CA4003 - Compiler Construction

Top-down Parsing

David Sinclair

Top-down Parsing

A *top-down parser* starts with the root of the parse tree, labelled with the goal symbol of the grammar, and repeats the following steps until the fringe of the parse tree matches the input string.

1. At a node labelled A, select a production $A \rightarrow \alpha$ and construct the appropriate child for each symbol of α.
2. When a terminal is added to the fringe that doesnt match the input string, backtrack.
3. Find the next node to be expanded (must have a label in V_n)

The key is selecting the right production in step 1

Example

Recall the simple expressions grammar:

1	`<goal>`	::=	`<expr>`
2	`<expr>`	::=	`<expr>` + `<term>`
3		|	`<expr>` - `<term>`
4		|	`<term>`
5	`<term>`	::=	`<term>` * `<factor>`
6		|	`<term>` / `<factor>`
7		|	`<factor>`
8	`<factor>`	::=	num
9		|	id

Consider the input string `x - 2 * y`

Example [2]

Prod'n	Sentential form	Input
–	`<goal>`	$\uparrow x \; - 2 * y$
1	`<expr>`	$\uparrow x \; - 2 * y$
2	`<expr>` + `<term>`	$\uparrow x \; - 2 * y$
4	`<term>` + `<term>`	$\uparrow x \; - 2 * y$
7	`<factor>` + `<term>`	$\uparrow x \; - 2 * y$
9	id + `<term>`	$\uparrow x \; - 2 * y$
–	id + `<term>`	$x \uparrow - 2 * y$
–	`<expr>`	$\uparrow x \; - 2 * y$
3	`<expr>` - `<term>`	$\uparrow x \; - 2 * y$
4	`<term>` - `<term>`	$\uparrow x \; - 2 * y$
7	`<factor>` - `<term>`	$\uparrow x \; - 2 * y$
9	id - `<term>`	$\uparrow x \; - 2 * y$
–	id - `<term>`	$x \uparrow - 2 * y$

Example [3]

Prod'n	Sentential form	Input
–	`id - <term>`	$x - \uparrow 2 * y$
7	`id - <factor>`	$x - \uparrow 2 * y$
8	`id - num`	$x - \uparrow 2 * y$
–	`id - num`	$x - 2 \uparrow * y$
–	`id - <term>`	$x - \uparrow 2 * y$
5	`id - <term> * <factor>`	$x - \uparrow 2 * y$
7	`id - <factor> * <factor>`	$x - \uparrow 2 * y$
8	`id - num * <factor>`	$x - \uparrow 2 * y$
–	`id - num * <factor>`	$x - 2 \uparrow * y$
–	`id - num * <factor>`	$x - 2 * \uparrow y$
9	`id - num * id`	$x - 2 * \uparrow y$
–	`id - num * id`	$x - 2 * y \uparrow$

To avoid backtracking the parse should be guided by the input string.

Example [4]

Another possible parse for `x - 2 * y`

Prod'n	Sentential form	Input
–	`<goal>`	$\uparrow x - 2 * y$
1	`<expr>`	$\uparrow x - 2 * y$
2	`<expr>` + `<term>`	$\uparrow x - 2 * y$
2	`<expr>` + `<term>` + `<term>`	$\uparrow x - 2 * y$
2	`<expr>` + `<term>` + …	$\uparrow x - 2 * y$
2	`<expr>` + `<term>` + …	$\uparrow x - 2 * y$
2	…	$\uparrow x - 2 * y$

If the parser makes wrong choices, expansion doesn't terminate.

This must be avoided!

Left-recursion

Top-down parsers **cannot** handle left-recursion in a grammar.

Formally, a grammar is *left-recursive* if
$\exists A \in V_n$ such that $A \Rightarrow^+ A\alpha$ for some string α.

Our simple expression grammar is left-recursive.

Eliminating Left-recursion

To remove left-recursion, we can transform the grammar.

Consider the grammar fragment:

$$\begin{array}{lcl} A & ::= & A\alpha \\ & \mid & \beta \end{array}$$

where α and β do not start with A.
We can rewrite this as:

$$\begin{array}{lcl} A & ::= & \beta A' \\ A' & ::= & \alpha A' \\ & \mid & \epsilon \end{array}$$

where A' is a new non-terminal

This fragment contains no left-recursion.

