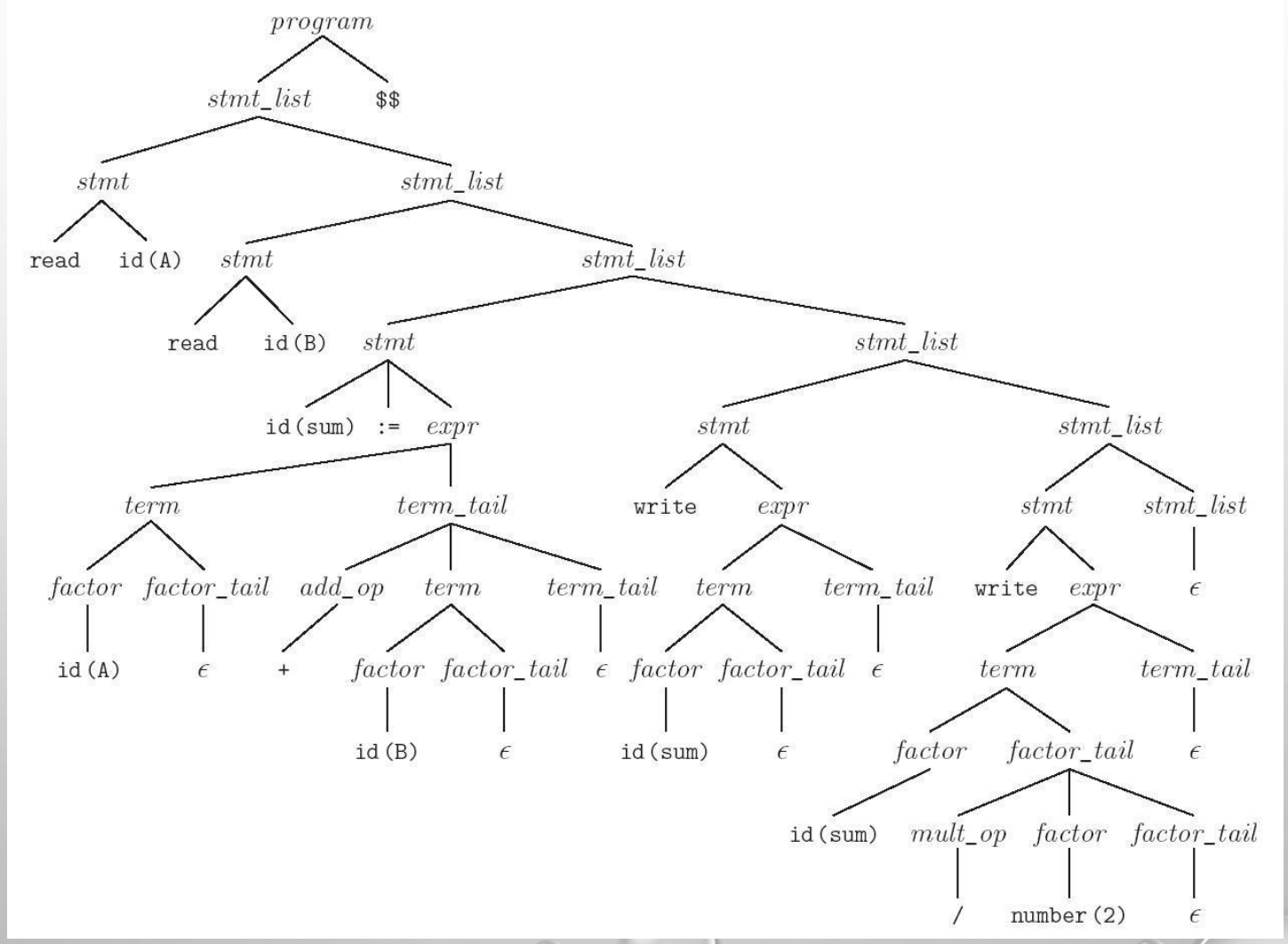
WRITE SUM / 2 $$

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* ⟶ / {/}

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	–

PARSE TREE FOR THE AVERAGE PROGRAM



LL PARSING

- PROBLEMS TRYING TO MAKE A GRAMMAR LL(1)
  - LEFT-RECURSION
    - EXAMPLE

$$\text{id\_list} \rightarrow \text{id} \mid \text{id\_list , id}$$

  - CAN REMOVE LEFT RECURSION MECHANICALLY

$$\begin{aligned} \text{id\_list} &\rightarrow \text{id id\_list\_tail} \\ \text{id\_list\_tail} &\rightarrow \text{, id id\_list\_tail} \mid \varepsilon \end{aligned}$$

LL PARSING

- MORE PROBLEMS
  - COMMON PREFIXES

$$\begin{aligned} stmt \rightarrow\ & id := expr \\ & \mid id\ (\ arg\_list\ ) \end{aligned}$$

