Syntax of switch in C (K&R)

In main text:

switch (*expression*) {
  case *const-expr*: *statements*
  case *const-expr*: *statements*
  default: *statements*
}

`Grammar Excerpt in Appendix:`

stmt => labeled stmt | selection stmt | …

selection stmt => … | switch (expression) stmt | …

labeled stmt => …|case const-expr: stmt | …

More C Syntax

- statement:

labeled-statement

expression-statement

compound-statement

selection-statement

iteration-statement

jump-statement

- labeled-statement:
  identifier `:` statement
  `case` constant-expression `:` statement
  `default` `:` statement
- expression-statement: expression$_{opt}$ `;`
- compound-statement:
  `{` declaration-list$_{opt}$ statement-list$_{opt}$ `}`
- statement-list:
  statement
  statement-list statement
- selection-statement:
  `if` (expression) statement
  `if` (expression) statement `else` statement
  `switch` (expression) statement
- iteration-statement:
  `while` (expression) statement
  `do` statement `while` (expression)`;`
  `for` (expression$_{opt}$`;` expression$_{opt}$`;` expression$_{opt}$) statement
- jump-statement:
  `goto` identifier`;`
  `continue;`
  `break;`
  `return` expression$_{opt}$`;`
- expression:
  assignment-expression
  expression `,` assignment-expression

K&R

- assignment-expression:
  conditional-expression
  unary-expression assignment-operator assignment-expression
- assignment-operator: one of
  `= *= /= %= += -= <<= >>= &= ^= |=`
- conditional-expression:
  logical-OR-expression
  logical-OR-expression `?` expression `:` conditional-expression
- constant-expression: conditional-expression
- logical-OR-expression:
  logical-AND-expression
  logical-OR-expression `||` logical-AND-expression
- logical-AND-expression:
  inclusive-OR-expression
  logical-AND-expression `&&` inclusive-OR-expression
- inclusive-OR-expression:
  exclusive-OR-expression
  inclusive-OR-expression `|` exclusive-OR-expression
- exclusive-OR-expression:
  AND-expression
  exclusive-OR-expression `^` AND-expression

...

- AND-expression:
  equality-expression
  AND-expression `&` equality-expression
- equality-expression:
  relational-expression
  equality-expression `==` relational-expression equality-expression `!=` relational-expression
- relational-expression:
  shift-expression
  relational-expression `<` shift-expression
  relational-expression `>` shift-expression
  relational-expression `<=` shift-expression
  relational-expression `>=` shift-expression
- shift-expression:
  additive-expression
  shift-expression `<<` additive-expression
  shift-expression `>>` additive-expression
- additive-expression:
  multiplicative-expression
  additive-expression `+` multiplicative-expression
  additive-expression `-` multiplicative-expression
- multiplicative-expression:
  multiplicative-expression `*` cast-expression
  multiplicative-expression `/` cast-expression
  multiplicative-expression `%` cast-expression

C Compiler accepts “wrong” code?

```
// switch.c
int main() {
 switch(1) { if (2) 3+2; else 1+1;};
 return(1);
}
```

```
cc -S -save-temps switch.c
switch.c:4:10: warning: no case matching constant switch condition '1'
  switch(1) { if (2) 3+2; else 1+1;};
         ^
1 warning generated.
```

```
        .section        __TEXT,__text,regular,pure_instructions
        .build_version macos, 12, 0     sdk_version 13, 1
        .globl  _main                           ; -- Begin
function main
        .p2align        2
_main:                                  ; @main
        .cfi_startproc
; %bb.0:
        sub     sp, sp, #16
        .cfi_def_cfa_offset 16
        str     wzr, [sp, #12]
        mov     w0, #1
        add     sp, sp, #16
        ret
        .cfi_endproc
                                        ; -- End function
.subsections_via_symbols
```

Language, Grammar, Derivations, Recognizers

- $a^n b^n$
- $a^n b^n c^n$
- $a^n b^n c^n d^n$
- CFL
  - Recursive descent parsers
  - SLR(k) with k lookahead tokens, k=0, 1, …
  - LL(k), k=0, 1, …
    - Left to Right, Leftmost derivation (“left recursive”)
  - LR(k), k=0,1,…
    - Left to Right, Rightmost derivation (“right recursive”)
  - LALR(1) same as LR(0) with 1 lookahead token
    - “optimized” LR(0)

In a **leftmost derivation**, at each step, the leftmost nonterminal is replaced.
In a **rightmost derivation**, at each step, the rightmost nonterminal is replaced.