Lookahead

Picking the "right" production rule reduces the amount of backtracking.
Picking the "wrong" production rule may result in the derivation not terminating.
By looking ahead into the input string we can use the information on the upcoming tokens to select the "right" production rule.
In general we need arbitrary lookahead to parse a CFG but there is a large number of subclasses of CFGs, including most programming language constructs, that can be parsed with limited lookahead.
Among the interesting subclasses are:

- LL(1): Left to right scan, Left-most derivation, 1- token lookahead
- LR(1): Left to right scan, Right-most derivation, 1-token lookahead

Predictive Parsing

For any two productions $A \rightarrow \alpha | \beta$, we would like a distinct way of choosing the correct production to expand.
For some $\texttt{RHS}\ \alpha \in G$, define $\texttt{FIRST}(\alpha)$ as the set of tokens that appear first in some string derived from α
That is, for some $w \in V_t^*$, $w \in \texttt{FIRST}(\alpha)$ iff. $\alpha \Rightarrow^* w\gamma$.

Key property:
Whenever two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like

$$\texttt{FIRST}(\alpha) \cap \texttt{FIRST}(\beta) = \emptyset$$

This would allow the parser to make a correct choice with a lookahead of only one symbol!

The example expression grammar has this property!

Left-factoring

What if a grammar does not have this key property?

Sometimes, we can transform a grammar to have this property.

For each non-terminal A find the longest prefix α common to two or more of its alternatives.

if $\alpha \neq \epsilon$ then replace all of the A productions
$A \rightarrow \alpha\beta_1|\alpha\beta_2|\ldots|\alpha\beta_n$
with
$A \rightarrow \alpha A'$
$A' \rightarrow \beta_1|\beta_2|\ldots|\beta_n$
where A' is a new non-terminal.

Repeat until no two alternatives for a single non-terminal have a common prefix.

Left-factoring Example

The following is a *right-recursive* version of the expression grammar that is *right-associative*:

1	`<goal>`	::=	`<expr>`
2	`<expr>`	::=	`<term>` + `<expr>`
3		|	`<term>` - `<expr>`
4		|	`<term>`
5	`<term>`	::=	`<factor>` * `<term>`
6		|	`<factor>` / `<term>`
7		|	`<factor>`
8	`<factor>`	::=	`num`
9		|	`id`

To choose between productions 2, 3, & 4, the parser must look beyond the `num` or `id` and look at the next token (+, -, * or /).

$$\texttt{FIRST}(2) \cap \texttt{FIRST}(3) \cap \texttt{FIRST}(4) \neq \emptyset$$

Left-factoring Example [2]

There are two nonterminals, <expr> and <term> that must be left factored:

<expr>	::=	<term> + <expr>	<term>	::=	<factor> * <term>
	|	<term> - <expr>		|	<factor> / <term>
	|	<term>		|	<factor>

Applying the transformation gives us:

<expr>	::=	<term> <expr'>	<term>	::=	<factor> <term'>
<expr'>	::=	+ <expr>	<term'>	::=	* <term>
	|	- <expr>		|	/ <term>
	|	ϵ		|	ϵ

Left-factoring Example [3]

Substituting back into the grammar yields:

1	<goal>	::=	<expr>
2	<expr>	::=	<term> <expr'>
3	<expr'>	::=	+ <expr>
4		|	- <expr>
5		|	ϵ
6	<term>	::=	<factor> <term'>
7	<term'>	::=	* <term>
8		|	/ <term>
9		|	ϵ
10	<factor>	::=	num
11		|	id

Now, selection requires only a single token lookahead, but it is still right-associative.

Left-recursion elimination after Left-factoring

Given a left-factored CFG, to eliminate left-recursion:

if $\exists A \rightarrow A\alpha$ then replace all of the A productions
$A \rightarrow A\alpha|\beta|\ldots|\gamma$
with
$A \rightarrow NA'$
$N \rightarrow \beta|\ldots|\gamma$
$A' \rightarrow \alpha A'|\epsilon$
where N and A' are new productions.

Repeat until there are no left-recursive productions.