  - CAN ELIMINATE COMMON FACTOR MECHANICALLY
    - ***LEFT-FACTORING***

$$\begin{aligned} stmt &\rightarrow id\ id\_stmt\_tail \\ id\_stmt\_tail &\rightarrow := expr \\ &\quad \mid (\ arg\_list\ ) \end{aligned}$$

LL PARSING

- ELIMINATING LEFT RECURSION AND COMMON PREFIXES DOES NOT MAKE A GRAMMAR LL
  - THERE ARE INFINITELY MANY NON-LL LANGUAGE
    - MECHANICAL TRANSFORMATIONS SHOWN WORK ON THEM JUST FINE
    - GRAMMAR IS STILL NOT LL(1)

LL PARSING

- MORE PROBLEMS MAKING GRAMMAR LL(1)
  - THE DANGLING-ELSE PROBLEM:
  - EXAMPLE (PASCAL): IF C1 THEN IF C2 THEN S1 ELSE S2
  - GRAMMAR:

$$
\begin{aligned}
\text{stmt} &\rightarrow \text{if cond then\_clause else\_clause} \\
&\quad \mid \text{other\_stuff} \\
\text{then\_clause} &\rightarrow \text{then stmt} \\
\text{else\_clause} &\rightarrow \text{else stmt} \mid \varepsilon
\end{aligned}
$$

  - THE ELSE CAN BE PAIRED WITH EITHER IF-THEN IN THE EXAMPLE!
  - FIX (DISAMBIGUATION RULE): PAIR AN ELSE WITH THE CLOSEST "ELSE-LESS" NESTED IF-THEN

LL PARSING

- A LESS NATURAL GRAMMAR FRAGMENT:

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

- BALANCED_STMT HAS EQUAL NUMBER OF *THENS* AND *ELSES*
- UNBALANCED_STMT HAS MORE *THENS*

LL PARSING

- USUAL APPROACH [REGARDLESS OF TOP-DOWN OR BOTTOM-UP]
  - USE AMBIGUOUS GRAMMAR
  - ADD DISAMBIGUATING RULES
    - I.E., *ELSE* GOES WITH CLOSEST UNMATCHED *THEN*
    - GENERALLY: THE FIRST OF TWO POSSIBLE PRODUCTIONS IS PREDICTED [OR SELECTED FOR REDUCTION]

LL PARSING

- NEWER LANGUAGES (SINCE PASCAL) INCLUDE EXPLICIT END MARKERS
- MODULA-2:

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

ADA USES END IF; SOME LANGUAGES USE FI

LL PARSING

- PROBLEM WITH END MARKERS: THEY ACCUMULATE

  IN PASCAL:

```
if A = B then …
  else if A = C then …
  else if A = D then …
  else if A = E then …
  else …;
```

  - WITH END MARKERS, THIS BECOMES:

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

LR PARSING

- LR PARSERS ALMOST ALWAYS TABLE-DRIVEN
  - LIKE A TABLE-DRIVEN LL PARSER, LR PARSER:
    - USES A LOOP
    - REPEATEDLY INSPECTS A TWO-DIMENSIONAL TABLE TO FIND NEXT ACTION
  - UNLIKE THE LL PARSER, LR DRIVER:
    - HAS NON-TRIVIAL STATE (LIKE A DFA)
    - HAS TABLE INDEXED BY CURRENT INPUT TOKEN & CURRENT STATE
  - STACK CONTAINS A RECORD OF WHAT HAS BEEN SEEN SO FAR (NOT WHAT IS EXPECTED)

LR PARSING

- A SCANNER IS A DFA
- AN LL OR LR PARSER IS A PDA [PUSH DOWN AUTOMATA]
  - EARLY'S & COCKE-YOUNGER-KASAMI ALGORITHMS DO NOT USE PDAS
  - A PDA CAN BE SPECIFIED WITH A STATE DIAGRAM AND A STACK
    - STATE DIAGRAM LOOKS JUST LIKE A DFA STATE DIAGRAM
      - EXCEPT ARCS ARE LABELED WITH <INPUT SYMBOL, TOP-OF-STACK SYMBOL> PAIRS
      - IN ADDITION TO MOVING TO A NEW STATE, PDA HAS THE OPTION OF PUSHING OR POPPING A FINITE NUMBER OF SYMBOLS ONTO/OFF THE STACK

ACTIONS

- WE CAN RUN ACTIONS WHEN A RULE TRIGGERS:
  - OFTEN USED TO CONSTRUCT AN AST (ABSTRACT SYNTAX TREE) FOR A COMPILER
  - FOR SIMPLE LANGUAGES, CAN INTERPRET CODE DIRECTLY
  - CAN USE ACTIONS TO FIX THE TOP-DOWN PARSING PROBLEMS

QUESTIONS