Dragon book

Leftmost derivation parse

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

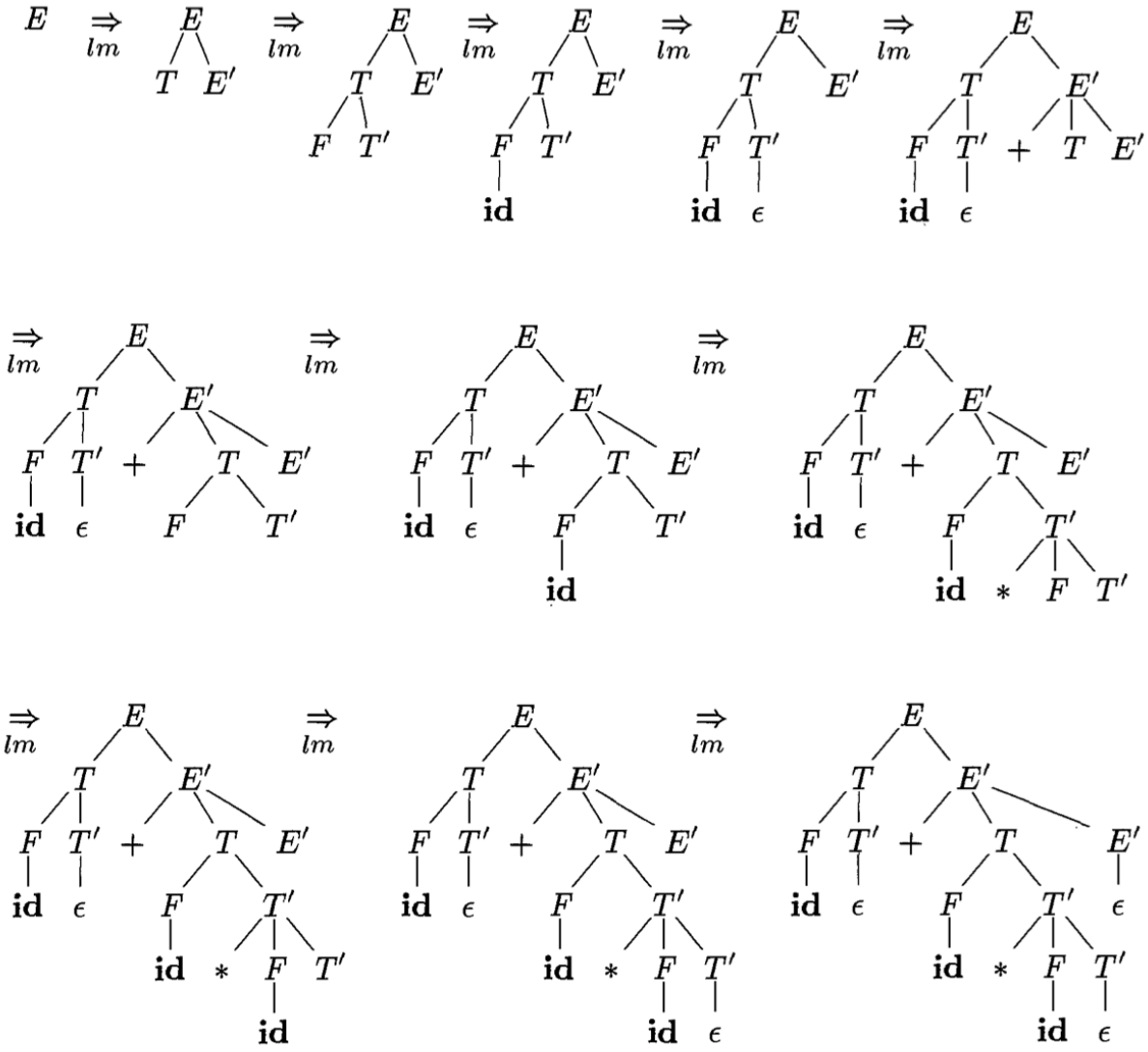

Figure 4.12: Top-down parse for $\mathbf{id} + \mathbf{id} * \mathbf{id}$

Parsing table

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		