Table-Driven Parsing

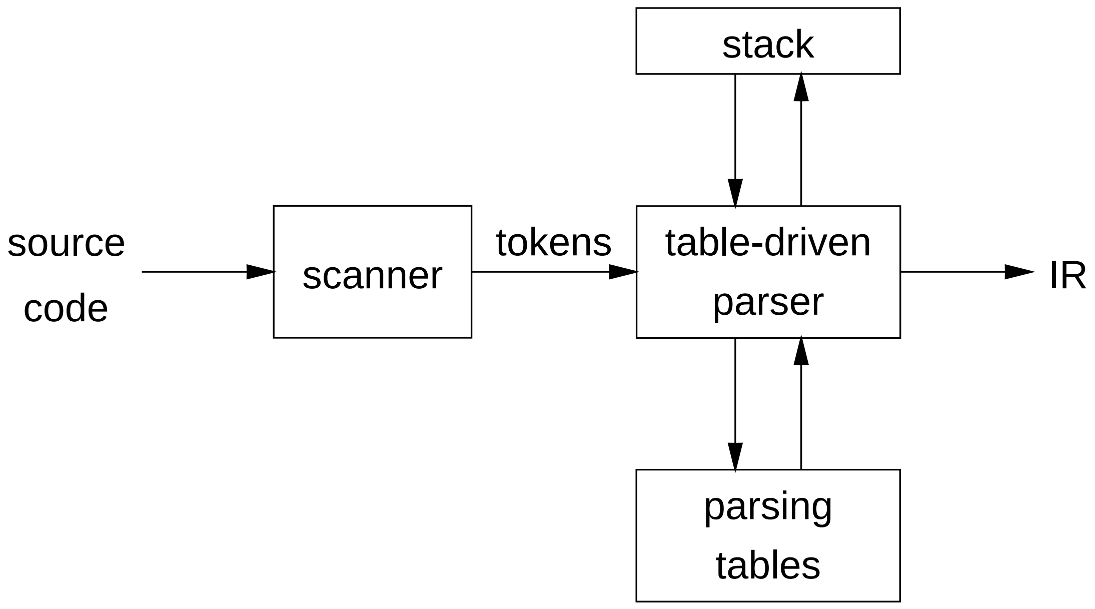

Table-Driven Parsing [2]

The generation of the tables can be automated.

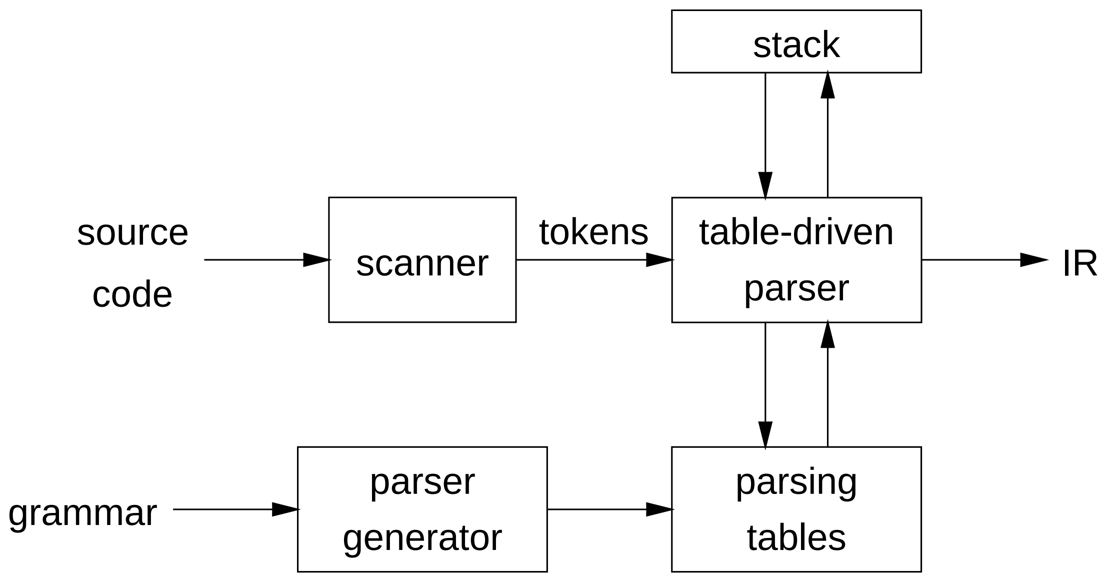

Table-Driven Parsing [3]

Input: a string w and a parsing table M for G.

```
tos = 0
Stack[tos] = EOF
Stack[++tos] = Start Symbol
token = next_token()
repeat
  X = Stack[tos]
  if X is a terminal or EOF then
    if X == token then
      pop X
      token = next_token()
    else error()
  else /* X is a non-terminal */
    if $M$[X,token] == $X \to Y_1 Y_2 ... Y_k$ then
      pop X
      push $Y_k, Y_{k-1}, ..., Y_1$
    else error()
until X == EOF
```

Table-Driven Parsing [4]

For the expression grammar:

1	`<goal>`	::=	`<expr>`
2	`<expr>`	::=	`<term> <expr'>`
3	`<expr'>`	::=	+ `<expr>`
4		|	- `<expr>`
5		|	ϵ
6	`<term>`	::=	`<factor> <term'>`
7	`<term'>`	::=	* `<term>`
8		|	/ `<term>`
9		|	ϵ
10	`<factor>`	::=	num
11		|	id

Its parse table:

	id	num	+	-	*	/	$
`<goal>`	1	1	–	–	–	–	–
`<expr>`	2	2	–	–	–	–	–
`<expr'>`	–	–	3	4	–	–	5
`<term>`	6	6	–	–	–	–	–
`<term'>`	–	–	9	9	7	8	9
`<factor>`	11	10	–	–	–	–	–

How do we generate this table?

FIRST

For a string of grammar symbols α, we define FIRST(α) as:

- the set of terminal symbols that begin strings derived from α: $\{a \in V_t | \alpha \Rightarrow^* a\beta\}$
- If $\alpha \Rightarrow^* \epsilon$ then $\epsilon \in \texttt{FIRST}(\alpha)$

Consider the following grammar:

$Z \rightarrow d$ $\quad Y \rightarrow$ $\quad X \rightarrow Y$

$Z \rightarrow XYZ$ $\quad Y \rightarrow c$ $\quad X \rightarrow a$

Then FIRST(XYZ) = $\{a, c, d\}$

Symbols that can derive the empty string,ϵ, are called *nullable* and we must keep track of what can follow a *nullable* symbol.

FIRST [2]

To compute FIRST(X) for all grammar symbol X, apply the following rules until no more terminals or ϵ can be added to any FIRST set:

1. If X is a terminal, then FIRST$(X) = \{X\}$
2. If X is a nonterminal and $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production rule for some $k \geq 1$, then add a to FIRST(X) if for some i, $a \in$ FIRST(Y_i) and $Y_1 \dots Y_{i-1}$ are *nullable*. If all $Y_1 \dots Y_k$ are *nullable*, add ϵ to FIRST(X).
3. If $X \rightarrow \epsilon$ is a production rule, then add ϵ to FIRST(X).

FOLLOW

FOLLOW(X) is the set of terminals that can immediately follow X. So $t \in$ FOLLOW(X) if there is any derivation containing Xt. This would include any derivation containing $XYZt$ where Y and Z are *nullable*.
To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S) where S is the start symbol and \$ in the end of input marker.
2. If there is a production rule $A \rightarrow \alpha B \beta$, then everything in FIRST(β), except ϵ, is in FOLLOW(B).
3. If there is a production rule $A \rightarrow \alpha B$, or a production rule $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

Example

Consider the following grammar where E is the start symbol.

$E \rightarrow TE'$	$T' \rightarrow *FT'$
$E' \rightarrow +TE'$	$T' \rightarrow$
$E' \rightarrow$	$F \rightarrow (E)$
$T \rightarrow FT'$	$F \rightarrow \textbf{id}$

$$\begin{aligned}
\texttt{FIRST}(F) &= \{\ (,\ \textbf{id}\} \\
\texttt{FIRST}(T) &= \texttt{FIRST}(F) \\
&= \{\ (,\ \textbf{id}\} \\
\texttt{FIRST}(E) &= \texttt{FIRST}(T) \\
&= \{\ (,\ \textbf{id}\} \\
\texttt{FIRST}(E') &= \{\ +,\ \epsilon\} \\
\texttt{FIRST}(T') &= \{\ *,\ \epsilon\}
\end{aligned}$$

$$\begin{aligned}
\texttt{FOLLOW}(E) &= \{\ ),\ \$\} \\
\texttt{FOLLOW}(E') &= \texttt{FOLLOW}(E) \\
&= \{\ ),\ \$\} \\
\texttt{FOLLOW}(T) &= \{\ +,\ ),\ \$\} \\
\texttt{FOLLOW}(T') &= \texttt{FOLLOW}(T) \\
&= \{+,\ ),\ \$\} \\
\texttt{FOLLOW}(F) &= \{+,\ *,\ ),\ \$\}
\end{aligned}$$

Example 2

We did the last example by inspection, but we could have also done it by iteratively applying the rule.

Consider the following grammar (again):

$Z \rightarrow d$	$Y \rightarrow$	$X \rightarrow Y$
$Z \rightarrow XYZ$	$Y \rightarrow c$	$X \rightarrow a$

We start with all nonterminals not nullable and initially empty FIRST and FOLLOW sets.

	nullable	FIRST	FOLLOW
X	no		
Y	no		
Z	no		

Example 2 [2]

In the first iteration:

	nullable	FIRST	FOLLOW
X	no	a	c, d
Y	yes	c	d
Z	no	d	

In the second iteration:

	nullable	FIRST	FOLLOW
X	yes	a, c	a, c, d
Y	yes	c	a, c, d
Z	no	a, c, d	

A third iteration finds no new information.

LOOKAHEAD

For a production rule $A \rightarrow \alpha$, we define $\texttt{LOOKAHEAD}(A \rightarrow \alpha)$ as the set of terminals which can appear next in the input when recognising production rule $A \rightarrow \alpha$.
Thus, a production rule's `LOOKAHEAD` set specifies the tokens which should appear next in the input before the production rule is applied.
To build $\texttt{LOOKAHEAD}(A \rightarrow \alpha)$:

1. Put $\texttt{FIRST}(\alpha)$ - $\{\epsilon\}$ in $\texttt{LOOKAHEAD}(A \rightarrow \alpha)$.
2. If $\epsilon \in \texttt{FIRST}(\alpha)$
   then put $\texttt{FOLLOW}(A)$ in $\texttt{LOOKAHEAD}(A \rightarrow \alpha)$.

A grammar G is LL(1) *iff* for each set of productions
$A \rightarrow \alpha_1|\alpha_2|...|\alpha_n$:
$\texttt{LOOKAHEAD}(A \rightarrow \alpha_1)$,$\texttt{LOOKAHEAD}(A \rightarrow \alpha_2)$,...,
$\texttt{LOOKAHEAD}(A \rightarrow \alpha_n)$ are all pairwise disjoint.

Example 3

$$
\begin{array}{l|l}
S \rightarrow E & T \rightarrow FT' \\
E \rightarrow TE' & T' \rightarrow *T \mid /T \mid \epsilon \\
E' \rightarrow +E \mid -E \mid \epsilon & F \rightarrow \text{id} \mid \text{num}
\end{array}
$$

	FIRST	FOLLOW
S	{num, id}	{\$}
E	{num, id}	{\$}
E'	$\{\epsilon, +, -\}$	{\$}
T	{num, id}	{+,-,\$}
T'	$\{\epsilon, *, /\}$	{+,-,\$}
F	{num, id}	{+,-,*,/,\$}
id	{id}	–
num	{num}	–
*	{*}	–
/	{/}	–
+	{+}	–
-	{-}	–

	LOOKAHEAD
$S \rightarrow E$	{num, id}
$E \rightarrow TE'$	{num, id}
$E' \rightarrow +E$	{+}
$E' \rightarrow -E$	{-}
$E' \rightarrow \epsilon$	{\$}
$T \rightarrow FT'$	{num, id}
$T' \rightarrow *T$	{*}
$T' \rightarrow /T$	{/}
$T' \rightarrow \epsilon$	{+,-,\$}
$F \rightarrow$ id	{id}
$F \rightarrow$ num	{num}

LL(1) parse table construction

Input: Grammar G

Output: Parsing table M

Method:

1. $\forall$ productions $A \rightarrow \alpha$:
   $\forall a \in \texttt{LOOKAHEAD}(A \rightarrow \alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$
2. Set each undefined entry of M to **error**

If $\exists M[A, a]$ with multiple entries then grammar is not LL(1).

Example 3 - again

	LOOKAHEAD
$S \longrightarrow E$	{num, id}
$E \longrightarrow TE'$	{num, id}
$E' \longrightarrow +E$	{+}
$E' \longrightarrow -E$	{-}
$E' \longrightarrow \epsilon$	{$}
$T \longrightarrow FT'$	{num, id}
$T' \longrightarrow *T$	{*}
$T' \longrightarrow /T$	{/}
$T' \longrightarrow \epsilon$	{+,-,$}
$F \longrightarrow$ id	{id}
$F \longrightarrow$ num	{num}

	id	num	+	-	*	/	$
S	$S \rightarrow E$	$S \rightarrow E$	–	–	–	–	–
E	$E \rightarrow TE'$	$E \rightarrow TE'$	–	–	–	–	–
E'	–	–	$E' \rightarrow +E$	$E' \rightarrow -E$	–	–	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	$T \rightarrow FT'$	–	–	–	–	–
T'	–	–	$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$	$T' \rightarrow *T$	$T' \rightarrow /T$	$T' \rightarrow \epsilon$
F	$F \rightarrow$ id	$F \rightarrow$ num	–	–	–	–	–

Building the Abstract Syntax Tree

Again, we insert code at the right points.

```
tos = 0
Stack[tos] = EOF
Stack[++tos] = root node
Stack[++tos] = Start Symbol
token = next_token()
repeat
  X = Stack[tos]
  if X is a terminal or EOF then
    if X == token then
      pop X
      token = next_token()
        pop and fill in node
    else error()
  else /* X is a non-terminal */
    if $M[X, token] == X \rightarrow Y_1 Y_2 \dots Y_k$ then
      pop X
        pop node for X
        build node for each child and
        make it a child of node for X
      push $\{n_k, Y_k, n_{k-1}, Y_{k-1}, \dots, n_1, Y_1\}$
    else error()
until X == EOF
```

Some facts about LL(1) grammars

Provable facts about LL(1) grammars:

1. No left-recursive grammar is LL(1).
2. No ambiguous grammar is LL(1).
3. Some languages have no LL(1) grammar.
4. A ϵ–free grammar where each alternative expansion for A begins with a distinct terminal is a *simple* LL(1) grammar.

Example:

$$S \longrightarrow aS|a$$

is not LL(1) because

$\texttt{LOOKAHEAD}(S \longrightarrow aS) = \texttt{LOOKAHEAD}(S \longrightarrow a) = \{a\}$

$$S \longrightarrow aS'$$
$$S' \longrightarrow aS'|\epsilon$$

accepts the same language and is LL(1).

Error Recovery

Key notion:

- For each non-terminal, construct a set of terminals on which the parser can synchronize.
- When an error occurs looking for A, scan until an element of $\texttt{SYNCH}(A)$ is found.

Building $\texttt{SYNCH}$:

1. $a \in \texttt{FOLLOW}(A) \Rightarrow a \in \texttt{SYNCH}(A)$.
2. place keywords that start statements in $\texttt{SYNCH}(A)$.
3. add symbols in $\texttt{FIRST}(A)$ to $\texttt{SYNCH}(A)$.

If we can't match a terminal on top of stack:

1. pop the terminal,
2. print a message saying the terminal was inserted, and
3. continue the parse.

(i.e., $\texttt{SYNCH}(a) = V_t - \{a\}$)

A grammar that is not LL(1)

```
<stmt>  ::=  if <expr> then <stmt>
          |  if <expr> then <stmt> else <stmt>
          |  ...
```

Left-factored:

```
<stmt>   ::=  if <expr> then <stmt> <stmt'> | ...
<stmt'>  ::=  else <stmt> | ϵ
```

FIRST(<stmt′>) = {ϵ,else}
FOLLOW(<stmt′>) = {else,\$}
LOOKAHEAD(<stmt′> ::= else <stmt>) = {else}
LOOKAHEAD(<stmt′> ::= ϵ) = {else,\$}

A grammar that is not LL(1) [2]

On seeing else, conflict between choosing
<stmt′> ::= else <stmt> and <stmt′> ::= ϵ

$\Rightarrow$ grammar is not LL(1)!

The fix:
Put priority on <stmt′> ::= else <stmt> to associate else with closest previous then.